Utilizing Runtime Information for Accurate Root Cause Identification in Performance Diagnosis

Lingmei Weng

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2023

Abstract

Utilizing Runtime Information for Accurate Root Cause Identification in Performance Diagnosis

Lingmei Weng

This dissertation highlights that existing performance diagnostic tools often become less effective due to their inherent inaccuracies in modern software. To overcome these inaccuracies and effectively identify the root causes of performance issues, it is necessary to incorporate supplementary runtime information into these tools. Within this context, the dissertation integrates specific runtime information into two typical performance diagnostic tools: profilers and causal tracing tools. The integration yields a substantial enhancement in the effectiveness of performance diagnosis.

Among these tools, `gprof` stands out as a representative profiler for performance diagnosis. Nonetheless, its effectiveness diminishes as the time cost calculated based on CPU sampling fails to accurately and adequately pinpoint the root causes of performance issues in complex software. To tackle this challenge, the dissertation introduces an innovative methodology called *value-assisted cost profiling* (vProf). This approach incorporates variable values observed during runtime into the profiling process. By continuously sampling variable values from both normal and problematic executions, vProf refines function cost estimates, identifies anomalies in value distributions, and highlights potentially problematic code areas that could be the actual sources of performance issues. The effectiveness of vProf is validated through the diagnosis of 18 real-world performance issues in four widely-used applications. Remarkably, vProf outperforms other state-of-the-art tools,

successfully diagnosing all issues, including three that had remained unresolved for over four years.

Causal tracing tools reveal the root causes of performance issues in complex software by generating tracing graphs. However, these graphs often suffer from inherent inaccuracies, characterized by superfluous (over-connected) and missed (under-connected) edges. These inaccuracies arise from the diversity of programming paradigms. To mitigate the inaccuracies, the dissertation proposes an approach to derive *strong* and *weak* edges in tracing graphs based on the vertices' semantics collected during runtime. By leveraging these edge types, a beam-search-based diagnostic algorithm is employed to identify the most probable causal paths. Causal paths from normal and buggy executions are differentiated to provide key insights into the root causes of performance issues. To validate this approach, a causal tracing tool named Argus is developed and tested across multiple versions of macOS. It is evaluated on 12 well-known spinning pinwheel issues in popular macOS applications. Notably, Argus successfully diagnoses the root causes of all identified issues, including 10 issues that had remained unresolved for several years.

The results from both tools exemplify a substantial enhancement of performance diagnostic tools achieved by harnessing runtime information. The integration can effectively mitigate inherent inaccuracies, lend support to inaccuracy-tolerant diagnostic algorithms, and provide key insights to pinpoint the root causes.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This work would not have been possible without assistance, guidance, and encouragement from many people. I express my heartfelt gratitude to all the individuals who have supported and contributed to the completion of this dissertation.

First, I sincerely appreciate my advisors, Professor Jason Nieh and Professor Junfeng Yang, for their inexhaustible support and expertise. Their guidance and constructive feedback have been invaluable in shaping the direction and quality of this dissertation. My sincere appreciation also goes to Professor Peng Huang for his relentless assistance and mentorship throughout this research journey.

I am also indebted to the members of my dissertation committee for their insightful comments and feedback. Their expertise and scholarly input have been instrumental in improving the overall quality of this dissertation.

I would like to express my gratitude to my family and friends. Their love, understanding, and patience have been my constant inspiration, and I am grateful for their presence in my life.

Last but not least, I would like to acknowledge the assistance and encouragement from Yang Tang, Gang Hu, Xinhao Yuan, Rui Gu, Jeremy Andrus, Yigong Hu, and David Williams-King. They helped me solve technical issues and proofread my manuscripts.

Dedication

to my husband, parents, and siblings

Your constant love and unwavering support have given me the strength to go through this journey.

Chapter 1: Introduction

1.1 Thesis Statement

Existing performance diagnostic tools often become less effective due to their inherent inaccuracies in modern software. To overcome these inaccuracies and effectively identify the root causes of performance issues, it is necessary to incorporate supplementary runtime information into these tools.

1.2 Extensive Efforts in Performance Diagnosis

Performance issues in software can result in substantial service disruptions and financial losses due to their non-fail-stop nature. For instance, Google has indicated that a 2% increase in latency could lead to a 2% decrease in search activity and user engagement [52]. A survey [64] also revealed negative business impacts of the performance challenges faced by Office365[53], including heightened user frustration, increased demands for IT service and support, and decreased employee productivity. Even a minor friction issue within an application can lead to reduced user engagement and prompt users to seek alternative products.

To assist developers, numerous powerful tools have been developed to ensure optimal performance throughout the software's lifecycle. Before release, many software products undergo load testing, during which testing utilities simulate high levels of user traffic to assess performance and identify bottlenecks. After release, developers often use profiling, tracing, and debugging tools to diagnose performance issues.

Many profiling tools [27, 28, 43, 34, 35] have been developed and widely used in the real world. They collect resource usage data of a program during runtime, including CPU utilization, memory consumption, and I/O operations. The collected data is then analyzed to identify per-

formance hotspots, which are the code areas that consume excessive resources. Once identified, optimizations can be applied to the code to improve performance by reducing unnecessary computations, memory usage, or I/O operations.

While these profilers are effective in diagnosing performance issues where hot paths are the culprits, they often lack the information necessary for developers to investigate causal relationships in complex performance issues involving many components and high concurrency. In response to these intricate performance challenges, researchers have proposed causal tracing, a method that identifies the step-by-step workflow leading to performance issues.

Causal tracing uses a tracing tool, such as DTrace [12] or SystemTap [36], to capture runtime information about user inputs, system calls, function calls, and interprocess communications during program execution. By analyzing the tracing data, developers can construct a dependency graph that illustrates the workflow of request handling across threads, processes, and even machines. Subsequently, developers can examine critical paths in the graphs and pinpoint the root causes of performance issues.

Despite decades of research efforts dedicated to automating the diagnosis of performance issues and substantial advancements in diagnostic tools, well-tested software like MariaDB [51] continues to encounter at least five performance issues each month, some of which can remain unresolved for years.

1.3 Ineffectiveness of Existing Performance Diagnostic Tools

Developers are aware that existing diagnostic tools can yield inaccurate results, potentially leading to erroneous conclusions. Nonetheless, there has been insufficient attention towards identifying the specific inaccuracies that can mislead the diagnosis. Moreover, there has been limited research exploring the integration of supplementary runtime information as a viable approach to mitigate these challenges.

Inaccuracies in Profilers Profiling tools are commonly recommended to identify performance bottlenecks in software, particularly in cases involving resource-intensive operations. Profilers can provide valuable insights to developers by highlighting the hot code paths. However, the functions reported by profilers do not consistently and precisely uncover the root causes of specific performance issues, leading to speculative guesses.

To better understand the inaccuracies of a profiler and potential solutions to overcome them, we studied `gprof` [28] using a bug from MariaDB [11]. This bug manifested during database recovery in MariaDB, resulting in unacceptably long execution times and elevated CPU usage. Upon analyzing the profiling results from `gprof`, we observed several sources of inherent inaccuracies.

Firstly, the reported hot functions can be inherently expensive and are often already optimized. They may not necessarily be the primary culprits behind performance issues. For instance, in the MariaDB case, `gprof` identified the function applying hashed records from recovery log as the culprit. However, this information proved unhelpful since the function was inherently costly. It also incurred significant costs during normal recovery processes.

Secondly, the function cost can be inaccurate due to the CPU sampling bias. The statistical histogram does not include PC samples that fall outside the text section of the application. If an inefficient function frequently calls expensive functions from dynamic libraries, the cost of the inefficient function will be underestimated.

As an illustration, in the MariaDB case, the root cause was ranked 454th in the profiling results. This discrepancy arose because `gprof` solely counted the profiling samples within the function, overlooking the significant time cost incurred through function calls.

Thirdly, the function rankings provided by profilers offer limited insights for in-depth debugging. Consequently, even when profilers accurately identify the root cause functions, developers often find themselves compelled to undertake supplementary actions, such as recording relevant variable values, to unearth the root causes of the performance issue.

Based on the communications in the bug report of the MariaDB case, we observed that although senior developers were able to identify the excessive invocations of the costly function,

they struggled to determine the root causes due to the lack of data flow information.

Inaccuracies in Causal Tracing Tools Causal tracing tools emerge as a solution for diagnosing performance issues in scenarios involving multiple interconnected components. This necessity arises from the limitations of conventional profilers, which are incapable of revealing causal relationships that span across component boundaries. However, causal tracing rules, derived from prior research, often fall short in providing accurate causal insights for performance issues within modern desktop applications.

To illustrate this challenge, consider a performance issue observed in Chrome [71] on macOS when users input non-English characters into a search bar on some webpages. This action led to a stall in the Chrome UI thread, rendering the application unresponsive to user inputs. The intricacy of this performance issue involved various daemons, helper tools, the browser itself, and the renderer processes within Chrome. Upon closer investigation, the root cause was traced to the main UI thread of the browser, which was repeatedly waiting for character rendering within the renderer. This waiting process continually timed out due to a dominant JavaScript request within the renderer's main thread.

In this specific scenario, conventional profilers provided limited information, indicating that the `runloop` within the UI thread consumed a significant portion of processing time, yet failing to offer insights into the underlying causes. In contrast, causal paths derived from systemwide tracing data showed considerable potential in revealing intricate interdependencies that spanned across multiple components. Therefore, we built a causal tracing tool for macOS, drawing upon prior research [15, 6, 63, 72, 26, 68, 40, 49, 67, 50, 60, 80] in distributed systems and mobile applications. As we attempted to diagnose the Chrome case, several inherent inaccuracies were identified in the causal tracing tool.

Firstly, the delineation of execution boundaries for a particular request were inaccurate. Accurately identifying the point at which a thread transitioned to the execution of a different request was hard from the tracing log. This difficulty stemmed from the adoption of customized programming paradigms, such as batch processing, in modern desktop applications for efficiency.

For example, the system daemon `fontd` [48], responsible for providing font sizes to applications, applied batch processing in its message-handling function. While processing messages from the browser process in Chrome, it continuously accepted messages from other applications throughout the system. Following the existing causal tracing rules, all activities occurring between the receipt and reply of a message were associated with the same request. Consequently, incorrect connections were established between messages from Chrome and those from other applications.

Secondly, system level activities, such as interrupt handling, often intermingled with user space activities in the tracing log, leading to false connections. This interference introduced noise into causal paths, making it challenging to discern causalities for a specific performance issue within complex workflows.

Thirdly, causal tracing tools were built under the assumption that one thread signaling another implied a causal relationship between the two threads. However, such signals did not invariably indicate causalities, resulting in an excess of connections in the trace graphs.

Lastly, existing causal tracing tools failed to capture connections arising from data dependencies and shared data flags, rendering the trace graphs incomplete as a consequence.

Sources of Inherent Inaccuracies Investigating the performance issues at hand, we summarized the sources of inherent inaccuracies in existing diagnostic tools.

Firstly, in an era of escalating software complexity, the information collected by these diagnostic tools becomes insufficient for effectively diagnosing performance issues. Profilers, for example, may inadvertently overlook inefficient functions due to their profiling strategies. Causal tracing tools may struggle to capture the entirety of connections across diverse programming paradigms. Furthermore, adapting to variations in software implementation poses a challenge, invariably leading to high overhead and consequently, distortion of the system's behavior.

Secondly, the information collected inherently contains noise. This is because complex software systems often present intricate performance issues that manifest solely when multiple concurrent requests are active within the system.

Thirdly, it is worth noting that applying diagnostic algorithms directly to analyze inaccurate

runtime information can be misleading. Profiling tools, for instance, highlight functions based on their cost, yet the root causes of performance issues may not necessarily correlate with high costs. Similarly, the critical paths reported by causal tracing tools may not always accurately represent the causal paths leading to performance issues.

The ineffectiveness of existing performance diagnostic tools stems from their inherent inaccuracies. This dissertation demonstrates vProf and Argus, emphasizing that integrating supplementary runtime information into traditional diagnostic tools is necessary to overcome these inaccuracies and effectively pinpoint the root causes of performance issues.

1.4 vProf: Value-Assisted Cost Profiling

Profiling tools are extensively utilized to identify resource-intensive functions within applications, enabling subsequent optimizations to address performance concerns. However, this conventional approach often falls short in accurately pinpointing the root causes of specific performance issues.

Function rankings in profiling results may lead to misguided efforts in bug hunting, because the top-ranked functions may not necessarily be the root causes of the observed performance issue. Instead, many performance problems can result from flawed code logic within seemingly lightweight functions. Such flaws are often intricately linked to improper data flow, which can be observed by monitoring the variables accessed within these functions. Unfortunately, existing profilers often miss this critical data flow information.

Moreover, the absence of data flow information in profiling results leads to vague outcomes, making it challenging to accurately diagnose performance issues. Developers, for instance, need to differentiate between excessive execution time attributed to a costly function itself and frequent function calls. In the former scenario, a comprehensive examination of branches and code blocks is required, while the latter necessitates the identification of the reasons behind these frequent function calls. Consequently, debugging complex performance problems often compels developers to resort to ad-hoc practices such as printf statements, software recompilation, execution, or debugger

attachment for deeper insights.

The key insight from our analysis is that the common inaccuracies plaguing current profilers often stem from the absence of program data flow information in profiling results. Essential runtime data flow, encompassing details like array length, variable values, and the historical evolution of these values during execution, proves indispensable for achieving effective performance diagnosis, as evidenced by our study of bug reports.

To determine which variables to record, it is crucial to generalize the requisite information for performance diagnosis. Performance issues typically manifest in three predominant patterns related to control flow and data flow — namely, *scalability*, *wrong constraint*, and *missing constraint*. Scalability-related issues arise when the array or loop size surpasses the developer's expectations, leading to repetitive iterations over objects that trigger performance problems. Issues linked to wrong constraints often result from erroneous variable configurations or unintentional negations of conditional expressions, which in turn lead to the execution of incorrect branches. Bugs stemming from missing constraints involve superfluous operations due to uniform execution. Consequently, variables utilized in conditional expressions, loops, and function calls are critical to enhance the effectiveness of the profilers.

Based on the insights, this dissertation introduces a new profiling methodology called *value-assisted cost profiling* (vProf). This methodology not only measures execution costs but also concurrently records the value of variables of interest throughout the profiling session. To enhance the profiling results for performance diagnosis, vProf integrates the runtime information of the variables to calibrate the function costs. Specifically, it corrects the underestimated functions and applies discounts to reduce the cost of inherently expensive ones. By comparing values collected during a problematic execution with those from a normal execution, vProf highlights root cause functions with associated anomalous values, identifies problematic code that accesses such values, and infers patterns of performance issues.

We implemented vProf by following the proposed profiling methodology and applied it to both resolved and unresolved performance issues in four large software systems to assess its effec-

tiveness. The results show that vProf significantly enhances the profiling results for diagnosing performance issues.

1.5 Argus: Annotated Causal Tracing for Modern Desktop Applications

The complexity of modern software poses a significant challenge for developers when diagnosing performance issues that propagate across multiple components from their origins. This challenge is particularly pronounced in modern desktop applications, which often exhibit high levels of concurrency and communication among threads to meet the evolving demands of users. When a performance issue arises, conventional profiling tools, such as macOS Instruments, usually identify the most resource-intensive functions. However, performance issues in modern desktop applications may not relate to the intensive resource usage. Instead, they can result from improper synchronization span different software components. For such performance issues, profilers often fall short in providing comprehensive insights into the underlying root causes.

Causal tracing provides a solution for identifying the root causes of these performance issues. It captures runtime information that reflects the causal relationships among components, constructs trace graphs based on these causalities, and subsequently identifies the causal paths leading to performance problems. While causal tracing tools have demonstrated their effectiveness in mobile applications and distributed systems, their potential for diagnosing performance issues in desktop applications remains untested.

We modified the implementation of a conventional causal tracing tool to investigate its effectiveness for desktop applications on macOS. However, the adapted causal tracing tool encountered inaccuracies in its generated trace graphs, featuring superfluous edges while omitting necessary ones. These inaccuracies stemmed from unaccounted programming paradigms and overlooked data dependencies.

While theoretically possible to rectify these inaccuracies through supplementary instrumentation, extensive instrumentation of desktop applications is practically unfeasible due to the significant overhead it would introduce. Furthermore, our comprehensive instrumentation attempts

revealed that the closed-source nature of desktop applications and their associated third-party components posed challenges and potential pitfalls.

Instead of attempting to eliminate all inaccuracies, we developed Argus to mitigate these inaccuracies while imposing minimal overhead on the system. To do so, Argus collects supplementary runtime information to ascertain the credibility of the trace graph edges, and automatically annotates trace graphs with *strong* and *weak* edges. Guided by the edge annotations in the trace graph, Argus employs a heuristic search algorithm that can tolerate the inaccuracies when identifying causal paths for performance diagnosis. Specifically, the algorithm expands the most promising causal paths within a limited set.

By prioritizing the most promising causal paths, Argus focuses on addressing performance issues characterized by either high processing activity or prolonged waiting, as determined by CPU status. Busy processing issues can be effectively diagnosed by analyzing the causal paths of the buggy execution. In cases involving prolonged waiting, performance issues often arise due to the absence of essential events that should have awakened the waiting threads. To identify missing events in problematic executions, Argus introduces a *differential-based* comparison algorithm. This algorithm uses the causal paths from a normal execution as a baseline and compares them to the partially similar paths in the problematic execution. The comparison algorithm identifies the problematic activities that lead to the absence of awakening.

Our evaluation of Argus on real-world performance issues in macOS demonstrates that Argus can effectively identify the root causes of complicated performance issues. Further details regarding the design and implementation of Argus are presented in Chapter 4.

1.6 Organization

The structure of this dissertation is as follows. Chapter 2 provides an overview of the efforts made for performance issue diagnosis. Chapter 3 discusses our work on enhancing profilers with data flow. Chapter 4 introduces Argus, an annotated causal tracing system for diagnosing modern desktop applications. Chapter 5 analyzes the remaining limitations of the enhanced performance

diagnostic tools and suggests potential improvements for the future. Finally, we conclude the dissertation.

Chapter 2: Background, Related Work and Research Goal

This chapter discusses the related works and the gaps in existing performance diagnostic tools that lead to the work in this dissertation. We begin with the examination of real-world performance issues in Section 2.1. In Section 2.2, we delve into the existing tools for performance diagnosis. Finally, we present our insights into the gaps of existing diagnostic tools in Section 2.3.

2.1 Studies on Real World Performance Issues

In the past few decades, performance issues have attracted significant attention. Several empirical studies [77, 56, 32] have been conducted to investigate how performance issues were introduced, discovered, and fixed, and to summarize the characteristics of performance issues.

Zaman *et al.* [77] conducted a qualitative analysis of 400 randomly selected bugs from Mozilla, Firefox, and Google Chrome. Their study aimed to identify the deficiencies in reporting, reproducing, tracking, and resolving performance issues. The findings revealed that developers and users often invested more time in discussing performance issues compared to other functional bugs, particularly in reproducing and debugging them. It was also observed that, at times, performance issues were occasionally tolerated as a trade-off due to the potential costs involved in addressing them.

Similarly, Nistor *et al.* [56] found that most performance issues were discovered through code review. Fixing performance issues proved to be more challenging compared to non-performance issues, taking on average 75 days longer to resolve. Developers need improved tool support in testing and addressing performance bugs.

Some research [38, 45, 32] has focused on unveiling common patterns of performance bugs that can provide valuable insights for guiding future software development.

Jin *et al.* [38] conducted a comprehensive study involving 109 real-world performance issues randomly sampled from Apache, Chrome, GCC, Mozilla, and MySQL. The root causes of the majority of real-world performance issues in their study fell into a few categories. Drawing from their insights, they summarized common patterns to assist developers. These patterns include *uncoordinated functions* caused by inefficient function-call combinations composed of efficient individual functions, *skippable functions* that perform unnecessary work given the calling context, *synchronization* issues, and others.

Liu *et al.* [45] conducted a similar study on 70 real-world performance issues from Android applications. They categorized performance issues specific to smartphones into three types: GUI lagging, energy leaks, and memory bloat. Based on how these performance issues manifested in the smartphone context, they found that common patterns causing the performance issues included lengthy operations in the main thread, wasteful computation for invisible GUI elements, and frequent invocations of heavyweight callbacks. Building upon their findings, they developed a static code analyzer named PerfChecker, which identified heavy APIs in the main thread and patterns that violated operating rules on the view holder.

PerfScope [32] was proposed based on the observation that performance regressions were often hidden within multiple patches. Through an analysis of 100 performance regression issues, the authors aimed to uncover the mechanisms that introduced such regressions. Their investigation revealed two distinct categories of code changes that led to performance issues: direct calls to resource-intensive functions and alterations propagated through data and control flow, which ultimately triggered an expensive function call. In light of these findings, PerfScope introduced strategies to prioritize testing of patches that were most relevant to the underlying performance issues.

2.2 Performance Diagnostic Tools

The studies discussed above revealed a diverse range of inefficient code that caused performance issues, especially in the context of multi-threading and multi-processing, which imposed

substantial challenges for developers in pinpointing their root causes. Such performance issues can propagate across thread, process, and even machine boundaries. Manual diagnosis would take a long time in those situations. Therefore, accurate diagnostic tools are important to support effective diagnosis.

In addition to widely used profilers and causal tracing tools, the goal of effective performance diagnosis has motivated the development of many new solutions for automating performance diagnosis over the past decades.

In the remainder of this section, we will explore the evolution of diagnostic tools. Specifically, we will first provide an overview of profilers in Section 2.2.1, followed by a discussion of causal tracing tools in Section 2.2.2. Finally, we will look into alternative techniques *replacing* profiling and causal tracing tools to automate performance debugging in Section 2.2.3.

2.2.1 Profilers

Existing profilers primarily focus on measuring execution costs, which can be quantified in two main ways: the number of invocations and execution time. The measurement granularity typically applies to the function level. This section reviews both the traditional profilers widely employed in production systems and the state-of-the-art profiling tools proposed by researchers.

Traditional profilers, such as `gprof` [28], `perf` [27], and `oprofile` [43], utilize scheduled periodic alarms to examine the value of the program counter at each interrupt. They then deduce the cost of a function from the collected program counter samples. For instance, `gprof` [28] samples the program by scheduling periodic alarms after a certain CPU time has elapsed. On the other hand, `perf` [27] and `oprofile` [43] sample programs based on hardware events, leveraging Performance Monitor Units (PMUs). When an overflow occurs in the configured PMUs, the CPU sends an interrupt signal to record the sample.

However, all of these tools inherently encounter inaccuracies. `gprof` requires compiling the target program with the `-pg` option, which prompts GCC to instrument every function in the program to call the monitoring routine `mcount()` at its entry point. This instrumentation can potentially

distort the performance of the target program. Furthermore, `gprof` exclusively profiles CPU time within the target program, which means that it does not examine performance issues arising from unexpected long waits or resource-intensive executions within libraries. On the other hand, `perf` and `oprofile` encounter delays between the overflow and the arrival of the interrupt signal, which leads to the well-known bias referred to as *skid*. Additionally, these profilers have unavoidable profiling bias for multi-threaded programs due to their inherent nondeterministic nature.

Decades of research have been devoted to proposing novel profiling techniques that assist developers in identifying inefficient code areas. In the context of prevalent concurrency, COZ [22] was a causal profiler aimed at optimizing complex and slow transactions. It randomly sampled lines from the program, applied a virtual speed-up to the chosen line, and calculated the potential speed-up for a target transaction. However, its substantial overhead and inability to profile child processes constrained its practical applicability in some real-world scenarios.

AlgoProf [79] recognized that the traditional algorithm complexity analysis often estimated function costs based on total executed instructions, overlooking the low-level impacts arising from cache and memory contentions within the system. These cost functions might have overly generous worst-case bounds or idealized average-case costs, which were potentially unrealistic. In response, AlgoProf introduced a novel profiler that automatically varied program inputs and measured function costs reflecting real-world performance. The resulting plots of function costs approximated the relationship between actual input and corresponding costs. They provide developers with profound insights into program performance.

Similarly, aprof [21] served as a Valgrind tool designed to aggregate performance costs based on input size for each executed function. It automatically measured input sizes for specific code segments. The collected information was then utilized to generate insightful performance plots. Additionally, aprof employed statistical curve fitting and bounding techniques to deduce trend functions. Due to this deduction, a notable distinction from Algoprof was that aprof could profile the program within a single run on typical workloads, while Algoprof required running the program multiple times.

Freud [65] was introduced to make algorithmic complexity more readable and comprehensible for programmers. This tool automatically generalized performance metrics, including CPU time, memory usage, and lock holding/waiting time, and established correlations between these metrics and the features of the input arguments. For instance, Freud can correlate the time cost with the size of the input array. To achieve this, the analyzer in Freud employed statistical analysis to construct mathematical models that inferred these correlations. Subsequently, it translated these models into a semantically meaningful annotation language. This approach empowered programmers with valuable insights into the relationship between algorithmic complexity and input arguments, facilitating a better understanding and informed decision-making.

Dmon [42] employed a selective profiling technique to identify data locality issues in production environments. To minimize overhead, it selectively and incrementally collected runtime information, utilizing the hierarchical top-down approach developed by Intel. The collected data were then used to automatically pinpoint data locality problems, identify access patterns that negatively impact locality, and subsequently apply targeted optimizations to address these patterns.

While those novel profiling techniques were powerful, their focus was primarily on optimizing specific functions rather than on performance diagnosis. Applying profilers to diagnose performance issues might miss root cause functions that were lightweight in terms of time cost but trigger resource-intensive executions in other functions.

2.2.2 Causal Tracing Tools

Profilers focus on identifying costly functions, but they do not answer two fundamental questions crucial for performance debugging: (1) *Why is the function costly?* (2) *Is the cost necessary for the task?* Conversely, causal tracing tools provide insights into these questions.

Causal tracing tools capture selected execution events along the workflow of a request handling, and construct trace graphs following the causal relationships among these events. Upon the trace graphs, a diagnosis algorithm, typically critical path analysis, is employed to pinpoint the root causes of performance issues. Various causal tracing solutions have been proposed to troubleshoot

performance issues in distributed systems. Prominent examples include Magpie [6], XTrace [26], Dapper [68], and Pivot Tracing [50].

Magpie [6] monitored specific events in the server and extracted the system's workload in the real world. To capture the workflow for each request, it employed a manually defined event schema to determine request boundaries within a thread and connect events for the same request. Through request standardization, Magpie accurately generated workload models and identified performance anomalies by detecting outliers among them. This approach can be utilized for performance prediction and change detection.

X-Trace [26] was a tracing framework designed to reconstruct a comprehensive view of service behavior. It associated metadata with payloads from clients and layers of network services. By propagating this metadata throughout the system, X-Trace identified and maintained the causal relationships. However, while X-Trace was useful for pinpointing points of failure, it required an additional detailed report for in-depth debugging.

Dapper [68] was Google's production distributed systems tracing infrastructure. It shared conceptual similarities with the tracing systems Magpie and X-Trace. They all aimed at assisting engineers in pinpointing the root causes of overall latency and addressing critical questions: which service is at fault and why it is performing poorly. However, Dapper distinguished itself from them in terms of the design choices. Dapper employed adaptive sampling and restricted core tracing instrumentation to a small corpus of ubiquitous threading, control flow, and RPC library code. This approach aimed to strike a balance between low overhead, application-level transparency, and scalability.

Pivot Tracing [50] integrated dynamic instrumentation with causal tracing techniques to build a monitoring framework for distributed systems. Its key contribution was a novel `happened-before` join, which effectively addressed the challenge of correlating events across component or machine boundaries when using dynamic instrumentation. Pivot Tracing empowered users to define arbitrary metrics at specific locations during runtime in the system. While tracing, Pivot Tracing made use of the defined metrics. It had the capability to select, filter, and group events based on specific

metrics, enabling the identification of causal relationships. However, it is worth noting that this tool required manual modifications to the source code to ensure the proper propagation of metadata for a given request.

In addition to distributed systems, prior works also investigated causal tracing techniques to help developers identify the performance bottlenecks in mobile applications.

AppInsight [60] intervened at the interface between the application and the framework to collect causal tracing logs and construct trace graphs. Specifically, it determined the boundaries of different requests within a thread by assuming that each callback function corresponded to a specific request. To construct trace graphs for requests, it treated the beginning and end of a callback function as vertices and connected them with an edge in the request graph, indicating that they belonged to the same request. The activities between the two vertices were all connected as an execution segment for a request. If this segment installed additional callbacks or signaled a waiting thread, AppInsight connected these corresponding execution segments with additional edges. As a result, by using the generated trace graphs, AppInsight estimated critical paths to pinpoint the root causes of execution exceptions or performance issues.

Panappticon [80] traced low-level events within the Android system. To construct the dependency graph, it captured causal relationships from two asynchronous programming idioms: the message queue and thread pooling. In the case of the message queue, it designated the handling of a message as a vertex and connected it to the event that queued the message. Regarding thread pooling, it marked a vertex from the wake-up of a worker thread to its waiting for locking primitives, and connected it to the vertex containing the event that woke up the worker thread. In addition to the defined connections across threads, Panappticon introduced a temporal join. This join connected events that appeared continuously within a thread. For instance, Panappticon treated the entry and exit of a callback function as vertices and connected all events between them. Ultimately, the generated trace graphs were utilized to identify and diagnose performance issues.

Since existing causal tracing tools are usually built upon causal relationships derived from specific programming paradigms, diagnosing performance issues becomes ineffective when cus-

tomized programming paradigms exist in the applications.

2.2.3 Automatic Performance Diagnosis

Profilers and causal tracing tools, as discussed, assume that the root causes of performance issues will be present in the profiling results or extracted paths. They often require developers to have prerequisite knowledge to understand the results for more in-depth debugging. For instance, causal tracing tools need developers to understand the causalities among vertices in a causal path to pinpoint the root causes of the performance issues. However, performance issues are complex and unpredictable, and obtaining the necessary information to comprehend the results can be challenging, especially for developers who may not have intimate knowledge of all components. To alleviate the debugging effort for developers, much research work [2, 5] has introduced inference rules to bridge the gap and automatically uncover the root causes.

Aguilera *et al.* [2] diagnosed performance issues in distributed systems by treating application services as black boxes. They employed timing analysis to establish correlations between input and output messages, allowing them to attribute the root causes of performance issues to specific nodes in the system.

X-ray [5] attributed the root causes of performance issues to program inputs or configuration settings, leveraging dynamic information flow analysis. It introduced a technique called performance summarization. The technique first assigned a cost to predefined events, such as a basic block from the user level. Next, it used dynamic information flow analysis techniques, such as taint tracking, to associate events with the program inputs or configuration settings and assign weights to the associations. These weights reflected the relative probabilities that the potential input or configuration caused the execution of the defined events. Finally, based on the association weight, X-ray outputted a ranked list of potential root causes by summing the costs of all predefined events for each of the potential root causes. The result can be used to diagnose performance issues or differentiate similar operations.

Several research works [29, 19, 66, 76, 23, 8, 9, 10] leveraged well-known bug patterns that

manifested in call stacks or logs to automate diagnosis. StackMine [29] associated callstack patterns with a set of performance metrics, aiding analysts in aligning and calculating similarities among call stacks to identify performance anomalies. Other research works [19, 66, 76, 23] applied machine learning methods to identify abnormal events in logs. Extensive research [8, 9, 10] employed inferred models from logs to automate the detection of abnormal behavior when systems are exposed to new workloads and environments.

Researchers [69, 62] also leverage statistical debugging to automate performance diagnosis. Statistical debugging is an effective technique originally employed to address functional bugs. It gathers program predicates, such as whether a branch is taken and interleaving sequences, during both successful and failed runs. It then employs statistical models to automatically identify the predicates most correlated with failures, which are referred to as failure predictors. For example, Holmes [16] collected path profiles from both successful and failed runs and constructed a statistical model using those path profiles to pinpoint predictors causing failures. Cooperative Concurrent Bug Isolation [39] tracked interleavings at runtime and established statistical models on these interleavings to identify strong failure predictors for concurrency bugs.

Song *et al.* [69] applied statistical debugging to diagnose performance issues. They monitored predicates, including conditional branches, scalar pairs, and function returns. Using the collected data from both normal and buggy executions, they ranked the predicates using a statistical model. Their statistical model followed the principle that a predicate serving as a more reliable predictor should be true in a greater number of failure runs and false or not observed in successful runs.

Perspect [62] is closely related to the work vProf in the dissertation. Both of them expand the scope of failure predictors to include runtime information, such as the execution frequency of instructions, data flow, and processing time for objects. Drawing inspiration from causal relationships and statistical debugging, Perspect employs instruction relationships to precisely identify the root causes of performance issues. It initially selects an effect instruction as the symptom of a performance issue manually. Then it identifies all potential pairs of causal and effect instructions through a comprehensive analysis of instruction causalities. Subsequently, statistical tests are used

to select relevant pairs from irrelevant ones by comparing normal and buggy runs. To pinpoint the root causes, Perspect explores causal paths to identify the instruction that is closest to the effect instruction in the buggy execution.

The methodologies employed by Perspect share similarities with those of vProf. Both approaches involve static analysis to narrow the scope of runtime tracking, a comparison of normal and buggy runs, and statistical methods to filter out data unrelated to the specific performance issue.

However, vProf distinguishes itself by incorporating variable values at runtime, as opposed to the frequency of instruction execution. This design decision is rooted in several considerations. Firstly, value samples can be used to calibrate function costs, prioritizing functions related to performance issues. Consequently, it eliminates the need to manually select symptoms from the source code. Secondly, value samples in vProf, carrying the program counter (PC) where variables are accessed, can infer the source code regions just like instructions. In contrast, the sheer number of instruction pairs can lead to overwhelming comparisons and excessive false positives. Thirdly, while Perspect may overlook instructions that solely appear in the buggy case, vProf consistently retains unique variables to prevent false negatives. This is primarily due to the manageable number of variables. Lastly, aligning and comparing the distribution of variable information is more precise and straightforward compared to instructions, especially when assessing runtime behavior across different software versions.

2.3 Improve Existing Diagnostic Tools

Despite the emergence of numerous automated performance diagnostic tools, profilers and causal tracing tools continue to provide substantial value for developers. They are mature, user-friendly, and practical in real-world scenarios.

However, as discussed in Chapter 1, profilers and causal tracing tools often prove ineffective in diagnosing complex performance issues. Within the context of the related works examined in this chapter, a research gap becomes evident: the results produced by existing diagnostic tools are in-

herently inaccurate, and the absence of comprehensive guidance makes the results fail to prioritize potential root causes for specific performance issues. Therefore, the primary objective of this dissertation is to enhance the effectiveness of performance diagnosis by incorporating supplementary runtime information.

This objective drives the dissertation to delve into two pivotal research questions: 'What information is requisite for the identification of root causes?' and 'How can this indispensable information be seamlessly integrated into existing tools to yield precise outcomes?' Specifically, this dissertation explores the necessary pieces of information for profilers and causal tracing tools. In the case of profilers, it is imperative to incorporate *data flow* to refine function costs, as a seemingly swift function in current profiling result might inadvertently trigger unnecessary computations due to improper value propagation. Similarly, the inherent inaccuracies of trace graphs primarily stem from customized programming paradigms and data dependencies. Mitigating these inaccuracies requires an understanding of the *semantics* within trace graphs, which can be derived from *call stacks during runtime*.

The corresponding improvement of a typical profiler is presented in Chapter 3, and the enhancement of causal tracing tools is discussed in Chapter 4.

Chapter 3: Value-Assisted Cost Profiling

In this chapter, we focus on analyzing the inherent inaccuracies in typical profilers, which render them ineffective for diagnosing performance issues. In practice, even with mature profilers, it often takes a developer a significant amount of time to identify the root cause of a performance issue. In a real-world performance debugging story [44], the developer "*spent 5 hours debugging, and finally moved a single line of code up 10 lines*", which reduced the CPU usage by 20×. Although the fix was simple, it took the developer many hours to find the bug, because the profiler results suggested the wrong places to investigate.

Such anecdotal examples widely exist. A key reason is that traditional profilers focus on identifying costly functions. They are effective when the performance bug happens to be in the function that takes the most time. However, tricky performance bugs are often caused by improper code logic. The buggy code itself may be fast and ranked low by profilers, misleading developers to waste effort trying to speed up costly functions that are necessary and already highly optimized. Consequently, we explore methods to address the ineffectiveness based on the widely used profiler `gprof`.

Figure 3.1 shows a real performance issue [11] in the widely used MariaDB as an example. Based on user-provided logs, developers suspected that the user's database caused an out-of-memory error. Existing profilers report that the function `recv_apply_hashed_log_recs` consumes most of the execution time, but this is not the root cause. From its call count, developers recognized that this function was called frequently. This could mean that the function is too costly to be executed frequently and needs to be further optimized. Alternatively, it could mean that there is an issue with the calling of the function. Knowing which answer is correct is difficult when the root cause is unknown. In this case, digging into and trying to optimize the `recv_apply_hashed_log_recs` function would waste huge amounts of time since it has hundreds of

```
 830 void recv_sys_init() {
 831   ...
 846   recv_n_pool_free_frames = buf_pool_get_n_pages() / 3;
 847

3192 bool recv_scan_log_recs(ulint available_mem, ...) {
3203   bool finished  = false;
3348   if (recv_parse_log_recs(checkpoint_lsn,
3349         store_to_hash, available_mem, apply)) {
3355     finished = true;
3356     goto func_exit;
3357   }
3376 }

3388 bool recv_group_scan_log_recs(lsn_t ckpt_lsn, ...) {
3417   ulint available_mem = srv_page_size *
3418         (buf_pool_get_n_pages() -
3419         (recv_n_pool_free_frames * srv_buf_pool_ins));
3424   do {                                          function has more
3431     recv_apply_hashed_log_recs(false);           than 200 LOC
3439     log.read_log_seg(&end_lsn, start_lsn + RSCAN_SIZE);
3440    } while (end_lsn != start_lsn &&
3441         !recv_scan_log_recs(available_mem, ...
```

Figure 3.1: A real performance issue in MariaDB (MDEV-21826). Existing profilers identify `recv_apply_hashed_log_recs` as the culprit because it is the most expensive function. But the root cause is the value of `available_mem` calculated in functions `recv_sys_init` and `recv_group_scan_log_recs`. The value is set to **zero** in the buggy case.

lines of code and 20 branches. The developers ended up not doing that and instead focused on the loop that calls the function. Nevertheless, they still ended up wasting significant time investigating the loop conditional and the call chains from `recv_scan_log_recs` to `recv_parse_log_recs`. Each function was complex, leading to a wild goose chase.

The real root cause is inside functions `recv_sys_init` and `recv_group_scan_log_recs`. The function `recv_sys_init` incorrectly sets variable `recv_n_pool_free_frames` to one-third of the buffer pool (line 846). It is used in `recv_group_scan_log_recs` to calculate variable `available_mem` (line 3417), incorrectly setting it to zero. As a result, `recv_scan_log_recs` returns false, causing wasteful computation in the loop (line 3441). The problem was not in the loop where the developers spent significant time, but in code before the loop. Developers missed focusing on the crucial beginning of the function `recv_group_scan_log_recs` before the loop, as profilers provided no indication that

this function was costly or important. Eventually, developers took 20 days to find the root cause, with the user being actively involved, even when their initial suspicions of an out-of-memory error turned out to be correct.

Our insight is that existing profilers' gaps are often caused by the lack of program data-flow information in the profiling result. Information such as the length of an array, the value of a variable, and the history of a variable's values during the execution is indispensable in debugging. Indeed, we observe that, in debugging complex performance issues, developers often have to take additional steps including adding *ad-hoc* `printf` statements, re-compiling and re-executing the software, and attaching a debugger like `gdb`, to obtain data-flow information to guide performance debugging.

Based on this insight, we introduce a new profiling methodology, *value-assisted cost profiling*. Its basic idea is to not only measure execution costs during profiling, but also *continuously* record the values of program variables to provide data-flow information. The recorded values are then used to distinguish anomalous costly functions from necessarily costly functions to localize the root cause in the code.

We build a tool *vProf* by modifying the popular gprof [28] profiler to realize value-assisted cost profiling, addressing three key challenges. First, vProf needs to decide which variables to record and how to locate them at runtime. Simply recording all variables and the complete program data-flow would incur unacceptable overhead, and invalidate the profiling results. Second, vProf needs to record variables efficiently in a manner that aligns well with other profiling information so it can be useful. Third, vProf needs to use the recorded value information to improve the diagnosis of performance issues.

vProf decides which variables to record by using static analysis to identify the types of program variables that commonly influence performance. vProf uses an LLVM [47] analysis pass to scan the source of the target program to identify these variables, typically generating hundreds to thousands of candidate variables.

vProf not only needs to identify which variables to record, but also reliably locate them at

runtime. The runtime location of a variable, especially a local variable, can change during program execution, such as being stored in different registers, pushed onto the stack, or becoming dead or out of scope. Like gprof, vProf presumes debugging information is available in the target program executable, which it statically analyzes to obtain variable scope and location information. This is used to record the variable values at runtime.

vProf records variables efficiently at runtime in a manner aligned with other profiling information by leveraging the same mechanism it uses for measuring execution costs. Like existing profilers such as gprof, to minimize the overhead, vProf uses program counter (PC) sampling to measure execution costs per function. It sets a periodic alarm such that at each alarm signal, vProf records the current PC to identify which function is executing. The executing cost of a function is determined based on how often PC samples occur in its address range. vProf leverages this same approach to passively record variable values at each alarm signal, which we refer to as *value samples*. To collect as many value samples without introducing significant overhead, vProf not only records value samples for variables accessible at the current PC, but also virtually unwinds the stack to record additional value samples in callers of the current execution context, as well as the PCs at which they are accessed.

Recording value samples for accessible variables concurrently for an arbitrary profiling signal, vProf introduces efficient data structures so that the variables accessible at a given PC can be quickly identified and recorded.

vProf improves the diagnosis of performance issues by introducing a novel post-profiling analysis algorithm that combines value samples with traditional profiling execution costs. Using only value samples is insufficient for performance debugging, as they can be noisy. The value samples themselves also do not carry any information about costs, while costs are central to performance reasoning. Instead, vProf uses value samples to calibrate raw execution costs in two ways.

First, in addition to computing function execution cost based on PC sampling, vProf uses value samples recorded with virtual stack unwinding to calculate a *variable-based function execution cost* based on how often value samples occur in functions. The idea is that a function that has

variables of interest that calls other functions should be considered more carefully even if its own execution time may not be that high. This is done by having the caller effectively inherit the execution cost of its callees, thereby making it appear more costly. A function that does not have variables of interest will have no value samples, so its variable-based execution cost will be zero. vProf assigns each function a raw execution cost which is the greater of the execution cost based on PC sampling and the variable-based execution cost.

Second, vProf computes a *discount ratio* for each profiled function based on the degree to which its associated value samples are anomalous. Anomalous values are determined by comparing value samples between normal and buggy executions of a target program. The more anomalous a function's variable values, the lower the function's discount ratio will be. vProf then weighs a function's raw execution cost by one minus its discount ratio. Discounting demotes necessarily costly functions and promotes suspicious, lower-ranked functions. vProf further identifies the basic blocks in which anomalous values occur to help developers localize the root cause of a performance issue.

We evaluated the effectiveness of vProf against other state-of-the-art tools, including gprof, perf [27], COZ [22], and statistical debugging [69]. We collected and reproduced 15 real-world performance bugs in large server applications, including Apache, MariaDB, PostgreSQL and Redis. We then used these various tools to attempt to diagnose the bugs. vProf ranks the root cause function first for seven of the bugs and within the top five for all 15 bugs. It significantly outperforms the other tools, which at best ranked the root cause function within the top five for at most six of the bugs.

We show that vProf has low profiling overhead, does not require explicit instrumentation or code changes to target programs, and provides a similar usage model to gprof. These properties make vProf a practical tool to assist developers to debug tricky performance issues. In fact, we used vProf to diagnose several previously unresolved performance bugs in MariaDB and Redis, which have been confirmed by their developers, demonstrating its usefulness in practice.

In summary, the contributions in this chapter include:

- A new value-assisted cost profiling methodology.
- Novel techniques to make this methodology work, including static analysis to select variables, passive value sampling, cost re-calibration and root cause analysis.
- An end-to-end tool vProf, and evaluation of it on large applications and real-world performance bugs.

3.1 Overview of vProf

Figure 3.2 shows the workflow of vProf, which can be decomposed into four steps. First, a developer runs vProf's schema generator to extract a list of variables in the target program to monitor during profiling. This schema generator uses static analysis on the program source code to automatically identify variables in instructions that likely influence a program's performance, such as global variables, variables in conditional expressions, and call parameters, as discussed in Section 3.2.

It records the definition locations of all identified variables. For example, in Figure 3.1, vProf identifies the variables `recv_n_pool_free_frames` and `available_mem` for monitoring, the former since it is a global variable and the latter since it appears in a conditional expression as a call parameter of the function recv_scan_log_recs (line 3441).

Second, the developer compiles the target program with the `-pg` flag, the same as using `gprof`, so that the resulting executable contains DWARF debugging information [24]. This is used to translate the generated schema into runtime location information for the variables of interest. For example, in Figure 3.1, the global variable `recv_n_pool_free_frames` is accessed via its memory address, but the local variable `available_mem` is accessed from a register determined by the compiler. vProf uses the debugging information to determine what register to use to access `available_mem`.

Third, the developer runs and profiles the program executable. The same `-pg` flag used for compilation alters linking to link the executable with the vProf profiling library. At the start of program execution, the library reads the generated schema into memory and sets periodic alarms, using the `profil` system call. At each alarm signal, vProf collects the PC and value samples,

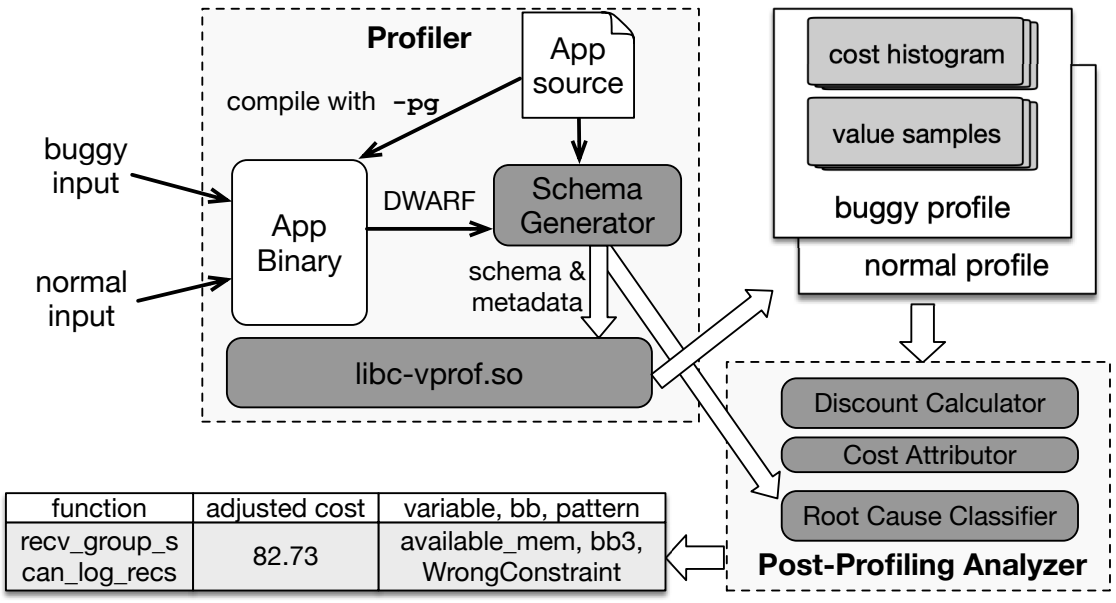


Figure 3.2: Workflow of vProf.

the latter by using the schema to determine which variables are accessible at the current PC and where to read their values. vProf also performs bounded virtual stack unwinding to record value samples in the callers of the current function. The developer is expected to profile the program at least twice using vProf, one to produce a profile of a normal execution and another to produce a profile of a buggy execution. Obtaining a normal execution is usually not difficult, as it often only requires executing the program with a smaller workload or less complex command. For example, in Figure 3.1, variable `recv_n_pool_free_frames` will have some constant value for each execution of the program, but the value will be different for a normal versus buggy execution. Similarly, variable `available_mem` will have some nonzero value for a normal execution, but be zero for a buggy execution.

Finally, the developer runs the vProf post-analysis tool, using the normal execution profile of the program as a baseline to compare against the buggy execution profile. PC samples are used to determine the execution cost of each function. If the alarm interval is t and the PCs that lie in the address range of function f are sampled n times during the profiling session, then the execution cost of f is calculated as $t \times n$. Value samples are grouped based on the func-

tions where they occur and used to calculate a variable-based execution cost and a discount ratio to adjust the cost of each function. The discount ratio is based on a comparison of the value samples from the normal and buggy profiles, with larger discounts for more similar value distributions between the profiles. vProf automatically classifies bug patterns based on the value samples and identifies where anomalous value samples occur to pinpoint suspicious basic blocks. For example, in Figure 3.1, function `recv_group_scan_log_recs` will be assigned a variable-based execution cost and have no discount to its execution cost because of the presence of anomalous values for its variables `recv_n_pool_free_frames` and `available_mem`. On the other hand, function `recv_apply_hashed_log_recs` will have a substantial discount to its execution cost. The end result is that vProf will rank the former ahead of the latter, alerting the developer to the correct root cause of the performance issue.

3.2 Schema Generator

To enable value-assisted cost profiling, we need to decide what variables to monitor during profiling. If a variable key to a performance issue is not monitored, vProf's effectiveness will become similar to conventional profiling. To address this challenge, we use program analysis to systematically identify the types of variables that commonly influence performance. Then, we make value recording efficient enough to allow vProf to sample many variables.

3.2.1 Source Code Static Analysis

vProf leverages LLVM to automatically identify the variables to monitor. For C/C++ programs, it uses the widely used Clang compiler frontend to parse the target program source code into LLVM's language-independent intermediate representation (IR). For each program source file, LLVM IR provides a call graph for all functions in the file. vProf introduces a simple LLVM analysis pass to traverse the call graph and identify where the variables of interest are defined. vProf identifies variables that are important to reason about performance bugs, specifically global variables and local variables from loops, branches, and function calls. vProf monitors all global

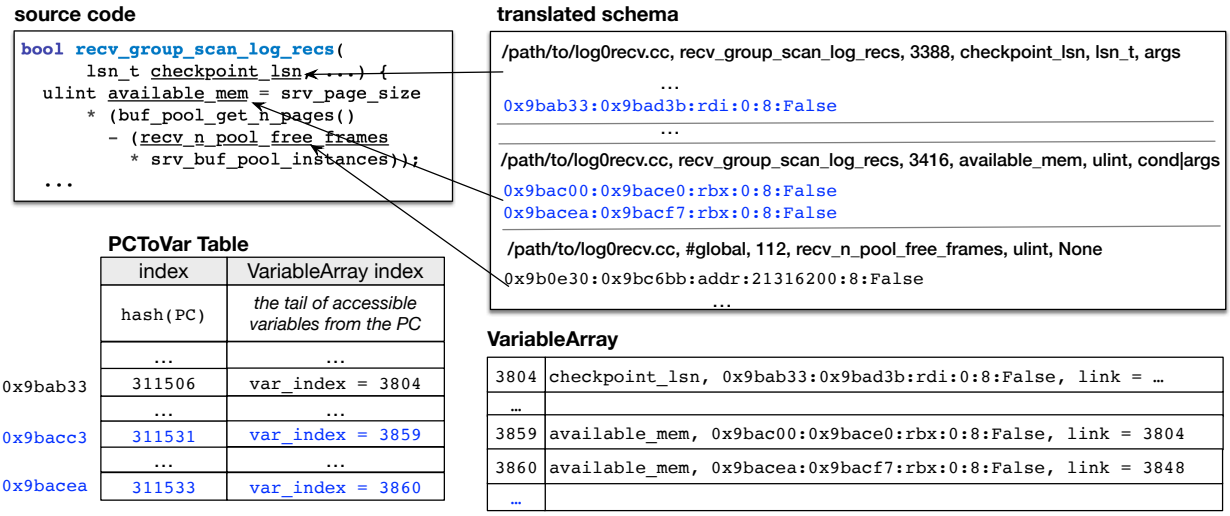


Figure 3.3: vProf generates variable metadata and initializes profiler data structures from the schema for the example in Figure 3.1. Highlighted entries indicate overlap in PC ranges with other variables.

variables in part because most programs contain only a relatively small number of them and they are accessible from any execution context, making them easy to monitor with low overhead. vProf is more selective with local variables, since monitoring all of them would be too costly. For loops, vProf monitors the induction variables, which can indicate not only the number of iterations but also timing information. For example, if an induction variable's sampled values are $3, 6, 6, 6, 6, 9$ in the buggy profile and $3, 6, 8$ in the normal profile, it could indicate a performance issue caused by a missing skipping or breaking condition inside the loop, because the iteration 6 lasts for a much longer time in the buggy profile. For branches, vProf monitors all variables in a conditional expression. For call instructions, vProf monitors all variables used as call parameters.

vProf typically affords the ability to monitor thousands of variables, which can include all relevant variables for small programs. For large programs, to reduce overhead, developers can limit the variables to monitor to specific components of the program related to a performance issue, e.g., the buffer pool component in MariaDB whose source code locates in `storage/innobase/buf`. vProf will then only extract variables in source files of the specific component. If the restricted value recording does not reveal the performance bug, developers can iteratively choose another component to monitor.

The analysis pass returns a schema showing where each variable being monitored is defined in the source code. Each variable is a schema entry in the following format:

```
file_path, function, line, variable, type, tags
```

`file_path` is the file path of the source code file that contains the variable definition. `function` is the name of the function that contains the variable definition if it is a local variable or the keyword `#global` if it is a global variable. `line` is the line number of the source code file where the variable definition is located. `variable` is the name of the variable. `type` is the type of the variable. `tags` is a set of vProf-specific tags that indicate how the variable is used, such as `loop`, `branch`, and `args`. For example, vProf monitors the variables `recv_n_pool_free_frames` and `available_mem` in Figure 3.1, which are represented in the schema shown in Figure 3.3. `recv_n_pool_free_frames` has `tags` equal to `none` since it is not used in any loop induction variables, branch conditional expressions, or call parameters. `available_mem` has `tags` equal to `cond|args` since it is used in conditional expressions and call parameters.

3.2.2 Binary Static Analysis

vProf transforms the schema to automatically identify the runtime locations of variables to monitor, which we refer to as *variable metadata*. Once the developer compiles the target program with the `-pg` flag, the program executable contains DWARF debugging information. vProf simply uses a DWARF parsing library [7] to search the debugging information to retrieve the scope and location information for each variable in the schema. vProf outputs a new schema of variable metadata, where each entry represents a contiguous range of PCs in which the variable can be accessed. Each entry of variable metadata is in following format:

```
pc_start:pc_end:location:offset:size:basic_type_ptr
```

`pc_start` to `pc_end` is the range of PCs for which the entry is valid. `location` indicates the location in which the variable can be accessed, such as a register. `offset` is either the offset at which to access a variable in a register or the address at which to access the variable in memory. `size` is the size of the variable. `basic_type_ptr` is a flag to indicate whether the variable is a pointer to a basic

type, such as a `char` or `int`, in which case vProf can dereference the pointer to obtain the actual value that is stored. vProf may generate multiple entries of variable metadata for each variable.

For example, Figure 3.3 shows some of the metadata entries generated for the variables in Figure 3.1. The entry for `recv_n_pool_free_frames` indicates it is accessible in memory at address 21316200, 8 bytes in size, and not a basic type pointer. The entries for `available_mem` indicate that it is accessible in register `rbx`, 8 bytes in size, and not a basic type pointer. Its offset is zero as it uses all bits of the 64-bit register.

DWARF debugging information may be incomplete, in that a variable may be accessible at a given PC but the information is not captured in the debugging information. For example, the entries for `available_mem` in Figure 3.3 cover two separate PC ranges in the function `recv_group_scan_log_recs`. The first entry includes the variable definition and the second entry includes its use in the conditional expression. However, there is a gap between them, likely because `available_mem` is pushed onto the stack due to the call to `recv_parse_log_recs`, and thus no longer accessible in a register. Efficiently determining the exact address on the stack from which to read such variables is a challenge. For simplicity, vProf assumes that a variable is not accessible at a given PC if there is no explicit DWARF debugging information that includes the PC to indicate its runtime location.

3.2.3 Profiler Intialization

Since profiling is done using PC sampling, we want an efficient mechanism to determine what value samples to record at a given PC. vProf accomplishes this by transforming the variable metadata into a more efficient representation used for profiling. vProf introduces two data structures in the profiler, a PC hash table, `PCToVarTable`, and an array for the variable metadata, `VariableArray`, shown in the example in Figure 3.3. The data structures are connected via a `var_index` field in each entry of `PCToVarTable` and a `link` field in each entry of `VariableArray`. By default, `PCToVarTable` is allocated to be half the size of the text section of the program being profiled.

Before executing the program to be profiled, vProf reads the variable metadata from a file.

For each metadata entry, vProf allocates an entry in `VariableArray` for the metadata and hashes each PC in the range of the metadata to an entry in `PCToVarTable`, which it fills in. For example, Figure 3.3 shows that the variable `checkpoint_lsn` is accessible starting at PC value 0x9bab33. vProf allocates the `VariableArray` entry at index 3804 to `checkpoint_lsn`, and fills in multiple `PCToVarTable` entries, including 311506 for PC 0x9bab33, whose `var_index` is set to 3804. Collisions from hashing different PCs to the same element of `PCToVarTable` are handled using separate chaining.

Multiple variables may be accessible at a given PC. If vProf finds an entry in `PCToVarTable` already filled in for a given PC, that means that some other variable metadata entry has an overlapping PC range with the one currently being processed. If the entry in `PCToVarTable` is already filled, vProf saves the `var_index` from `PCToVarTable` to the `link` field of the current `VariableArray` entry for the variable metadata currently being processed. It then updates the `PCToVarTable` entry with the index of the current `VariableArray` entry. In this way, multiple `VariableArray` entries are chained together to a related `PCToVarTable` entry.

For example, in Figure 3.3, the `var_index` of `PCToVarTable` entry 311531 for PC 0x9bacc3 stores the index 3804 for the `checkpoint_lsn` `VariableArray` entry since for PC 0x9bacc3 falls within the PC range for `checkpoint_lsn`. When processing the variable metadata for `available_mem`, PC 0x9bacc3 also falls within the PC range. The `link` field of the `available_mem` `VariableArray` entry is thus set to 3804. The `var_index` of `PCToVarTable` entry 311531 is then updated to the index 3859 for the `available_mem` `VariableArray` entry.

Note that Figure 3.3 shows the state of `PCToVarTable` and `VariableArray` before processing the variable metadata for `recv_n_pool_free_frames`, a global variable that is accessible at all PCs shown in `PCToVarTable`. For example, after that variable metadata is processed, the `var_index` of `PCToVarTable` entry 311531 will be updated to the index of a new `VariableArray` entry, storing the metadata for the global variable `recv_n_pool_free_frames`, which in turn will have its `link` set to 3859.

After this process, the metadata of all variables is stored in `VariableArray` and accessible by PC

from PCToVarTable. vProf also stores the mapping from the schema to VariableArray in a Layout Log, which is used later for post-profiling analysis.

3.3 Value Sample Recording

vProf's program analysis and data structure design make it straightforward to efficiently record value samples during profiling. vProf uses PCToVarTable, VariableArray and a SampleArray to store value samples. When the alarm fires and the PC is sampled, vProf reads all *accessible* variables according to the metadata. It looks up the sampled PC in PCToVarTable and follows its var_index and subsequent link fields in the chain of VariableArray entries. For each VariableArray entry in the chain, vProf checks that the sampled PC falls within its PC range, in which case it accesses the variable value and stores it, as well as the sampled PC, to a new SampleArray entry.

For example, when profiling the program shown in Figure 3.3, if the alarm fires and the PC sampled is 0x9bacc3, vProf will look up the PCToVarTable and follow its var_index. We assume for this example that the PCToVarTable and VariableArray have been updated to include the variable metadata for the global variable recv_n_pool_free_frames. Thus, var_index will be the index to a recv_n_pool_free_frames VariableArray entry. vProf will record the recv_n_pool_free_frames value in a new SampleArray entry. vProf will then follow the link to VariableArray entry 3859 and record the available_mem value in a new SampleArray entry. vProf will then follow the link to VariableArray entry 3804 and record the checkpoint_lsn in a new SampleArray entry.

Checking that the sampled PC falls within the variable metadata's PC range is necessary as it is possible for this not to be true due to the manner in which VariableArray entries are linked together when their PC ranges overlap, especially since the property is not transitive. Since most variables are local with limited PC ranges only accessible within their respective functions, we do not expect to encounter many VariableArray entries linked to a PCToVarTable entry which are not accessible.

SampleArray entries are chained together with their corresponding VariableArray entry. Each SampleArray entry has a link field. Each VariableArray entry has a sample_tail field, which is

used to record the index of the most recently recorded `SampleArray` entry for that variable. When a value is stored to a new `SampleArray` entry, its `link` is set to the `sample_tail` from the respective `VariableArray` entry, and the `sample_tail` is updated to the index of the new `SampleArray` entry.

vProf's passive value recording approach relies on having PC samples occurring within the PC range of the variables being monitored. For functions that do not run much, vProf may not get enough value samples. This can be an issue especially for callers with time consuming callees. For example, in Figure 3.1, the root cause function `recv_group_scan_log_recs` calls the costly function `recv_parse_log_recs`, so vProf almost always only observes PCs from `recv_parse_log_recs` when it samples the PC. Thus, vProf has few samples for local variables like `end_lsn` and `available_mem` in the root cause function, which are not accessible in the PC range of `recv_parse_log_recs` based on the DWARF debugging information available. A related shortcoming of `gprof`, on which vProf is based, is that when a target program calls into a dynamic library, `gprof` does not record PC samples since they are outside the range of the target program.

To address this issue, vProf introduces virtual stack unwinding. For each sampled PC, it unwinds the call stack by a bounded depth (default 3) and records variables accessible at the caller PC, which is PC before the `call` instruction. Specifically, we restore the registers in each step and begin the value sampling using the caller PC. We also add a field `stack_depth` in the `SampleArray` entry to indicate how many stack layers are unwound before the sample is recorded. The stack frames are restored to their normal state before virtual unwinding at the end of the sampling. Virtual stack unwinding allows vProf to obtain many more value samples to improve the fidelity of profiling. For example, in Figure 3.1, virtual stack unwinding results in value samples for `recv_n_pool_free_frames` and `available_mem` in `recv_group_scan_log_recs` even when the PC sampled occurs in `recv_apply_hashed_log_recs`. Note that virtual stack unwinding will generate no additional samples if there are no variables of interest accessible at the caller PCs.

vProf dumps the profiling data to disk at program exit. It saves PC samples and variable samples separately. The samples are then processed as part of post-profiling analysis.

3.4 Post-profiling Analysis

After value sample recording, vProf analyzes the data files from both normal and buggy executions. The data files include the PC samples, which `gprof` refers to as the PC cost histogram, value samples, and layout mapping used to connect value samples to variable information. vProf performs two post-profiling analyses. Cost calibration computes raw execution costs and then adjusts them based on anomalous value samples to promote suspicious functions in a function cost ranking. Bug pattern inference infers potential root cause patterns to help developers narrow down the root cause.

3.4.1 Cost Calibration

Traditional profilers only rank functions based on their raw cost, where a function may be ranked high due to unavoidably costly operations, while the real culprit of a performance issue is lower in the raw cost rank. vProf calibrates the cost of functions by increasing the cost of functions that contain many variables of interest, and decreasing the cost of functions whose variables are not anomalous.

vProf increases the cost of functions with variables of interest by computing an alternative execution cost based on the frequency of value samples, which we refer to as the variable-based execution cost. The standard approach to determine the execution cost of a function using PC sampling is to count the number of PC samples that lie in the PC range of the function and multiply it by the alarm interval. Instead of counting PC samples, vProf determines the variable-based execution cost by counting the number of value samples with distinct PCs that lie in the PC range of the function and multiplying it by the alarm interval. Multiple value samples at the same PC are counted as one sample. vProf then uses the maximum of the two costs as the raw execution cost of the function.

The variable-based execution cost will be higher than the standard execution cost if the number of value samples with distinct PCs in a function is higher than the number of PC samples. This can

occur especially due to virtual stack unwinding if some variables being monitored are accessible within the function, and the function calls some other function with higher execution cost. The idea is to use the higher variable-based execution cost as the function has variables of interest which could be related to a performance issue. For example, `recv_group_scan_log_recs` has a higher variable-based execution cost than its standard execution cost since variables being monitored such as `available_mem` are accessible within the function and it calls `recv_apply_hashed_log_recs`. This will result in it having many more value samples than its own PC samples because the value samples will occur at the frequency of the PC samples of its more time consuming callee due to virtual stack unwinding.

vProf decreases the cost of functions whose variables are not anomalous by introducing a *variable-discounter*, which is vProf's main cost calibration mechanism. It computes a discount ratio for each sampled variable based on how anomalous are its samples. The less anomalous the samples are, the greater the discount ratio, meaning that the variable is unlikely to be contributing to the performance issue. Discount ratios for variables are aggregated to the functions in which they are accessible to compute a discount ratio for each function. The cost of a function is calculated by multiplying its raw execution cost and one minus the discount ratio, which is between zero and one. As a result, a greater discount ratio (less anomalous samples) results in a greater decrease in the calibrated execution cost, so that the respective function will be less likely to be considered in diagnosing a performance issue.

We first describe how vProf determines how anomalous are a variable's samples and computes a discount ratio. The idea is to compare the value samples collected from the normal execution versus those collected from the buggy execution. vProf defines samples as anomalous based on how different the sample distributions are between the normal and buggy executions. The idea is to consider distributions to be different if they have different shapes. For example, if two distributions with the same normal distribution shape will be considered the same even if their means are different, but a normal and uniform distribution will be considered different.

Specifically, given the null hypothesis that the distributions are identical, vProf applies the k-

sample Anderson-Darling test [3] to the distributions to determine if the null hypothesis holds with some probability. By default, vProf uses a probability of 0.05. This means that vProf assumes the distributions are the same by default unless it can determine with high (95%) confidence that they are different. If the null hypothesis holds, vProf sets a discount ratio of DefaultDiscount for the variable, which is 0.8 by default. If the null hypothesis is rejected, vProf calculates the Hellinger distance [55], a measure of how different the distributions are. Its value is between 0 and 1, where a larger value indicates greater difference. The discount ratio for the variable is set to one minus the Hellinger distance, unless it is below a ValidDiscount threshold, in which case the ratio is zero. ValidDiscount is 0.1 by default.

Assuming the variable is a basic type, vProf considers the degree of anomaly in a variable along three dimensions. First, it considers values, as previously described. Second, it considers deltas of values in adjacent samples. This quantifies how much the values change. Third, it considers processing costs of values, specifically how many alarm intervals a variable value stays the same. This quantifies how often the values change. vProf determines the discount ratio for a variable in each of the three dimensions, and uses the lowest of the discount ratios. For pointers to non-basic types, vProf only uses the discount ratio based on processing costs, since the differences in pointer values, meaning differences in addresses, is not generally a useful distinction between normal and buggy executions.

We next describe how we aggregate discount ratios for variables to functions. For local variables, their discount ratios are attributed to the function in which they are defined. For global variables, their discount ratios are attributed to the functions which contain recorded PCs at which the variable was sampled. When a function has multiple associated variables with different discount ratios, vProf uses the lowest discount ratio among them, because the most anomalous variable often suggests the function is worthy of further examination. For each function, if its raw execution cost is x and its discount ratio is r, its calibrated cost is $(1 - r) \times x$. By using a DefaultDiscount of 0.8, vProf can significantly demote costly functions without anomalous value samples, but avoid eliminating them entirely. By using a ValidDiscount of 0.1, vProf can preserve the ordering of

functions by cost for functions with similarly low discount ratios, as value samples may be noisy. Section 3.5.3 evaluates how sensitive vProf is to these defaults.

For large programs, the variables being monitored may be limited to functions located in certain program components, resulting in no discount ratio being available for functions outside of those program components. To derive a discount ratio for these functions as well, vProf includes a simple *hist-discounter*, which computes a discount ratio by comparing how the function ranks in terms of raw execution cost between normal and buggy executions. Because of potential variability in the rankings, the hist-discounter is based on profiling the program multiple times. Given n buggy profile(s) and m normal profile(s), we perform a cross-comparison among the two groups for each function. We maintain a counter h for each function to record in how many comparisons this function ranks higher in the normal profile(s) than in the buggy profile(s). We also record c ($c \leq n \times m$) as the number of comparisons for the function. Then we set the discount ratio to $r = \frac{h}{c}$. The ValidDiscount threshold is also used with hist-discounter to avoid reordering the rankings of functions with similar low discount ratios. The hist-discounter is only used for functions which otherwise would have no discount ratio available.

3.4.2 Bug Pattern Inference

Since providing a high-level characterization of potential root cause patterns can further ease performance debugging, vProf provides a root cause classifier to infer potential root cause patterns for top-ranked functions based on their calibrated costs. We observe three common performance bug patterns:

1. *Wrong constraint*: These bugs cause the program execution to unnecessarily fall into a costly path. They often happen when a conditional expression or its evaluation is incorrect. For example, Figure 3.1 shows the while loop condition is evaluated with an incorrect `available_mem`.
2. *Missing constraint*: These bugs occur when the code performs some operations uniformly instead of discriminating based on some constraint, such as a conditional expression. For example, Figure 3.4 shows such a bug in Apache fixed by adding a conditional expression.

```
void ap_mpm_pod_killpg(ap_pod_t *pod, int num) {
  for (i = 0; i < num && rv == APR_SUCCESS; i++) {
+   if (ap_image->servers[i].status != SERVER_READY ||
+     ap_image->servers[i].pid == 0)
+     continue;
    rv = dummy_connection(pod);
  }
}
```

Figure 3.4: Root cause for performance issue HTTPD-54852. When using the Multi-Processing Module (MPM), the graceful restart of Apache httpd can sometimes take a few minutes. The problem is the `dummy_connection` call becomes much slower due to polling if all the children have already exited. Developers fixed the bug by adding a check in the loop to skip unnecessary `dummy_connection` calls when there are no more children.

3. *Scalability*: These bugs usually arise when the program processes data larger than the developers expected, such as traversing a large list in a critical section. For example, Figure 3.5 shows such a bug in MariaDB.

To infer the bug pattern for each function, the classifier queries the variable-discounter for information about which sampled variable was most anomalous. Specifically, for each function, it finds the anomalous sampled variable with the minimum discount ratio and the dimension used in calculating that ratio. Then, it obtains the the variable's abnormal samples from the buggy execution. The variable-discounter provides this by computing a variable's normal range from the normal execution and identifying the value samples in the buggy execution that are out of the normal range. Since each value sample contains the PC at which it was recorded, the classifier uses the DWARF information to map the PC back to the text section to localize the code region for abnormal samples and get the basic block label and control flow structures.

The classifier then checks how an anomalous variable is used in the code region based on its tags, as discussed in Section 3.2. With the discount ratio, dimension, and tags, the classifier infers the bug patterns by using the following rules in order:

1. If some loop induction or conditional expression variable stays the same for an abnormally long time, which is identified as a variable with a `loop` or `cond` tag and anomalous samples based on a discount dimension of processing cost, the function is labeled with a *Missing Constraint* bug.
2. If some loop induction variable has abnormal values, which is identified as a variable with a

```
bool buf_LRU_scan_and_free_block(bool scan_all) {
  ulint scanned = 0;
  for (bpage = buf_pool.lru_itr.start(); bpage && scan_all;
      ++scanned, bpage = buf_pool.lru_itr.get())
    ...                    the LRU list search was slow, scanned=134468
}
buf_block_t* buf_LRU_get_free_block() {
loop:
  mutex_enter(&buf_pool.mutex);
  block = buf_LRU_get_free_only();
  ...
  if (n_iterations || buf_pool.try_LRU_scan)
    freed = buf_LRU_scan_and_free_block(n_iterations > 0);
  ...
  mutex_exit(&buf_pool.mutex);
  n_iterations++;                    scan the whole LRU list
  goto loop;                         when n_iterations > 0
}
```

Figure 3.5: Root cause for performance issue MariaDB-23399. Under I/O-bound TPCC workloads, MariaDB throughput gradually decreases and is worse than a previous version. The problem arises when the buffer pool is full. The function `get_free_block` calls `buf_LRU_scan_and_free_block` which contains a linear scan of 1.6 million buffer pool blocks. The thread holds the `buf_pool.mutex`, preventing other threads stopping the scan by releasing pages to the buffer pool.

`loop` tag and anomalous samples based on a discount dimension of value or delta of the value, the function is labeled with a *Scalability* bug.

3. If a conditional expression variable is abnormal, which is identified as a variable with a `cond` tag and anomalous samples, the function is labeled with a *Wrong Constraint* bug.
4. If the most costly function is normal and has no variables of basic types being sampled, meaning it has a DefaultDiscount and discount dimension of processing cost, the function is labeled with a *Scalability* bug. Without values of basic types, vProf does not have enough information to identify other bug patterns in this case.

3.5 Implementation and Evaluation

We implemented vProf for C/C++ programs, mostly by modifying `gprof`, though vProf is compatible with any profiler based on PC sampling. This involved changes to `glibc`, mainly in `gmon.c` and `profil.c`. We modified `gmon.c` to set up the in-memory profiling schema metadata on initial-

ization, which is called from `__monstartup`. We modified `profil.c` to collect value samples. We extended the profiler signal handler to read values of variables accessible from the current PC. We implemented virtual stack unwinding using the `libunwind` library [54]. We fixed issues in `gprof` to better support multiple-process programs, such as renaming the `gmon.out` file with the process id, setting profiling timers for child processes, and unblocking `SIGPROF` signals. We implemented the schema generator using an LLVM analysis pass and a Python library. We implemented the post-profiling analysis in Python.

We evaluated vProf in diagnosing performance issues in widely used applications. We performed a comparative study against other state-of-the-art solutions on previously diagnosed performance issues to quantify effectiveness. We further used vProf to diagnose several previously unresolved performance issues in widely used applications. We also quantify vProf's performance overhead. All measurements were done on a desktop computer with a 6-core (12 hyper threads) Intel 2.60 GHz Core i5 CPU and 48 GB DRAM, running Ubuntu-20.04 with Linux kernel 5.11.0.

To collect bugs for evaluation, we considered four large applications: MariaDB [51], Apache HTTPD [4], Redis [61], and PostgreSQL [57]. We queried their official issue trackers using keywords *slow* and *performance*, randomly selected from among the issues, read their reports, and included the issues if they were truly performance-related and the reports had sufficient information for bug reproduction. We then excluded bugs that developers found from just reading source code as such bugs typically do not impact real users. In total, we collected 26 issue tickets. Three of the issues could not be reproduced by following the reports. Five of the issues were database-related and could be resolved by simply comparing the SQL explanations in the normal and buggy cases. Our evaluation focused on the remaining 18 out of the 26 issues, including 15 resolved issues, listed in Table 3.1, and three unresolved issues, discussed in Section 3.5.4.

3.5.1 Comparative Study

We used the bugs in Table 3.1 to evaluate the effectiveness of vProf versus other widely used and state-of-the-art tools in diagnosing performance issues in widely used applications. The other

ID	Description	Bug Pattern
b1: MDEV-21826	After a power failure, mariadb-server-10.3 starts a crash recovery but it seems to loop on the same log sequence number(LSN) forever.	Wrong Constraint
b2: MDEV-23399	Performance with IO-bound tpcc drops over a varying time period when there is a competition for buffer_pool space.	Scalability
b3: MDEV-13498	Deleting a table with CASCADE constraint is very slow.	Missing Constraint
b4: MDEV-15333	Slow start-up even when `.ibd` file validation is off.	Wrong Constraint
b5: MDEV-17933	Checking the server status takes >10 seconds with 3M tables.	Scalability
b6: HTTPD-62668	Multiple threads spin at 100% in the server after the request from Google PageSpeed Insights (PSI) has processed.	Missing Constraint
b7: HTTPD-54852	Gracefully restart service with MPM workers takes long time.	Missing Constraint
b8: HTTPD-62318	Health check is executed more often than configured interval.	Wrong Constraint
b9: HTTPD-64066	Slow startup/reload when many `vhosts` are configured	Scalability
b10: HTTPD-52914	When the server processes a POST request with a Content-Length header but only part (but not all) of the request body sent, the body timeout will be triggered. After the timeout, the workers in the server would loop with 100% CPU even though no client sent requests.	Wrong Constraint
b11: Redis-8145	`cluster nodes` command is costly in a large cluster.	Scalability
b12: Redis-8668	`BRPOP` becomes slow when a large number of clients exist.	Missing Constraint
b13: Redis-10310	`ZREVRANGE` command 50% slower after Redis is upgraded.	Missing Constraint
b14:Postgres-17330	`EXPLAIN` query hangs for some query plans.	Scalability
b15:Postgres-14b1	Vacuum process fails to prune all heap pages and endlessly retries.	Wrong Constraint

Table 3.1: Reproduced real-world performance issues.

tools we tried were `gprof`, `perf`, `perf` with an enhancement using Intel Processor Trace (`perf`-PT), COZ [22], and statistical debugging [69] (stat-debug). Table 3.2 briefly describes each tool and how it was configured; similar configurations were used whenever possible.

Several of the tools, `perf`-PT, statistical debugging, and vProf, required profiling normal execution in addition to the buggy execution. Normal executions were obtained for MariaDB-21826 and Redis-10310 by running the same command on a different version. Normal executions for all other issues were mostly obtained by using smaller inputs on the same software version. Specifically, we reduced the number of tables in the database for MariaDB, the number of virtual hosts in Apache httpd, and the number of nodes in a cluster for Redis. For example, in MDEV-13498, we deployed a database with the test script provided by the user in the bug report. Deleting the first table took 20 minutes, which exposed the symptom. Deleting a second table from the same script took 2 minutes, which we used as the normal execution. We simply reran the same command with the same inputs multiple times if multiple profiling runs were needed.

Because the applications are large, several of the tools require some identification of the component in which the performance issue occurs, to limit overhead. For `perf`-PT, we only performed its control-flow profiling on the top ten most costly functions by using the Intel Processor Trace address filter feature to limit the size and decoding time of the resulting branch traces. For COZ, we identified the top-level function in the source code file that contains the performance issue to limit runtime since it can otherwise take many hours to run as it randomly picks source code lines to virtually speedup to measure potential performance improvement. For statistical debugging and vProf, we identified the source code file that contains the performance issue to limit the predicates and variables sampled, respectively.

For each issue, we ran each tool on a buggy execution that reproduced the issue based on descriptions in the bug reports. We then measured how the tool ranked the root cause function in its output; lower number rank is better. The best result is for a tool to rank the root cause function first, meaning the tool pinpoints the function that causes the performance issue. Table 3.3 lists the results. vProf outperforms all other tools, ranking the root cause function within the top five (2nd

Name	Description and Configuration
`gprof`	Version 2.34 with glibc-2.31, default options used.
`perf`	Version 5.11.22, default options used.
`perf`-PT	`perf` with top-10 functions re-ranked using control-flow profiling: profile normal and buggy executions, Intel Processor Trace counts branches taken, calculate difference in branches taken per function for normal versus buggy executions, and use ratio of difference over total branches to scale top-10 function cost.
COZ	Determines which basic block if optimized further will improve overall performance the most; user identifies which functions to consider by identifying file that contains root cause function and top-level function in that file that will eventually call root cause function.
stat-debug	Records values of predicates, namely conditional statements and return values of functions, then ranks functions based only on how different the predicate distributions are between normal and buggy executions; user identifies file that contains root cause function and predicates only considered for functions in that file, 5 normal and 5 buggy executions used.
vProf	User identifies file that contains root cause function to limit number of variables sampled to that file, 5 normal and 5 buggy executions used for hist-discounter, but only one of each was used for variable-discounter.

Table 3.2: Configurations of tools to diagnose performance issues.

on average) in all 15 cases. In comparison, `gprof`, `perf`, `perf`-PT, COZ, and statistical debugging ranked the root cause function within the top five in only six, three, two, three, and two cases, respectively. In fact, vProf ranked the root cause function first in seven cases, more cases than the less precise top-five results for all of the other tools. In comparison, none of the other tools ranked the root cause function first in any of the cases, with the exception of `gprof` which did so for only two cases. Of all the tools, COZ performed the worst, failing to rank the root cause function in 11 cases, of which one was due to the tool crashing and four were due to its inability to support multiprocess applications.

For comparison purposes, Table 3.3 also shows the result when using vProf with zero variables monitored and only its hist-discounter (hist-disc), discussed in Section 3.4.1. hist-discounter alone reports the root cause function within top five for only three cases. This demonstrates the key vProf mechanism is not just comparing normal and buggy profiles, but doing so using variable value information, in conjunction with cost discounting using variable value information. Note

that the hist-discounter is still useful for large applications in which variables are only monitored in some components. For example, without hist-discounter, vProf has worse results for four cases, causing the ranking of the root cause function to drop from first to third in one case and dropping it out of the top five in two cases. Even without using hist-discounter for components without any monitored variables, vProf still far outperforms all other tools.

This observation that values are important for profiling is reinforced in comparing the results with vProf versus other tools such as statistical debugging or `perf`-PT. Statistical debugging also compares normal and buggy profiles, but uses only predicates, which may be noisy, without accounting for the actual function execution costs. Furthermore, statistical debugging requires the monitored predicates to be observed many times in both normal and buggy executions. In contrast, vProf uses variable value samples and conventional function execution costs, correlating them together with its analysis. Similarly, `perf`-PT compares normal and buggy profiles, but by monitoring control flow based on branch information as an alternative idea. Modern applications have abundant branches and many sources of non-determinism, so their control flow traces are noisy. In general, a performance issue may not be visible in control flow. For example, a performance bug that causes a loop to iterate many more times likely shows the same control flow as a normal execution. In fact, `perf`-PT, which enhances `perf` with control flow profiling, shows no overall improvement over just `perf`.

Table 3.3 also shows how effective vProf is in identifying the specific root cause basic block. Since vProf may report multiple basic blocks, we calculate the mean and minimum distance between the basic block reported by vProf and the one in which the developers fixed the bug. Shorter distances generally make diagnosis easier. Table 3.3 shows that in six cases, the basic block vProf reports in the root cause function is exactly where developers fixed the bug. For MDEV-13498, vProf did not report a basic block because DWARF did not provide sufficient information to map a PC sample of an anomalous value sample to basic blocks.

Furthermore, Table 3.3 shows how effective vProf is in classifying bugs using its bug patterns. vProf infers correct bug patterns for 13 out of 15 cases. It misses the bug pattern in Redis-10310

ID	vProf			Other Tools					
	rank	bb-dist	bug-patten	gprof	perf	perf-PT	COZ	stat-debugging	hist-discount
b1	1st	5, 0	✓	454th	32nd	32nd	NR	4th	447th
b2	1st	7, 0	✓	5th	2nd	2nd	NR	12th	1st
b3	1st	n/a	✓	2nd	3rd	6th	1st	30th	177th
b4	3rd	9, 0	✓	21st	9th	5th	NR	18th	31st
b5	4th	0, 0	✓	13th	4th	9th	NR	566th	22nd
b6	5th	19, 0	✓	36th	13th	13th	NR	NR	15th
b7	3rd	0, 0	✓	182nd	1024th	1024th	crash	7th	181st
b8	1st	0, 0	✓	1st	6th	7th	child	3rd	6th
b9	2nd	21, 0	✓	11th	28th	28th	NR	9th	11th
b10	1st	0, 0	✓	4th	16th	16th	child	161st	4th
b11	1st	0, 0	✓	1st	10th	10th	2nd	NR	59th
b12	1st	7, 5	✓	5th	19th	19th	1st	8th	2nd
b13	2nd	0, 0	NC	16th	13th	13th	9th	NR	33rd
b14	4th	17, 0	✓	NR	163rd	163rd	child	13th	NR
b15	3rd	2, 0	NC	14th	56th	56th	child	18th	8th

Table 3.3: Diagnosis effectiveness of tools. NR denotes the root cause function was not ranked, crash denotes the tool crashed, and child denotes the tool failed diagnosis because the root cause function was run in a child process. For vProf, *bb-dist* shows the (mean, minimum) distance between the basic block vProf identified and the root cause, and *class* shows whether the bug pattern reported matched the root cause; NC denotes the root cause could not be classified.

because the identified variable invokes a function pointer and has no labels. Similarly, it misses the bug pattern in Postgres-14b1 because of missing information on a variable that is stored inside a class pointer.

Case Studies. This session describes typical cases in further detail, focusing on how vProf compares to `gprof`, the tool on which it is based. These cases demonstrate how the sampled values help developers.

MDEV-21826: This is the example in Figure 3.1. `gprof` ranks `recv_apply_hashed_log_recs` first, while the actual root cause function `recv_group_scan_log_recs` ranks 454th. vProf ranks the root cause function first, promoting it based on its monitored variables `available_mem` and `pool_free_frames` using the variable-based execution cost, and demoting 44 other functions based

on its variable-discounter. vProf assigns a zero discount ratio to `recv_group_scan_log_recs` as its value samples are quite different between the normal and buggy executions, as shown in Figure 3.6a. vProf calculates high discount ratios for many other functions. For example, variables such as `end_lsn` have no significant differences in their distributions between normal and buggy executions, discounting the cost of `recv_apply_hashed_log_recs`. Furthermore, vProf translates the PC of the anomalous variable sample into lines and corresponding basic blocks. One of the line numbers is right before the while loop in `recv_group_scan_log_recs`. The basic block distance is zero.

MDEV-23399: In this case, the root cause function `buf_LRU_free_from_common_LRU_list` is ranked 5th by `gprof`, following functions such as `buf_page_get_low`, `buf_LRU_get_free_block`, `rw_lock_s_lock_spin`, and `MYSQLparse`. Upon analyzing the case with vProf, we observed that `buf_page_get_low`, `rw_lock_s_lock_spin`, and `MYSQLparse` are inherently costly. Their costs are reduced due to similarities in their ranks between normal and buggy executions. However, the root cause function exhibits different value distributions for the variable `scanned`: no value samples are captured in normal executions, while large values are recorded in the buggy cases. The variable is used within a loop, identified by `loop` tag. Consequently, vProf identifies the bug as a `Scalability` issue. In reality, the values of `scanned` are the crucial information that developers have extensively examined during their manual debugging.

MDEV-13498: The root cause of this issue is that the `std::find` searches a list unnecessarily when the database is not in a cluster. The function `row_upd_sec_index_entry` and function `row_upd_del_mark_clust_rec` are the culprits responsible for the improper invocation of `std::find`. While `gprof` correctly ranks the function `row_upd_sec_index_entry` at 2nd in the profiling result, it overlooks the function `row_upd_del_mark_clust_rec` due to profiling bias. However, vProf applies hist-discounts to 12 functions which exhibit high cost in both normal and buggy cases. As a result, `row_upd_sec_index_entry` is ranked 1st. Furthermore, vProf calibrates the cost of `row_upd_del_mark_clust_rec` using value samples, elevating its position to 5th in the list. vProf identifies an anomaly in the processing time of the variable `parent` inside the root cause functions.

The variable is accessed in the same conditional statement where the `std::find` is evaluated.

MDEV-15333: In this case, the database upgrade causes unnecessary table validation during database restart. `gprof` ranks the root cause function `fil_ibd_open` at 21st, which unnecessarily invokes `fsp_flags_try_adjust`, ranked at 165th. This performance issue arises when the number of tables is large. For comparison, we have a baseline with a small number of tables. vProf adjusts function costs for 11 inherently costly functions, such as `buf_chunk_init`. These functions are also costly during normal startup. After applying discounts to functions based on the distribution of variable values, vProf ranks `fil_ibd_open` at 3rd. For instance, the function `fil_node_prepare_for_io` is adjusted with a high discount of 0.8 due to similar distributions of processing times for all selected variables. In addition, the cost calibration also elevates `fsp_flags_try_adjust` from 165th to 6th. While vProf ranks the functions `fil_node_complete_io` and `fil_io` at the top after applying a default discount of 0.8, such false positives can be identified by investigating the code logic. They are inherently much costlier due to a larger number of tables in the buggy execution.

MDEV-17933: MariaDB server takes a long time to display the engine status when there are many active transactions in the system. To reproduce the issue, we simulate query transactions concurrently in the background while querying the system status. The root cause function is `dict_sys_get_size`, where the server caculates memory usage statistics by traversing the data dictionary cache. In the profiling result, `gprof` ranks the root cause function 13th. It is proceeded by inherently costly functions, such as `ut_delay` and `bug_calc_page_new_checksum`. vProf removes nine inherently costly functions with hist-discounts. As a result, the root cause funtion is ranked 4th, following `ut_dealy`, `sync_array_cell_print`, and `sync_array_print_long_waits`. All these false positive functions are symptomatic of the performance issue. They produce warnings of long waits because the root cause function `dict_sys_get_size` holds a mutex and blocks other threads unnecessarily, preventing background transactions from acquiring the mutex.

HTTPD-62668: Filters in HTTPd are configured to process a sequence of data buckets. They are maintained in a filter chain until encountering an EOS(end of stream) data bucket. The root

cause in this case is that the core request filter bails out on EOS data bucket without cleaning up all related data structures, leaving the filter function `ap_request_core_filter` in the filter stack. When the function `ap_filter_output_pending` iterates over all the filter functions in the filter stack, it invokes `ap_request_core_filter` and illegally accesses either corrupted or reused memory. Consequently, the updates of induction variable in `ap_filter_output_pending` will invalidate the conditional statement designed to break its `for-loop`, resulting in an infinite loop.

`gprof` ranks the root cause function `ap_filter_output_pending` at 14th and the related function `ap_request_core_filter` at 36th. In this case, the first important clue needed to figure out is whether `ap_filter_output_pending` has an infinite loop or is invoked repeatedly. vProf ranks `ap_filter_output_pending` at 5th after adjusting the function costs, and indicates its local variable `c`, of type `conn_rec`, experiences an unexpected long processing for a fixed value inside `for-loop`. Such information confirms that the issue is related to the loop. Further, vProf reveals the anomalous values are accessed inside the loop and labels it as a Missing Constraint bug.

HTTPD-54852: The root cause in this case is the cascading polling cost for child processes. When the main process gracefully restarts a child process that has already exited, it will poll for a second in `dummy_connection`. During the polling period, more child processes will exit, leading the main process to poll for all the exited child processes. As a result, the graceful restart takes a few minutes.

The core issue lies within the function `ap_mpm_pod_killpg`, which invokes `dummy_connection` for all children without checking their status. `gprof` ranks `ap_mpm_pod_killpg` at 182nd, and `dummy_connection` is not ranked because `gprof` does not consider samples about socket operations in libraries. Conversely, vProf ranks `dummy_connection` at 1st, `ap_mpm_pod_signal` at 2nd, and `ap_mpm_pod_killpg` at 3rd after calibrating the function costs using value samples collected by virtually unwinding the callstack. For the root cause function `ap_mpm_pod_killpg`, vProf identifies an anomaly in the processing costs of the variable `i` within its loop. The false positive `ap_mpm_pod_signal` is identified by the anomalous value of the variable `ap_daemons_max_free` from the configuration. Such false positive can be excluded by examining the source code or attempting

the buggy executions with a smaller value of the variable `ap_daemons_max_free`.

HTTPD-62318 : This case is related to a multithread error that is notoriously hard to diagnose. If threads are enabled and one of them takes an unusually long time to finish a health check, it will update the finishing time of the nearest health check with a past timestamp when the health check begins. This past value causes watchdog to invoke a new health check request for every heartbeat because it assumes the last health check happened a long time ago.

Ideally, the useful information for diagnosis can be the unusual health check in the function `hc_check` and the variables used in a conditional statement inside `hc_watchdog_callback`. Although `gprof` ranks `hc_watchdog_callback` at 1st already, it does not provide hints for developers to figure out the cause of the unusually frequent invocations of `hc_check`.

vProf ranks `hc_check` at 1st and indicates abnormal return values from its callee `hc_check_http`, which is ranked 2nd. The return value is used in a conditional statement in the function `hc_check`, and thus vProf marks the case as a `WrongConstraint` issue. Additionally, `hc_watchdog_callback` is ranked 3rd and also marked as `WrongConstraint` due to the abnormal delta values of the variable `now`. The former information about `hc_check` is more helpful in diagnosing the performance issue, as it suggests that the problem is triggered by a corner case where a health check takes too long. Without this information, developers simply moved the code updating the last health check timestamp upwards at the beginning of the health check, which cannot fix the bug ultimately, as documented in the related bug report at https://bz.apache.org/bugzilla/show_bug.cgi?id=63010

HTTPD-64066: This is a typical case related to scalability, where repeated computations for thousands of virtual hosts cause a slow start in HTTPd. The root cause is that each virtual host reads and searches commands in its HTTPd config file using `ap_find_command_in_modules`, which, in turn, invokes `ap_cstr_casecmp` to compare strings. The performance issue arises from the accumulated time cost. `gprof` ranks the root cause function `ap_find_command_in_modules` at 11th. Preceding it are functions related to string operations.

To diagnose the issue with vProf, we configured HTTPd with a smaller number of virtual hosts as a normal case. Comparing function rankings between the normal and buggy cases for

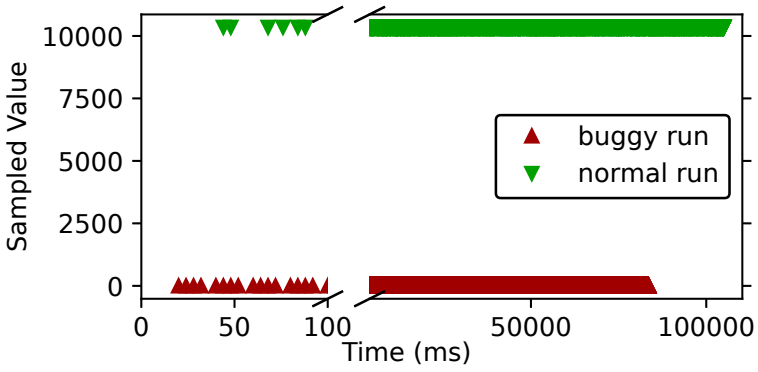
calculating hist-discount is not helpful because no similar function rankings are detected. On the other hand, based on similarities in values of variables, vProf discounts the inherently costly string-comparing functions. It also calibrates the cost of the root cause function based on the number of value samples collected. The cost adjustment promotes `ap_find_command_in_modules` to the 2nd in the result. vProf identifies it as a `Scalability` issue because the distributions of processing times for variables in both cases are similar, and the root cause function remains costly after applying the discount value of 0.8.

HTTPD-52914: When the HTTPd server attempts to discard a response body, it calls the function `ap_discard_request_body` to read and test any message body in the request before discarding it. This process involves a two-dimensional loop that terminates either upon encountering an EOS bucket or when an input filter returns an error value indicating a failure in passing the data bucket of the message.

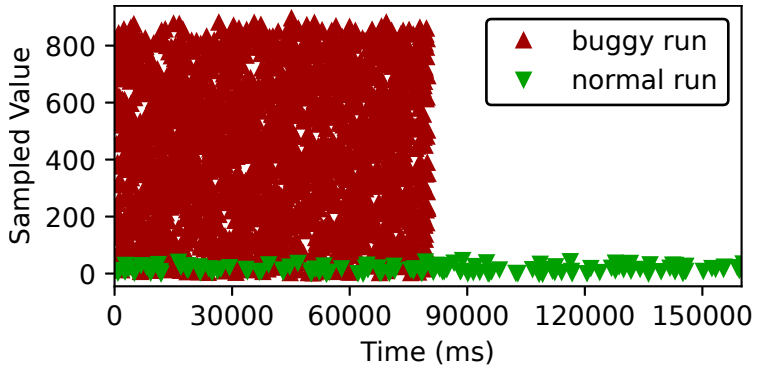
In this case, the connection between the server and the client times out while the client has a POST request delivering a large body content. However, the filter function `dumpio_input_filter` incorrectly returns APR_SUCCESS even when it fails to retrieve data buckets from the next filter function in the filter stack. The incorrect return value leads to a loop in `ap_discard_request_body`.

`gprof` ranks the root cause function `dump_io_input_filter` at 4th and `ap_discard_request_body` at 8th. However, the ranking is misleading without data flow. Developers often focus on verifying the correctness of the loop logic. According to the bug report, the reporter submitted a wrong patch for the function `ap_discard_request_body` because the attached debugger indicated that `ap_discard_request_body` repeatedly invoked `dumpio_input_filter`.

vProf ranks `dump_io_input_filter` at the top and reports a zero discount for the function based on the value of the variable `readbytes`. It has many value sample of a fixed value in buggy case, but much fewer samples of value zero in the normal case. The anomalous value samples in the buggy case are recorded in the basic block with a conditional statement, so vProf identifies the issue as a `WrongConstraint`. The basic block is exactly where an error return value should have been emitted. Thus, vProf reports a block distance of zero.



(a) samples of `available_mem`



(b) samples of `numclients`

Figure 3.6: Value samples of key variables in *MDEV-21826* and *Redis-8668* respectively.

Redis-8145: In a Redis cluster with many nodes, the command `cluster nodes` queries mapping information from cluster hash slots to actual nodes, aiming to acquire the list of hash slots served by each node in the cluster. This case is similar to *HTTPD-64066*, associated with a large number of instances configured in the system.

Both `gprof` and vProf place the root cause function `clusterGenNodesDescription` at 1st, but vProf also indicates the bug pattern and the locations where value samples are collected. Since all sampled varibles are of pointer type and their processing costs are similar between normal and buggy cases, vProf reports the bug pattern related to `Scalability`, with a default discount 0.8.

Redis-8668: `gprof` ranks functions from the `zmalloc_*` family and `dictEncObjKeyCompare` above the root cause function `serveClientsBlockedOnKey` which is ranked 5th. vProf ranks the root cause function 1st, demoting other functions based on its hist-discounter and keeping the root cause function highly ranked based on its variable-discounter. vProf finds the `zmalloc_*` are inherently costly

in both normal and buggy executions, have no variables being monitored, so its hist-discounter assigns a discount ratio of 1.0 to them. For similar reasons, `dictEncObjKeyCompare` is assigned a discount ratio of 0.76.

vProf assigns a zero discount ratio to the root cause function as its value samples for `numclients` are quite different between the normal and buggy executions, especially in terms of processing costs. Specifically, Figure 3.6b shows that the distribution of the value samples in normal versus buggy executions are different, but this results in a discount ratio of 0.12. Instead, the distributions based on processing costs are even more different, resulting in a discount ratio of zero, which vProf uses since it is the smaller of the two. Furthermore, vProf translates the PC of the anomalous variable sample into lines and corresponding basic blocks. One of the line numbers falls in the invocation of `listRotateHeadToTail`, which makes up the costly part of a while loop in `serveClientsBlockedOnKey`. The mean basic block distance to the while loop is five.

Redis-10310: This is a performance regression observed after upgrading Redis. In this case, the function `genericZrangebyrankCommand` inappropriately employs deferred replies, even when the result size is already known. This leads to unnecessary computations in the heap. `gprof` ranks functions responsible for managing sorted sets and memory allocations, such as `je_malloc`, ahead of the root cause function `genericZrangebyrankCommand`.

vProf profiles the normal and buggy executions across two different versions of Redis. It elevates the root cause function `genericZrangebyrankCommand` to 2nd by applying hist-discounts to inherently costly functions. The discount for the root cause function is zero because the function and variables are unique to the buggy version of Redis.

vProf also highlights anomalous values in the variable `handler`, which is dereferenced to invoke `beginResultEmission` and `finalizeResultEmission`. Although vProf fails to directly infer the bug pattern due to `handler` being an argument for function invocation without labels, it recognizes the difference in the implementation of the `ZREVRANGE` command. This information provides insights to pinpoint the functions contributing to the observed performance regression.

postgres-17330: PostgreSQL employs the function `get_actual_variable_range` to determine

the maximum and minimum values of a variable. This information allows PostgreSQL to estimate costs for query plans. To prevent the inflation of value ranges due to dead tuples, PostgreSQL ignores the values of dead tuples. This is achieved by specifying the snapshot type within the `get_actual_variable_range` function.

Basically, when accessing a data page, PostgreSQL can skip the tuples in the middle if the first and last tuples on the data page are active. This is because the tuples are ranked. However, if dead tuples are present at the beginning or end of the data page, `get_actual_variable_range` invokes `_bt_readpages` to traverse and ignore the dead tuples one by one, due to the improper snapshot type.

In this case, `gprof` ranks inherently costly functions such as `_bt_checkkeys`, `pglz_decompress` and `SearchCatCache` at the top but misses the function `get_actual_variable_range`.

vProf compares the performance issue with the same query upon a database with fewer deleted tuples. As a result, the normal case has fewer data pages with dead tuples at the top and bottom. In this scenario, vProf applies hist-discounts to inherently costly functions: 1.0 on `pglz_decompress`, 0.76 on `SearchCatCache`, which lowers their ranks. Furthermore, vProf calibrates the costs for `_bt_readpage` and `get_actual_variable_range` based on value samples, assigning a zero discount ratio to `_bt_readpage` and 0.8 to `get_actual_variable_range`. Therefore, vProf ranks the root cause function `get_actual_variable_range` at 4th and identifies it as a `Scalability` issue. This bug pattern is drawn from the similarity in processing costs for local variables inside the function in both normal and buggy cases. Regarding the false positive, `_bt_readpage`, ranked at the top, analyzing its code logic helps developers understand the expensive scans on data pages.

postgres-14b1: This performance issue occurs during PostgreSQL reclaims stale tuples and releases memory through its autovacuum process. The user reported that an autovacuum worker consumed 100% CPU after `reindex` and `analyze` had been running concurrently for a few minutes. The problematic execution takes place in the function `lazy_scan_prune`. It invokes `heap_page_prune` to delete non-indexed dead tuples first and then scans the page to collect indexed dead tuples. During the scan, the function `HeapTupleSatisfiesVacuum` identifies a deletable non-indexed tuple and

restarts the process from `heap_page_prune`. However, `heap_page_prune` identifies that the tuple is dead but not deletable. Therefore, `lazy_scan_prune` loops between the two functions back and forth.

The traditional profiler, `gprof`, ranks the root cause function `lazy_scan_prune` at 14th. Functions ranked before it are its callees, including `heap_page_prune` and `HeapTupleSatisfiesVacuumHorizon`. Without hints on the data flow, it is hard to determine whether a busy loop is specific to a particular tuple or inherently a hot code path for dead tuples.

vProf compares the profiling data from the buggy run with a normal run where `reindex` and `analyze` are executed sequentially. After adjusting function costs based on the value samples, vProf ranks the root cause function at 3rd and reports an anomaly in its local variable `tuple.t_data`.

In the buggy case, all recorded samples are for a single tuple, while in the normal case, different values are sampled, and each of them costs much less time. This information indicates that the transaction states stored in the tuple can be problematic and are related to the inconsistent behaviors in the functions `HeapTupleSatisfiesVacuum` and `heap_page_prune`.

An ideal diagnosis tool needs to report the specific data flow within the tuple that causes the discrepancy. However, vProf fails to provide this very data flow because it does not track the variables stored via pointers of complex classes, such as `vistest->maybe_needed` and `vacrel->OldestXmin` in this case. Adding support for these complex pointers would introduce overhead for sanity checks and negatively impact the scalability of the profiler. We leave this problem for future work.

False Positives. Like all profilers, vProf cannot guarantee that the root cause function is always ranked first. Fortunately, a performance issue often involves multiple functions, which are also helpful for performance diagnosis. For example, in HTTPD-54852, vProf ranks `dummy_connection` above the root cause function `ap_mpm_mod_killpg`. However, `dummy_connection` is called by the root cause function, so revealing that function in addition to the root cause function can help with performance diagnosis since the root cause function is still highly ranked. This connection is less clear with `gprof`, which ranks the root cause function well outside its top 100 ranked functions.

However, if the top ranked functions are unrelated to a performance issue, they can waste

developers' investigation time and are considered false positives. For vProf, we computed the false positive ratio for each issue by counting the number of functions unrelated to the performance issue before the developer reaches the root cause function and dividing that by five. The false positive ratio would be 100% if all top five ranked functions are unrelated to the performance issue. Across all 15 cases, the average false positive ratio was only 10.6%. Given that vProf ranked the root cause function first in almost half the cases and in the top five in all cases, this means that when vProf does not rank the root cause function first, on average at most one other function was ranked ahead of the root cause function that was unrelated to the performance issue.

The false positive ratio does not imply that the developers would necessarily waste time investigating unrelated functions, which depends on the sources of false positives. First, an inherent costly function can be top-ranked even though it has a high discount ratio. For example, in MDEV-17933, vProf ranks the function `ut_delay` first but with a high discount ratio. In such cases, the discount ratio indicates the function is inherently costly in normal and buggy cases, so the developer can consider it lower priority to investigate. Second, some functions are costly as a side effect of a buggy execution. For example, in HTTPD-62668, vProf ranks the function `listener_thread` first because it takes a long time in the buggy case waiting for a request timeout, but it returns immediately in the normal case. Such false positives are hard to eliminate but usually help confirm the causes of performance issues. Third, false positives can also be due to the limitations of statistical methods. Developers can exclude such functions by verifying the annotated bug pattern or increasing the accuracy with repeated experiments.

3.5.2 Performance Overhead

We measured the memory and runtime overhead when using vProf to profile buggy executions of the performance issues in Table 3.1. For each case, Table 3.4 shows how many variables were monitored, the time for initializing the vProf-specific profiler data structures, how much memory was consumed by vProf during profiling to store metadata and value samples, and the time to profile the buggy execution. In almost all cases, vProf monitored hundreds of variables for a

ID	Variables	Init Time	PCToVar Table	Variable Array	Value Samples	Run Time
b1	233	7.4 ms	3862 KB	430 KB	21133 KB	105 s
b2	65	0.9 ms	4143 KB	29 KB	153 KB	1903 s
b3	399	0.4 ms	4005 KB	26 KB	38563 KB	1140 s
b4	852	15.9 ms	3987 KB	67 KB	58 KB	338 s
b5	577	18.2 ms	3575 KB	22 KB	8 KB	1635 s
b6	501	31.1 ms	673 KB	287 KB	2 KB	1448 s
b7	113	0.3 ms	162 KB	6 KB	16 KB	147 s
b8	169	4.5 ms	260 KB	127 KB	43 KB	553 s
b9	374	6.2 ms	194 KB	16 KB	25 KB	36 s
b10	164	1.4 ms	642 KB	186 KB	13 KB	139 s
b11	531	3.4 ms	612 KB	382 KB	1216 KB	885 s
b12	623	5.5 ms	591 KB	44 KB	1755 KB	112 s
b13	564	7.1 ms	641 KB	754 KB	132 KB	10 s
b14	479	5.2 ms	2037 KB	1031 KB	79 KB	68 s
b15	805	6.4 ms	2297 KB	927 KB	3269 KB	29 s

Table 3.4: Memory overhead and execution time for profiling performance issues.

program component. In all cases, vProf-specific profiler initialization was fast enough to appear instantaneous to a user, and memory overhead was small for vProf's core data structures except in some cases for storing variable samples, which scales based on the number of samples recorded. We further measured the application memory footprint under profiling with vProf and gprof. The maximum memory footprint with vProf scales as expected based on the measurements in Table 3.4, but the difference versus `gprof` is modest overall. For example, MDEV-13498 has the largest memory footprint, but vProf's maximum memory footprint is only 8% larger than `gprof`. On average, the maximum memory footprint with vProf is 7% (8 MB) larger than with `gprof`.

Figure 3.7 shows the runtime overhead of vProf when profiling each performance issue, with performance normalized to execution without using the profiler. For comparison, we also measured the runtime overhead of `gprof` on these issues. vProf runtime overhead is modest in all cases except for when `gprof` overhead is higher, in which case vProf overhead tracks that of `gprof`, on which it is built. We also used sysbench to measure the latency and throughput of MariaDB under a

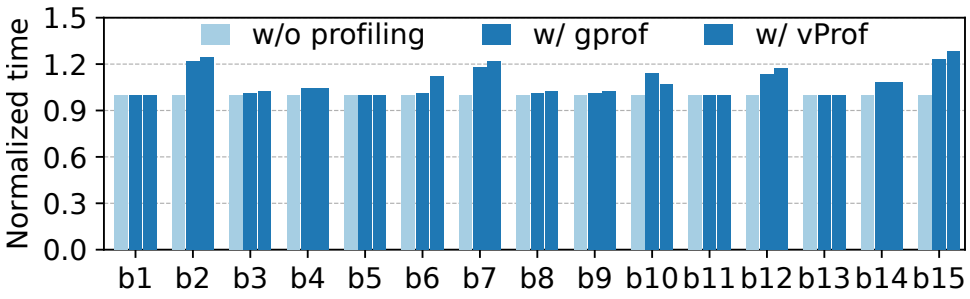


Figure 3.7: Profiling overhead for performance issues.

TPCC workload, with and without profiling. Both vProf and `gprof` incurred the same latency and throughput overheads, 32% and 20%, respectively; vProf shows no increased overhead for the features it adds. Overall, these results show that vProf is lightweight and practical for diagnosing performance issues in large applications.

vProf also incurs some cost for its schema generator and post-profiling analysis, which we quantified for the 15 issues in Table 3.1. vProf's LLVM pass increases compilation time by an average of 5 s. Using DWARF debugging information to obtain variable metadata takes an average of 142 s. Post-profiling analysis takes an average of 117 s. If we monitor variables across the entire program instead of per program component, analysis can take much longer. For example, doing so for Redis-8145 resulted in 17,930 variables being monitored and 7 GB of value samples being recorded, which took post-profiling analysis roughly six hours to process.

3.5.3 Sensitivity

We evaluated how vProf's effectiveness is affected for the 15 issues in Table 3.1 for different values of DefaultDiscount and ValidDiscount. We measured effectiveness by how many issues had their root cause function ranked in the top five. We first used the default ValidDiscount of 0.1 and set the DefaultDiscount to different values between 0.1 and 1.0. We then used the default DefaultDiscount of 0.8 and set the ValidDiscount to different values between 0.1 and 1.0. Figure 3.8 shows that vProf is most effective with a DefaultDiscount of at least 0.8 and a ValidDiscount of less than 0.3.

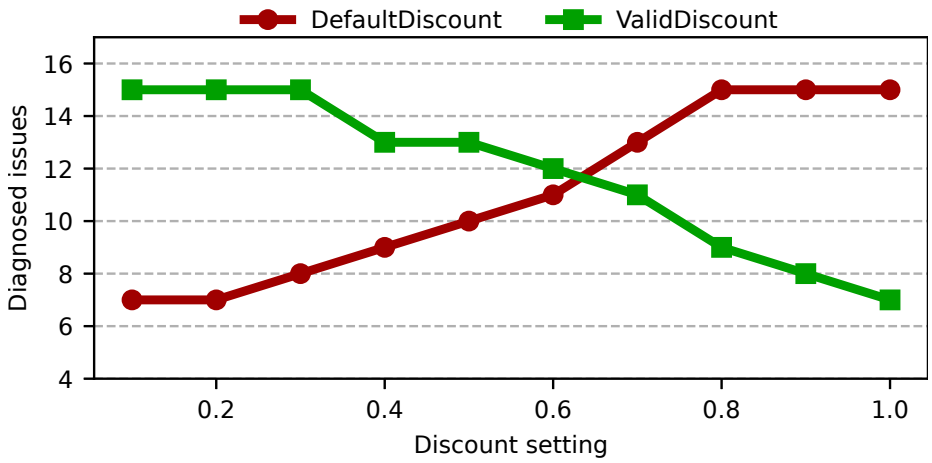


Figure 3.8: Sensitivity of settings for discount parameters.

3.5.4 Diagnosing Unresolved Issues

We further used vProf on three real unresolved performance issues to demonstrate its effectiveness at diagnosing unknown root causes in practice. These issues are listed in Table 3.5.

Redis-10981: Developers investigated the issue by bisecting their commits but could not draw a definitive conclusion for the performance degradation in version 7.0.3. In both 7.0.3 and the earlier version, traditional profilers attribute the highest costs to functions `_addReplyToBuffer` and `addReply`. Comparing the ranking of functions in profiling reports from the two versions does not provide useful information either.

We used vProf to diagnose the performance issue, which had remained unresolved for more than six months. We first investigated the component `db.c`. vProf ranks its function `lookupKey` first. It shows that the variable `key` has different processing costs and sampled values in the buggy version. Looking into the code, we found that function `expireIfNeeded` was moved into `lookupKey`. The code refactoring caused a longer execution time and different values samples, leading to a false positive.

We next investigated the component `networking.c`. Although `_addReplyToBufferOrList` is the function ranked first in vProf, it is new in 7.0.3 due to code refactoring, so we excluded it from further consideration. vProf ranks the function `clientHasPendingReplies` second as the processing

ID	Description	Date
Redis-10981	`lrange` command takes longer to finish when `redis` is upgrade from version 6.2.7 to 7.0.3	07-14-2022
MDEV-16289	Query runs unexpectedly slow; the query selects records created within a given time period in one table, and excludes records whose certain fields are after a given time by checking another table.	05-25-2018
MDEV-17878	Searching for the query execution plan for a SELECT query involving many joins takes forever for larger datasets, using 100% CPU	11-30-2018

Table 3.5: Unresolved performance issues diagnosed using vProf.

cost for its variable `client` differs in the two versions. vProf reports the anomalous variable samples are accessed in a conditional expression. The condition was introduced in 7.0.3. We verified our findings by reverting this condition, which caused the performance degradation to disappear. vProf successfully identified the unresolved issue that was unable to be clarified previously using the commit-bisecting method or traditional profilers. We reported our findings to the developers, who quickly confirmed the diagnosis.

It took about four-person hours per component to generate schemas for a specified program component, run test cases with vProf, and investigate the source code based on the vProf reports. Since we investigated two components, the total time to diagnose the performance issue was eight person-hours.

MDEV-16289: A developer reproduced the issue and reported that different timezone settings caused different processing costs, identifying it as a performance bug because he believed the query results should be independent of the timezone. In trying to diagnose the issue, the developer traced the query execution plans for two different timezone settings, but the results were similar and provided limited hints for further debugging.

We used vProf to diagnose the performance issue, which had remained unresolved for more than four years. We investigated the component `row0sel.cc`, which implements row selection in MariaDB. The function `row_search_mvcc` was ranked first. Although this function is costly

whether or not the query runs slow, its discount ratio is zero because the sample distributions for local variable `clust_index` differ between fast and slow queries. No value samples are captured when the query is fast, but over 30 are captured when the query is slow. We also noticed a similar issue for the variable `result_rec`. Both variables appear to be pointers to temporary storage of intermediate query results.

Because references to additional temporary storage only appear when the query runs slow, we suspected the queries might return different numbers of records for different timezone settings. We verified our hypothesis by changing the query's timestamp to refer to the same absolute time in different timezones. For example, instead of querying with 8pm in all timezones, we queried with 8pm EST and 5pm PST. By doing the latter, the difference in query performance disappeared. We further confirmed our hypothesis by checking the number of records returned; many more records were returned for the slow query case. Contrary to the developer's belief, this issue turned out not to be a performance bug, but correct operation with different query times for what are actually different queries. Diagnosing the issue using vProf took roughly five person-hours. We reported the findings to the developer.

MDEV-17878: The user who reported the issue also profiled the issue using `perf`, which ranks function `prev_record_reads` first. In trying to diagnose the issue, developers obtained query execution plans from both MariaDB and a different version of MySQL that finishes the query quickly. The information obtained did not provide enough hints for the developers to diagnose the performance issue.

We used vProf to diagnose the performance issue, which had remained unresolved for more than four years. We identified the program component involved in optimizing the query execution plan and monitored its variables using vProf. We then needed to profile a useful normal execution, which took us three tries. First, because the report indicates that the performance issue does not occur for small datasets, we created a small dataset to profile a normal execution. However, the query finished too fast and resulted in no value samples being collected. Second, we took the original dataset causing the bug and reduced the number of joins so that the performance issue dis-

appeared. vProf ranked the functions `best_access_path` and `best_extension_by_limited_search` first and second, respectively; the latter calls the former. However, vProf set both their discount ratios to DefaultDiscount, indicating a lack of anomalous value samples.

Finally, because the report was specific to a version of the application, we tried a different version with the original dataset that caused the bug and found that the performance issue disappeared. We used this different version with the original dataset as the normal execution. In this case, vProf ranked the function `best_extension_by_limited_search` first. vProf labels it a `Missing Constraint` bug because of anomalous value samples for `use_condition_selectivity`, which is used in a conditional expression. This variable value comes from the system variable `optimizer_use_condition_selectivity` in `sys_vars.cc`, which has different default values for different versions of MariaDB. `use_condition_selectivity` decides the heuristics used to estimate the cost of the current partial query plan. The query plan search algorithm stops if the cost is greater than the current best heuristic. However, if the value of `optimizer_use_condition_selectivity` is one by default, the search algorithm fails to stop searching through more costly heuristics to find a better plan.

Diagnosing the issue using vProf took roughly 12 person-hours, eight of which were for going through the three approaches to profile a normal execution, and four of which to investigate the source code. In this and the other cases, the process could be faster for actual developers who are familiar with the program source code. This case also shows how using a different program version can be useful to profile a normal execution. We reported the root cause to developers, who confirmed our diagnosis and updated the issue ticket to include our reported root cause.

3.6 Conclusions

Value-assisted cost profiling is a new profiling methodology that provides effective diagnosis of performance issues in real-world applications. It measures execution costs together with program data-flow information to more accurately reason about whether a costly function is necessary and why a function is slow. vProf is a practical tool that implements this methodology. It leverages

static analysis to identify variables that commonly influence performance and determine their runtime locations. It builds efficient data structures for profiling to quickly index accessible variables and continuously records value samples with PC sampling. It provides post-profiling analysis to compare value samples across normal and buggy program executions to identify anomalous samples, use them to calibrate function costs, and pinpoint root causes. vProf significantly outperforms other state-of-the-art tools in diagnosing real-world performance bugs in large applications, yet incurs only modest performance overhead. We used vProf to diagnose longstanding unresolved performance issues in real applications, which have been confirmed by developers.

Chapter 4: Annotated Causal Tracing for Modern Desktop Applications

Diagnosing performance issues can be extremely challenging in modern desktop applications. These applications are often built using assorted frameworks and libraries to break down the handling of user interface (UI) events into numerous small execution segments [31], which run concurrently on multi-core hardware, to ensure responsiveness. For instance, macOS applications manage UI events by sending messages to delegate objects containing code that reacts to these events asynchronously. The messages are generated by the closed-source Cocoa framework [18], which, in turn, interacts with the operating system (OS), daemons, and other libraries. The predominantly asynchronous and concurrent interactions complicate the identification of the root cause of a performance anomaly.

In Chapter 3, we proposed `value-assisted cost profiling`, a method in which data flow is recorded during profiling, to assist developers in identifying the root causes of performance issues. While profilers can excel at analyzing individual components, they are not well-suited to troubleshoot intricate performance issues in modern desktop applications, which often involve high degrees of concurrency and communication. Profilers lack the capability to analyze the causal relationships that span across numerous frameworks, libraries, system daemons, the kernel, and applications.

To understand the causalities across components, causal tracing [15, 6, 63, 72, 26, 68, 40, 49, 67, 50, 60, 80] was initially proposed. It generates trace graphs in which *vertices* represent *execution segments* containing system activities, such as user operations, system calls, or messages, and *edges* indicate causal relationships between vertices. To diagnose a performance issue, developers usually conduct critical path analysis on the trace graph to identify an end-to-end path that takes the greatest amount of time.

Unfortunately, we have observed that previous causal tracing approaches are ineffective for

desktop applications, as they cannot accurately identify the boundaries of execution segments and their causal relationships. For instance, a long-standing performance anomaly in the Google Chrome web browser [17] on macOS occurs when a user enters non-English words in the search box, causing Chrome to hang with the infamous macOS spinning pinwheel—a visual indicator that the application is unresponsive to user input. Employing previous approaches to construct trace graphs for the multi-threaded, multi-process browser results in many missing execution segments and numerous additional irrelevant execution segments. Attempting to diagnose the issue using these incomplete and inaccurate graphs would lead to incorrect identification of events as the culprit. In theory, these tracing inaccuracies could be rectified by adding instrumentation, such as incorporating constraints in noisy trace points to filter out irrelevant events. However, frameworks and libraries used by desktop applications encompass diverse programming idioms and are often closed-source, rendering deep instrumentation challenging. In addition, extensive instrumentation would also entail prohibitive overhead, resulting in unacceptable performance.

To address these challenges, we have developed Argus, a causal tracing tool specifically designed to assist users in diagnosing performance anomalies in desktop applications. Argus is built upon the insight that tracing inaccuracies are inherently inevitable in real desktop systems. Instead of attempting to eliminate all inaccuracies, our approach is to design tracing solutions that can accommodate certain inaccuracies. Guided by heuristics derived from supplementary runtime information, Argus introduces a novel concept of *annotated* trace graphs. In this approach, edges are explicitly and automatically marked as *strong* or *weak* edges. Strong edges represent connections among segments that adhere to typical programming paradigms that must be causal, such as sending and receiving an IPC message. Weak edges, on the other hand, represent uncertain relationships among segments. For instance, when one thread awakens another thread, it could be a causal relationship, such as a *lock*/*unlock* mechanism, or just an artifact of regular OS scheduling. To further enhance its effectiveness, Argus boosts or prunes unnecessary weak edges by leveraging operation semantics and call stacks collected at runtime.

Argus introduces a new beam search diagnosis algorithm based on edge strength and a novel

method of comparing trace subgraphs across normal and abnormal executions of an application. The algorithm is motivated by our observation that critical path analysis used in prior work is ineffective due to inaccuracies inherent in trace graphs. Beam search embraces more possibilities while exploring the annotated noisy trace graph. Our algorithm efficiently selects likely causal paths in the massive trace graph and tolerates noises. Comparing trace subgraphs across normal and abnormal executions also helps with diagnosis when the problem is due to missing operations in the abnormal execution.

Argus provides system-wide tracing by extending existing tracing support in the OS kernel and applying binary patching for low-level libraries. This allows Argus to easily track objects across process boundaries, account for kernel threads involved in communications among processes, and cover customized programming paradigms by operating in a common low-level substrate used by higher-level synchronization methods and APIs that may be introduced and evolve over time. Argus does not require any application modifications.

We have implemented and evaluated a prototype of Argus across multiple versions of macOS. This presents a harsh test for Argus given the many complex, closed-source frameworks, libraries, and applications in the macOS software stack. We evaluated Argus on 12 real-world spinning pinwheel issues in widely-used macOS applications, such as Chrome, Inkscape, and VLC. Argus successfully pinpoints the root cause and sequence of culprit events for all cases. This result is particularly notable given that 10 of the 12 cases are open issues whose root causes were previously unknown to developers. Argus incurs runtime overhead low enough such that users can leave Argus tracing always-on in production without experiencing any noticeable performance degradation. Source code for Argus is available at `https://github.com/columbia/ArgusDebugger`.

4.1 Motivation and Observations

We experienced first-hand the Chrome web browser performance issue on macOS. Typing non-English words in a search box while a web page is loading causes Chrome to freeze and trigger a spinning pinwheel. The spinning pinwheel appears when an application is not responsive to user

input for more than two seconds. Others have also experienced this issue with the Chromium web browser and reported it to Chromium developers [17]; Chrome is based on Chromium.

We study the bug in Chromium since it is open-source, so we can verify its ground truth. Chromium is a multi-process macOS application involving a browser process and several renderer processes, each process having dozens of threads. When a user types a string in the browser search box, a thread in the browser process sends an IPC message to a thread in the renderer process, where the rendering view code runs to calculate the bounding box of the string, which in turn queries `fontd`, the font service daemon, for font dimensions.

To diagnose the bug, we first tried using `spindump` [1], a widely-used macOS debugging tool, which shows the main thread of the browser process is blocking on a condition variable. However, `spindump` provides no clue as to why the condition variable is not signaled. Using macOS Instruments [34] was also ineffective, as it simply analyzes what functions take the most time, which are not the root cause in this case. These traditional debugging and profiling tools are fundamentally not well suited to analyzing causality in highly concurrent execution flows across multiple components over time.

We next tried state-of-the-art causal tracing techniques. Specifically, we use Panappticon [80], a system-wide tracing tool originally built for Android. We reimplemented a version for macOS with more complete tracing of asynchronous tasks, using non-intrusive interposition to trace asynchronous tasks, IPCs, and thread synchronizations from the system and libraries. We use the tool when running Chromium and reproduce the anomaly by typing non-English search strings. After the browser handles the first few characters normally, the remaining characters trigger a spinning pinwheel. We then stop the tracing. The entire session took around five minutes.

Dividing up the trace graph into separate graphs each beginning from a user input event results in 359 trace graphs; user input events are dispatched from the macOS `WindowServer` process to Chromium. The trace graphs are highly complex, with 888,236 vertices and 751,332 edges in total. They span across 11 applications, 79 daemons including `fontd`, `mdworker`, `nsurlsessiond`, and various helper tools started by the applications. They cover 90 processes, 1177 threads, and

```
// worker thread in fontd:
block = dispatch_mig_server;
dispatch_async(block);
```

```
// main thread in fontd:
// dequeue blocks
block = dequeue();
dispatch_client_callout(
      block);
```

```
// implementation of dispatch_mig_server
dispatch_mig_server()
  for (;;) { // batch processing
    mach_msg(send_reply,recv_request)
    call_back(recv_request)
    set_reply(send_reply)
 }
```

Figure 4.1: Dispatch message batching. `dispatch_mig_server` can serve unrelated applications together.

644K IPC messages.

Studying the trace graphs, we observe: (i) connections exist between graphs from different UI events; (ii) some long execution segments have no boundaries; (iii) there are orphaned vertices with no edges; (iv) the trace graph that contains the anomalous event sequence triggering the spinning pinwheel contains 12 processes—3 are clearly unrelated to the transaction, and 6 are daemons whose relationships are unclear without further investigation. Based on further analysis of these graphs with call stacks and reverse engineering techniques, we conclude that they have significant inaccuracies. Running diagnosis on them leads to a wild goose chase, investigating components such as `fontd`, as it sends out messages after a long execution, which turn out to be completely unrelated to the root cause. We observe two general inaccuracies: *over-connections* and *under-connections*.

Over-connections usually occur when intra-thread execution segment boundaries are missing. We summarize three common programming patterns responsible for this—dispatch message batching, piggyback optimization, and superfluous wake-ups.

Dispatch message batching. Frameworks and daemons often implement event loops for handling multiple events inside callback functions. For example, Figure 4.1 shows two threads from the *fontd* daemon in macOS; the worker thread installs a callback function `dispatch_mig_server()` in a dispatch queue and the main thread dequeues and calls the function via `dispatch_client_callout`.

`dispatch_mig_server()` has an event loop which batch processes requests from different applications, presumably for performance. It invokes `call_back` to process a message and `set_reply` to post a reply. However, previous causal tracing tools like Panappticon assume the execution of a callback function is entirely on behalf of one request. `dispatch_mig_server` is thus treated as a single execution segment and edges are added between the vertex representing `dispatch_mig_server` and the many unrelated applications for which it handles requests. These edges incorrectly indicate causal relationships that would result in misleading diagnoses.

Piggyback optimization. Frameworks and daemons may piggyback multiple tasks in a system call to reduce kernel boundary crossings. For example, Figure 4.2 shows the macOS system daemon `WindowServer` uses a single system call `mach_msg_overwrite` to receive data and piggyback the reply for an unrelated event. However, previous causal tracing tools like Panappticon treat the execution of a system call as a single execution segment for one event, artificially making many events appear causally related.

Non-causal wake-up. Desktop applications typically have multiple threads synchronized via mutual exclusion, such that a thread's unlock operation wakes up another waiting thread. Such a wake-up may be, but is not always, intended as causality. For example, in Chromium, a wake-up is commonly followed by a batch processing block, but it is unclear whether the following events being batch processed depend on the wake-up event. Previous causal tracing tools assume any wake-up is causal, which may artificially make events appear causally related when they are not.

Under-connections usually occur due to missing intra-thread data dependencies and inter-thread shared flags.

Data dependency. Frameworks and daemons may have internal state that causally link different execution segments of a thread. For example, Figure 4.2 shows that a `WindowServer` thread calls the function `CGXPostReplyMessage` to save the reply message, which it internally stores in a variable `_gOutMsg`. When the thread later calls `CGXRunOneServicePass`, it sends out `_gOutMsg` if there is any pending message.

Shared data flags. Frameworks and daemons may use shared flags that causally link different

```
1 //a thread in WindowServer
2 while (true){
3   //postpone a reply
4   CGXPostReplyMessage(msg);
5   //receive requests
6   CGXRunOneServicePass();
7 }
```

```
8  CGXRunOneServicePass(){
9    if (_gOutMsgPending)
10     mach_msg_overwrite(
11      SEND|RECV,
12      _gOutMsg, RecvMsg)
13   else
14     mach_msg(RECV,RecvMsg)
15 }
```

Figure 4.2: Piggyback optimization and intra-thread data dependency. `mach_msg_overwrite` combines the reply of a previous event. Operations inside a thread have dependencies on `_gOutMsg`.

```
1 // worker thread needs
2 // UI update
3 obj->need_display = 1
```

```
1 //main thread
2 if (obj->need_display == 1)
3   render(obj)
```

Figure 4.3: Shared data flag across threads.

threads. Figure 4.3 shows a worker thread sets a field `need_display` inside a `CoreAnimation` object whenever the object needs to be repainted. The main thread iterates over all animation objects and reads this flag, rendering any such object. Existing tools do not track these kinds of shared-memory communication.

4.2 Overview of Argus

We have designed Argus to diagnose performance issues in desktop applications. Argus satisfies four key requirements not met by previous causal tracing tools: (1) use minimal instrumentation, (2) support closed-source components, (3) extract rich information from heterogeneous components with minimal manual effort, and (4) incur low runtime overhead.

Central to its design is the construction of *annotated trace graphs* from low-level trace events. Argus introduces the notion of *strong* and *weak* edges in trace graphs to mitigate inherent inaccuracies in tracing. When there is strong evidence of causality, such as an IPC message event, Argus adds a *strong edge* between vertices. When an execution segment is created by events that may not necessarily represent causality, such as non-causal wake-ups, Argus adds a *weak edge*. During

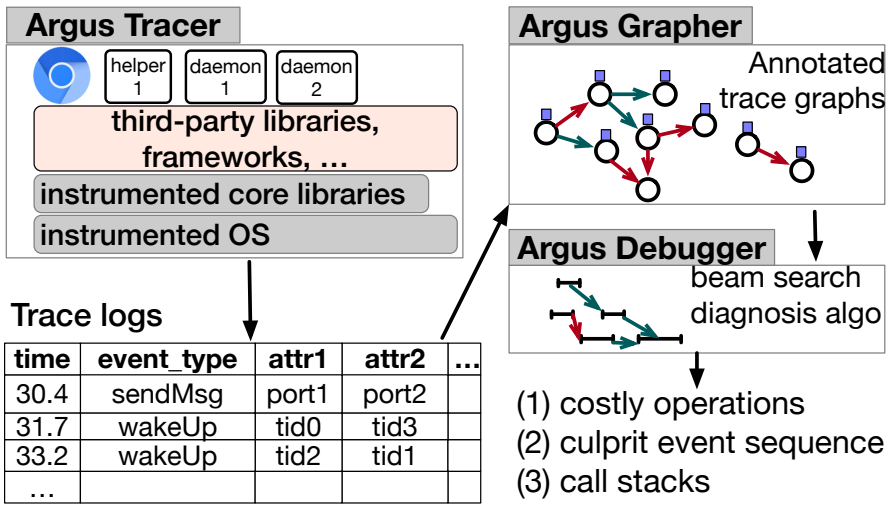


Figure 4.4: Overview of Argus.

diagnosis, Argus prefers traversing through strong edges when possible. Argus also stores extra semantic information in the graph vertices, including user input events, system calls, and sampled call stacks. This extra information is used to improve weak edge annotation and align and compare trace graphs for normal and abnormal execution to aid diagnosis.

Figure 4.4 shows an overview of Argus. It consists of three main components—a tracer, a grapher, and a debugger. The tracer runs continuously in the background on a user's machine, transparently logging events from low-level system libraries and the kernel, without any need to modify applications. When a user encounters some performance anomaly, she reports the issue about the problematic application, along with the timestamp of the anomaly occurrence. The reported issue and trace logs are sent to the developer, the logs containing events for both normal execution and abnormal execution when the performance anomaly occurs. The developer feeds the logs into the grapher to construct the annotated trace graphs for both normal and abnormal execution, and runs the debugger on the graphs to output the diagnosis results.

4.3 Argus Tracer

Argus traces events inside the kernel and low-level libraries, with minimal instrumentation. This provides three advantages over tracing in user applications. First, tracing in the kernel and libraries ensures coverage of custom programming paradigms. For instance, Argus traces general thread scheduling events and wake-up and wait to ensure coverage of a variety of custom synchronization primitives in desktop applications, because their implementations almost always use kernel wake-up and wait. Second, tracing in the kernel helps connect tracing events across process boundaries, because the addresses of the traced objects in kernel space are usually unique, while tracing in user programs requires maintaining and propagating unique identifiers. Third, tracing kernel threads helps bridge communications among processes. For instance, a kernel thread sends out a message to a process when the process needs to execute a delayed function.

In the macOS XNU kernel, Argus traces system calls, thread scheduling, interrupts, time-delayed calls, and Mach messages. Argus leverages existing macOS kernel tracing support [73], but adds enhancements to log more information and enable always-on tracing using a ring buffer to avoid exhausting storage. The enhancements require roughly 500 lines of code (LOC) in the XNU kernel, which are straightforward to add given that the kernel is open source. Trace events are asynchronously flushed to a file with a size limit. The limit is by default 2 GB, which can store roughly 20 million trace events; this is about 5 minutes of tracing when running large applications like Chrome. It can be easily adjusted to accommodate longer execution times. We used the default limit for all experiments in Section 4.6.

Argus logs kernel events to identify when threads are executing and their causal relationships. All system calls are traced to provide high-level semantics that can be used to identify causal relationships. Argus simply records return values for most system calls, but call stacks are also logged for a small set of system calls, namely those pertaining to Mach messages and synchronization using conditional variables and semaphores. Call stack information is later used by the Argus debugger to provide debugging information for developers. Thread scheduling is traced to track when

a thread becomes idle and which thread wakes it up. Argus logs three types of thread scheduling events: *wait* to indicate when a thread becomes idle, *wake-up* to indicate when the current thread wakes up another thread, and *preempt* to indicate when a thread is preempted due to its timeslice being used up or priority policies. Interrupts are logged to indicate when threads are preempted by interrupts, with call stacks also logged for interprocessor interrupts (IPIs). Argus traces the internal kernel implementation of time-delayed calls, which are used to implement asynchronous calls in libraries such as Grand Central Dispatch (GCD). Finally, Argus traces the internal kernel implementation of Mach messages, not just their invocation via system calls, to enabling tracing of all use of Mach messages, including use within the kernel among kernel threads.

To aid developers in interpreting the virtual addresses in call stacks via `lldb`, Argus also logs in userspace the virtual memory layout of images for all processes. The tracer records the virtual memory maps for all running processes when tracing is enabled or terminated; processes launched during tracing are also recorded. The memory layout information is also fed to the Argus debugger.

In addition to kernel tracing, Argus traces four closed-source macOS frameworks, `AppKit`, `libdispatch.dylib`, `CoreFoundation`, and `CoreGraphics`, to track UI events and batch processing paradigms used by applications. Because these frameworks are closed source, the trace events are added via binary instrumentation using a mechanism similar to Detour [33]. `AppKit` is used to dispatch UI events to handlers. Argus traces where a UI event is fetched from the `WindowServer` and dispatched to an event handler. `libdispatch.dylib` implements GCD, managing dispatch queues to balance work across the entire system. Argus adds trace events to track when objects are pushed into a dispatch queue and popped off of the dispatch queue and executed. `CoreFoundation` supports event loops for GUI applications, which are widely used to process requests from timers, customized observers, and sources such as sockets, ports, and files. Argus adds trace events so the handling of different requests inside event loops can be tracked separately.

To deal with the under-connection issues (Section 4.1), we annotate a handful of data flags in `CoreGraphics`. Given the shared flag variable names, Argus monitors the respective virtual addresses with watchpoint registers. Reads or writes to the addresses will invoke a signal handler that

records trace events with the values stored in those addresses. Argus adds code to `CoreFoundation` to install this signal handler.

Argus can use the same watchpoint mechanism to trace shared data flags in applications. To assist developers in finding these shared data flags, Argus provides a lightweight tool that uses `lldb` to record the operand values of each instruction and finds ones that lead to divergence in control flow, which are likely data flags. The shared flag variable names are recorded in an Argus tracer configuration file, which are then traced using the same signal handler installed by `CoreFoundation`. Since `CoreFoundation` is imported by all GUI applications, Argus can trace these shared data flag accesses without any application modifications.

Note that the annotation effort for shared data flags is in general small. This is because execution segments that access shared variables are usually connected already by some types of causality, e.g., wait/signal events; developers mainly need to provide Argus with shared flags that are accessed through ad-hoc synchronization [75]. In our experience, only a few shared flags need to be monitored. Also for this reason, although hardware watchpoint registers are limited, Argus is unlikely to exhaust them. In fact, none of the applications we evaluated in Section 4.6 needed shared flags to be identified or traced in the applications themselves. Mechanisms such as Kprobe [41] could potentially be used to extend Argus to support monitoring more shared flags.

4.4 Argus Grapher

Argus uses the trace logs to build an annotated trace graph by first identifying the boundaries of execution segments in each thread to determine the graph vertices, then adding annotated edges between vertices. The annotated edges contain type metadata to indicate *strong* versus *weak* edges, which is used during diagnosis to mitigate inaccuracies due to over-connections and under-connections, as discussed in Section 4.1.

Argus first determines the execution segments that will form the graph vertices. Using various trace events as boundaries, Argus splits the execution of each thread is into separate execution segments. First, Argus splits nesting of tasks executed from dispatch queues. If an execution

Edge	Rules for Edge Annotation
Strong	1. IPC message send and receive; 2. Asynchronous calls (work queue, delayed call); 3. Direct wake-up of a thread on purpose; 4. Data dependency.
Weak	1. Non-causal wake-up; 2. Execution segments divided between a wait event and a wake-up event, excluding following cases: wait or wakeup are introduced by system call `workq_kern_return`, or they are in `kern_task`; 3. Split suspicious batching execution segments, except known batching APIs: `RunLoopDoObservers`, `CGXServer`, `RunLoopDoSources1`, etc.
Boosted Weak	Continuous execution segments matching weak edge rules but are on behalf of the same task.

Table 4.1: Edge annotation rules.

of `dispatch_callout` invokes several other `dispatch_callout`, each dispatched task is separated. Second, Argus recognizes batch processing patterns such as `dispatch_mig_server()` in Figure 4.1 and splits the batch into separate execution segments. Third, when a wait operation blocks a thread execution, Argus splits the execution into separate segments at the entry of the blocking wait. The rationale is that blocking wait is typically done as the last step in event processing. Finally, Argus uses Mach messages to split execution when the set of communicating peers differs. Argus maintains a set of peers, including the direct sender or receiver of the message and the beneficiary of the message; macOS allows a process to send or receive messages on behalf of a third process. Argus splits execution when two consecutive messages have non-overlapping peer sets. By splitting thread execution using these four criteria, Argus avoids potential over-connections due to batching and piggyback optimizations.

Argus next determines the edges that should be added between vertices. Edges are introduced to reveal the causality of two execution segments and thus guide the causal path exploration. Based on the rules in Table 4.1, Argus annotates three types of edges: *strong*, *weak*, and *boosted weak*.

First, Argus adds strong edges by identifying Mach message, dispatch queue, time-delayed call, and data flag trace events associated with a vertex and finding the corresponding peer events

and peer vertices. For Mach message events, Argus adds a strong edge from the vertex with the message send event to the vertex with its associated receive event. If a message requires a reply, the received message can produce a reply message, which can be sent by a third thread, in which case Argus adds a strong edge from the vertex with the received message event to the one with the send event for the reply message. For dispatch queue events, Argus adds a strong edge from the vertex where the callback function is pushed to a dispatch queue to the vertex where the callback function is invoked. For time-delayed calls, Argus adds a strong edge from the vertex where the timer is armed to the vertex where the callback function is fired. For shared data flags, Argus adds a strong edge from the vertex with a data flag write event to the vertex with its corresponding read event, avoiding potential under-connections.

Second, Argus adds edges by identifying thread scheduling trace events and finding the events and vertices corresponding to the pair operations. Argus adds strong edges only when the context clearly indicates causality, such as the signal and wait operations of a condition variable. Otherwise, Argus adds only weak edges. One hint Argus takes from macOS is that, if a wake-up is not followed by a specific communication operation (*e.g.*, message receive), and does not target a specific thread but all threads on the wait queue, then it is likely not causal, in which case a weak edge is added.

Third, because Argus splits the execution of a thread into segments (graph vertices) based on heuristics that may not always be valid, Argus adds weak edges between these adjacent execution segments, as shown in Figure 4.5. Argus converts a weak edge into a *boosted weak edge* if two continuous execution segments are on behalf of the same task. It infers whether the segments are for the same task by leveraging call stack symbols. We calculate frequencies for all symbols across the whole tracing and notice a low-frequency (bottom 10%) symbol usually only appears in a task from a specific application, compared to high-frequency symbols from system routines or framework APIs. Thus, if the two segments share the same low-frequency symbols, Argus infers they are collaborating on the same task and sets a boost flag for the weak edge between them.

However, abuse of weak edges could generate excessive false positives during diagnosis, so

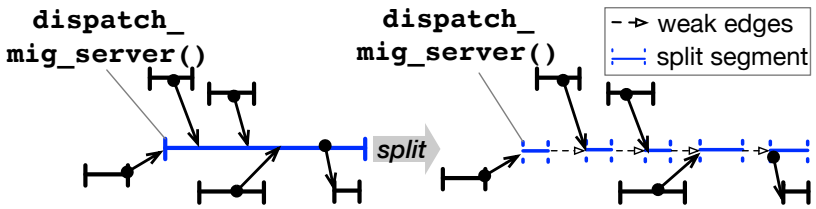


Figure 4.5: The segment for batch processing in `dispatch_mig_server` is split into multiple segments to distinguish different items. Weak edges are added among the split segments.

Argus takes advantage of high-level semantics to avoid adding unnecessary weak edges between adjacent execution segments. First, if the call stacks of two segments of a thread share no common symbols or share a recognized system library batching API, Argus does not add a weak edge between them. Second, because wait and wake-up events are mostly from system calls, Argus leverages system call semantics to determine the necessity of weak edges. For example, we find the wait event from system call `workq_kern_return` indicates an end of a task in the thread, while the wake-up event formed in `workq_kern_return` intends to acquire more worker threads for concurrent tasks in the dispatch queue. Execution segments containing such event sequences do not need bridging with weak edges. Finally, the kernel task in macOS acts as a delegate to provide service for many applications, such as I/O processing and timed delayed invocations. The kernel task threads contain execution segments beginning with a wake-up event and ending with a wait event. Each segment serves different requests and they are not causally related, so weak edges are not added between those kernel task execution segments.

4.5 Argus Debugger

Argus uses the constructed trace graphs to diagnose performance issues by starting with the vertex that contains the performance anomaly and traversing the graphs to identify the causal paths including the root cause vertices. The typical critical path analysis used in existing causal tracing solutions cannot effectively handle the noises in the trace graphs. Argus introduces a new diagnosis algorithm based on *beam search* to efficiently explore the causal paths likely related to the

performance anomalies. It also introduces a novel subgraph comparison mechanism to find missing vertices not present in the trace graph for abnormal execution that are present in the graph for normal execution. This comparison is helpful to identify the root cause that would be otherwise unknown.

4.5.1 Causal Path Search—Beam Search

From a given vertex that contains the anomaly, such as the spinning cursor, Argus finds what path "caused" the anomaly by using beam search based on a cost function for annotated edges. Beam search is similar to breadth-first-search, but at each search step, it sorts the next level of graph vertices based on a cost function and only stores β—the beam width—best vertices to consider next. Argus customizes its beam search with a *lookback* scheme such that the algorithm evaluates the cost function for multiple levels of edges before pruning. Argus evaluates the vertices and prunes them with β only after the search advances the configured *lookback* steps to avoiding pruning paths with weak edges too early.

Argus's beam search algorithm provides two key advantages. First, compared to brute-force search, beam search only explores the most promising vertices, which is essential given that trace graphs are highly complex with millions of edges; searching all paths would be too inefficient and, given graph inaccuracies, result in an overwhelming number of options to consider. Second compared to local search methods such as hill-climbing, beam search embraces more possible causal paths because it ranks partial solutions and the ranking changes during the exploration. For example, assuming strong edges are preferred to weak ones, a path with a weak edge followed by a series of strong edges is likely to get a higher ranking and be returned by beam search, but will be missed by a hill-climbing search algorithm.

Figure 4.6 illustrates the algorithm. It searches for causal paths *backwards* from the anomaly vertex. For each incoming edge of the current vertex, the algorithm computes the *penalty score* for the new path. At every lookback step, the search branches are pruned: it sorts the paths by their penalty scores and only retains at most β paths with low penalties. A path is added to the result if

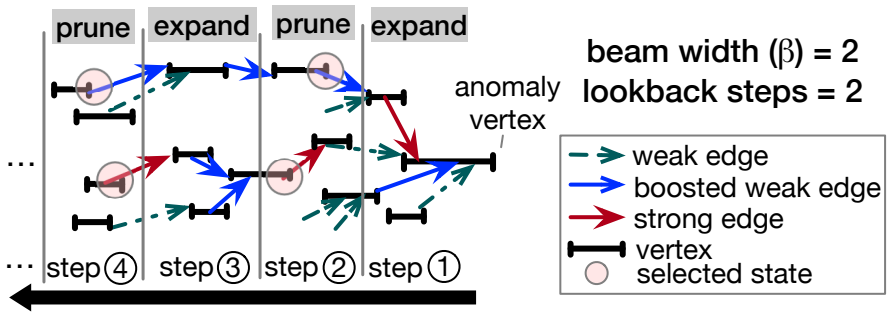


Figure 4.6: Beam search diagnosis algorithm. Search backwards from the anomaly vertex; choose the best β states to expand next. For every lookback steps, prune the existing states to at most β paths.

a vertex is reached containing a UI event or has no incoming edges, and the beam width decreases by one. Using such vertices as for path termination helps developers understand causality in an end-to-end request handling transaction.

Algorithm 1 lists the pseudo-code of the search algorithm. Lines 16 – 18 compute penalty scores for new paths after incoming edges are added to the path. Lines 22 – 25 prune the searched branches every L lookback steps. Paths are sorted by their penalty scores and paths with high penalties are discarded. Penalty scores are calculated with a linear function on edge values, where a strong edge is -1, a weak edge is 1, and a boosted weak edge is 0. A path with n edges has a penalty $p = \sum_{i=1}^{n}(a \times E_i + b)$, where E_i is the ith edge value. This approach guides search towards paths with stronger causality. While more complex non-linear functions may be feasible, this simple function works well for many diagnosis cases.

The beam width setting affects the search efficiency and diagnosis accuracy. A setting too large would cause path explosion and noisy paths to be returned. A setting too small may easily miss the true causal path. We set $\beta = 5$ to strike a good balance. Tuning this parameter is relatively easy in practice. The lookback step setting is set based on observing that traversal of most graphs encounters a weak edge within five steps. We set $L = 5$ to tolerate weak edges. Given this setting, a path of x strong edges, y weak edges, and z boosted weak edges has a penalty of $p = -a \times (x - y) + 5 \times b$. If all edges are strong, the penalty is negative only when $b < a$. If there are weak edges, the penalty is positive only when $(x - y) \times a < 5 \times b, where -3 < x - y < 3$.

Algorithm 1: Causal Path Search Algorithm (Beam Search).

Data: **g** - event graphs, **curVertex** - vertex inspected in current search state, **beamWidth** - search branches at most, **lookbackSteps** - searching steps taken before pruning current search branches

Result: paths

```
1  Function BeamSearch(g, curVertex, beamWidth, lookbackSteps):
2      curStates.init(curVertex);
3      curSteps ← 0;
4      while curStates.incoming_edges() > 0 && beamWidth > 0 do
5          ++curSteps;
6          newStates.clear();
7          for each state ∈ curStates do
8              if beamWidth <= 0 then
9                  break;
10             end
11             if state.path.reach(UI) || state.path.incoming_edges = ∅ then
12                 paths.add(state.path);
13                 −−beamWidth;
14             end
15             for each edge ∈ state.path.incoming_edges do
16                 newState.path ← state.path + edge;
17                 newState.score ← state.score + penalty(edge.val);
18                 newStates.add(newState);
19             end
20         end
21         curStates ← newStates;
22         if curSteps = lookbackSteps then
23             pruneStates(curStates, beamWidth);
24             curSteps ← 0;
25         end
26     end
27     pruneStates(curStates, beamWidth);
28     paths.append(curStates.paths);
29     return SortIncPenaltyScore(paths);
30 Function pruneStates(newStates, beamWidth):
31     SortIncPenaltyScore(newStates.paths);
32     while newStates.size() > beamWidth do
33         newStates.pop_back();
34     end
35     return;
```

Therefore, we set the default penalty function coefficients $a = 3$ and $b = 2$.

Algorithm 2: Subgraph Comparison Algorithm.

```
Data: anomVertex – problematic vertex, anomGraph – trace graph for anomaly case,
      normGraph – trace graph for normal case
Result: ret- potential culprits of anomaly
Function SubGraphCompare(anomVertex, anomGraph, normGraph):
    ret.clear();
    similarVertices ← FindSimilarVertices(normGraph, anomVertex);
    baselineVertex ← GetBaseLine(similarVertices, anomVertex);
    targetVertex ← woken(normGraph, baselineVertex);
    causalPaths ← BeamSearch(normGraph, targetVertex, beamWidth, lookbackStep);
    // sub-graph is constituted with paths;
    for each causalPath ∈ causalPaths do
        for each vertex ∈ causalPath do
            expectVertex ← SimilarVertex(anomGraph, vertex);
            if expectVertex = ∅ then
                // missing similarity to vertex ;
                anomThr← SearchThread(anomGraph, vertex.thread);
                // get the vertex that causes the dissimilar ;
                suspVertex← VertexInThread(anomGraph, anomThr);
            else if DifferentVertices (expectVertex, vertex) then
                // vertex acts different from normal case ;
                suspVertex← expectVertex;
            else
                countinue;
            end
            ret.push_back(suspVertex);
        end
        if !ret.empty() then
            return ret;
        end
    end
    return ret;
```

4.5.2 Subgraph Comparison

If we run causality analysis only on the trace graph constructed with the anomalous performance issue, the root cause may not be exposed in some cases. For example, a blocked function

could be caused by a missing wake-up from one of the background threads. If the thread does not perform the wake-up during abnormal execution, there will be no execution segment with the wake-up, and therefore no vertex in the anomalous trace graph that can be identified correctly as the root cause. Argus addresses this problem by first constructing the trace graphs for both normal and abnormal execution. It then uses its beam search method on the normal trace graph to identify the causal paths in that graph that corresponds to the desired normal behavior that does not occur during abnormal execution. We refer to those causal paths a *subgraph*.Argus then uses the vertices in the subgraph to identify the missing root cause in the abnormal execution. This is done by introducing a novel subgraph comparison method between the trace graphs for both normal and abnormal execution, which is listed in Algorithm 2.

Argus first determines a baseline vertex in the normal graph that is comparable to the anomaly vertex in the anomalous graph. Argus computes a signature for each vertex based on the trace event sequence in its execution segment. The signature is composed of two parts, one that encodes the types corresponding to the event sequence e.g. 0 for IPC event, 1 for syscall event, etc., and another that is a hash of the event parameters, e.g., process names of IPC events. Argus calculates the similarity of two vertex signatures using string edit distance. Among the vertices in the normal graph that are similar to the anomaly vertex, Argus chooses one that behaves differently from the anomaly vertex, based on return values of system calls and execution times. For example, a vertex whose last event is a blocking system call with a timed wait may behave in two different ways, timing out or quickly woken up.

After Argus identifies a baseline vertex, it obtains its causal paths using Algorithm 1. The result is a subgraph of the normal trace graph rooted from the baseline vertex to some ending vertex. Argus examines the subgraph from the most related causal path. Starting with the ending vertex V, whose execution segment was executed by some thread T, Argus identifies vertices in the abnormal trace graph that were also executed by T. For each identified vertex, Argus checks whether it behaves differently from V, in which case it is flagged as a suspicious vertex. If no such vertices are found, Argus repeats this procedure with the next vertex in the subgraph. Otherwise, for each

suspicious vertex that has incoming edges, Argus recursively repeats the subgraph comparison by treating the suspicious vertex as the initial anomaly vertex. The recursive procedure effectively keeps working backwards through vertices to eventually find a set of root cause candidate vertices in the anomalous trace graph with no incoming edges. Argus then returns the vertex whose path to the original anomalous vertex has the lowest penalty score, identifying that vertex as the root cause.

Figure 4.7 shows a simplified example of the subgraph comparison method applied to the Chromium performance issue discussed in Section 4.1. Vertex E' in the anomalous graph is the initial anomaly vertex. Argus identifies vertex E in the normal graph as having a similar signature but behaving differently, and treats it as a baseline vertex. Argus applies beam search to the normal graph starting with vertex E, resulting in the subgraph $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E$. Argus starts with A, identifies its browser thread, and determines that A cannot be the root cause since the same browser thread contains the performance anomaly E' in the anomalous trace graph. Argus then considers B, identifies its renderer thread, and finds all vertices in the anomalous trace graph executed by the renderer thread. F' is similar to F, so it is not considered a suspicious vertex, but J' is not similar to any vertex in the normal trace graph, so it is considered suspicious. J' has no incoming edges and is identified as a root cause candidate. If there are no other candidates identified, J' is returned as the root cause.

4.5.3 Debug Information

Argus further provides the calling contexts of the anomaly vertex and the root cause vertex to help developers localize the bug in code. To do so, Argus examines the call stacks it attaches in the graph vertices. If the anomaly or root cause vertex has a blocking call, the call stack Argus tracer collects would reveal the context of the blocking call directly. If the vertex has a long runtime cost, the problematic vertex usually contains periodic IPIs, where the Argus tracer collects call stacks. In this case, the Argus debugger calculates the longest common sequence of frames from those call stacks. The top frame in the sequence reflects the costly function call.

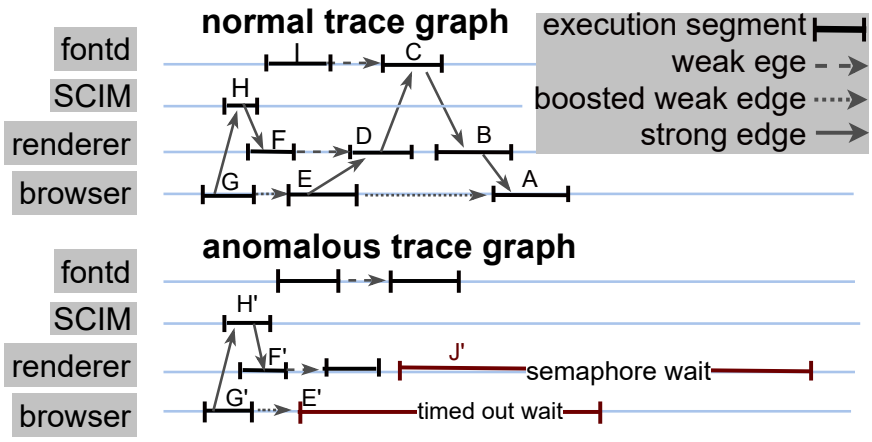


Figure 4.7: Chromium normal and anomalous trace graphs after user typed in a search box (vertex G/G'). Vertex E' (requesting a bounding box for input) is the anomaly vertex. Sub-graph in normal trace graph is extracted from baseline vertex E. Vertex J'(javascript processing blocks on semaphore) is the root cause Argus reported. Trace graphs are simplified for clarity; only processes are shown and communications with processes such as `imklaunchagent` are omitted.

For instance, in Figure 4.7, Argus reports the following information: (i) the calling context of problematic vertex E' and its causal path $E' \leftarrow G'$; (ii) the calling context of root cause vertex *J'* along with its *unmatched causal path* in baseline trace graph: $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E \leftarrow G$, and vertex B is marked because its thread should have waken up the blocking thread in the anomaly case.

4.5.4 Diagnosis for Spinning Pinwheel in macOS

Argus's debugger can be used to effectively diagnose spinning pinwheel performance issues in macOS applications. Recall that a spinning pinwheel appears when the UI thread of an application can not process any user inputs for over two seconds. During normal execution, the two-second interval may cover many vertices, but when the spinning pinwheel appears, the main thread of the application is stalled and the two-second interval covers only a single vertex. Leveraging this timing information, Argus identifies the anomaly vertex in the main thread of the targeted application and classifies the issue as either a *LongRunning* and *LongWait* anomaly.

LongRunning. The main thread is busy performing lengthy CPU operations and therefore its

execution segment is in the anomalous trace graph. Argus uses its beam search method to identify the causal path between the anomaly vertex and the vertex with the UI event resulting in the issue. Argus reports the costly API, event handler, and causal path to the developer.

LongWait. A UI thread is blocked, but it is hard to tell why. Argus uses its subgraph comparison method together with its beam search method to deduce which vertex is missing from the anomalous trace graph. A long-wait event could be caused by another long-wait event. Argus supports recursively diagnosing "the culprit of the culprit." Therefore, it can reveal deep root causes. At the end of each iteration of diagnosis, the calling context of problematic vertex, root cause vertices in the anomalous trace graph, and causal paths are ranked and reported to users.

Some LongRunning issues may be diagnosed with existing tools such as `spindump` if the profiling is accurate and complete. However, Argus is better in that a call stack is usually not enough to connect the busy processing to the event handler, due to the prevalence of asynchronous calls. Also, call stack profiles after the anomaly may miss the real costly operations. LongWait issues usually involve multiple components and are extremely hard to understand and fix with current tools. Those issues may remain unresolved for years and significantly hurt user experience and developer productivity.

4.6 Evaluation

We have implemented Argus across multiple versions of macOS, ranging from El Capitan to Catalina. We evaluate Argus to answer several key research questions: (1) Can Argus effectively diagnose real-world performance anomalies for modern desktop applications? (2) How does Argus compare to other performance debugging tools? (3) How useful are Argus's weak edges and their optimizations in mitigating tracing inaccuracies? (4) How much overhead does Argus's tracing tool incur? Unless otherwise indicated, all applications and tools were run on a MacBookPro12,1 with an Intel Core i7 CPU, 16 GB RAM, and an APPLE SM0512G SSD.

4.6.1 Diagnosis Effectiveness

We evaluated Argus on 12 real-world user-reported performance issues in 11 popular desktop applications, which we collected and reproduced, as listed in Table 4.2. We are especially interested in evaluating performance issues that have been hard to troubleshoot. Except for B11, all of these are open issues, meaning their root causes were previously unknown to developers. For B2, the reported issue was "fixed" in the latest version (due to refactoring or platform upgrade) but the root cause remained unknown. Nine applications, or some of their components, have source code available, whereas two applications are closed-source. Source code was used to validate whether the correct root cause was diagnosed for the performance issues, but all evaluation was performed on the released application binaries. We have also used Argus with proprietary applications like Microsoft Word for macOS, but without source code, we need to wait for vendors' confirmation and responses; in our experience, vendors are reluctant to communicate issues with an external party.

Table 4.3 shows that Argus was able to diagnose all 12 performance issues, including all long-standing open issues. As listed in Table 4.4, we checked the correctness of Argus's diagnosed root causes in three ways: (1) inspecting the corresponding source code if available, (2) dynamic patching with `lldb` based on the diagnosed root cause to fix the problem, and (3) confirmation by developers. The last one is ideal, but not always feasible; we reported our findings to developers for seven issues, but only received two responses. Only the root cause of B11 was previously known, which Argus returned correctly (Grd). For B1, B7, and B10, we validated the diagnosed root causes by analyzing the source code (Src). For B2 and B4, we received confirmation from the respective application developers that Argus correctly diagnosed the root cause for these open issues [70, 74] (Dev). For example, for B4, the Sequel Pro developers suspected a particular Cocoa Framework API does not work as expected, but could not pinpoint the exact place to fix it. Argus determined the defect was in their installed callback function, and we submitted a pull request [74] to fix the issue. B8 was fixed in an official developer patch after we reported the root cause (Fix). For the remaining issues, we confirmed the issue was resolved by dynamically patching the appli-

ID	App	Performance Issue	Age
B1	Chromium	Typing non-English in searchbox, page freezes.	7 yr
B2	TeXstudio	Modifying Bib file in other app gets pinwheel.	2 yr
B3	BiglyBT	Launching BiglyBT installer gets pinwheel.	1 yr
B4	Sequel Pro	Reconnection via ssh causes freeze.	4 yr
B5	Quiver	Pasting a section from webpage as a list freezes.	5 yr
B6	Firefox	Connection to printer takes a long time.	1 mo
B7	Firefox	Some website triggers pinwheel in the DevTool.	3 yr
B8	Alacrity	Unresponsive after a long line rendering.	6 mo
B9	Inkscape	Zoom in/out shapes causes intermittent freeze.	1 yr
B10	VLC	Quick quit after playlist click causes freeze.	7 mo
B11	QEMU	Unable to launch on macOS Catalina.	1 mo
B12	Octave	Script editing in GUI gets pinwheel.	2 yr

Table 4.2: Real-world performance issues in macOS applications.

cation based on the root cause (Dyn). We describe the typical performance issues in further detail to show how Argus diagnose them effectively.

B1-Chromium: This is the Chromium performance issue discussed in Section 4.1. Argus analyzes the trace graph, pinpoints the circular waits between renderer main thread and browser main thread with the interactions of daemon processes like `fontd`. Argus not only localizes the problematic execution segment (waiting on a condition variable), but also the sequence of events leading to this issue. The same issue occurs in Chrome. We also reported our findings to Chrome developers, but received no reply.

B2-TeXstudio: TeXstudio [78] is an IDE for creating LaTeX documents. Users reported when they modified a bibliography file with another application, TeXstudio froze with a spinning pinwheel. We reproduced this case by running `touch` from a terminal on a 500 entry bibliography file, which immediately caused a spinning pinwheel to appear in *TeXstudio*'s window. Argus analyzes the trace graph and identifies five causal paths, ordered by likelihood of causality. The first path is from Terminal to TeXstudio: Terminal→`WindowServer`→`bash`→`kernel_task` →`fseventd`→TeXstudio. It connects multiple entities and suggests the following root cause chain. The `touch` command triggers a change in the file metadata. `fseventd` notifies TeXstudio and in-

ID	Root Cause Identified
B1	circular wait between renderer and browser main threads.
B2	long running function calculating line indices in document.
B3	recursive invocations of accessible objects in GUI.
B4	UI event loop mishandling input causes deadlock with ssh.
B5	paragraph value never equals last paragraph inside web view.
B6	sleep waiting on chain of deamons, the last being nsurlsessiond.
B7	excessive garbage collection on the main thread.
B8	excessive copy of rendering cells when searching potential URL.
B9	excessive memory operations for trimming and compositing.
B10	termination signal before displaying thread ready; deadlocks.
B11	window adjustment before it finishes launching; deadlocks.
B12	readline thread writing tty repeatedly, main thread waiting.

Table 4.3: Root causes identified by Argus.

vokes a callback handler. TeXstudio executes `QDocument::startChunkLoading`, and causes busy processing in TeXstudio's main thread. Argus also outputs the call stack with the busy APIs, `startChunkLoading` and `QDocumentPrivate::indexOf()`. We reported our findings to the developers and received confirmation that the diagnosis is correct.

B5-Quiver: *Quiver* [59] is a closed-source notebook application for mixing text, code, Markdown, LaTeX, etc. Users report that applying bullet points to a text cell without an empty line at bottom causes a spinning pinwheel [58]. Based on the Argus trace graph, there is a hanging vertex in the WebKit component used by Quiver. In particular, WebKit hangs in executing `InsertListCommand::doApply` when applying the list command to the Webview context from Quiver. The hang occurs because of an infinite loop bug in WebKit rather than Quiver. We verified the root cause by changing the comparison result of the loop with `lldb`, which enables Quiver to display the bulletin points without a spinning pinwheel. We reported our findings to the developers, but received no reply.

4.6.2 Comparing with Other Approaches

We compared Argus versus other state-of-the-art tools for diagnosing the performance issues in Table 4.2. We used two widely-used traditional debugging and profiling tools from Apple,

`spindump` [1] and `Instruments` [34]. For `spindump`, we enable it once the performance issue appears, and repeat the process five times to eliminate bias on the start timing. `spindump` separately ranks the symbols from all sampled call stacks and only the top of call stacks. We examined the top$N$ symbols and their corresponding call stack information. For `Instruments`, we enable its time profiler in the background when reproducing the bugs, and analyze its data from two seconds before the performance issue occurs to three seconds after. We rank APIs in the reported call trees with CPU time percentage and filter out system routines. Then, we select the top$N$ APIs for investigation. We used values from $N = 1$ to $N = 10$. We also used two causal tracing tools, the macOS version of Panappticon, as discussed in Section 4.1, and AppInsight [60]. Since AppInsight was originally built for Windows, we reimplemented a version for macOS which captures trace events, constructs trace graphs, and follows the path analysing rules for diagnosis according to AppInsight's design.

Table 4.4 shows the results for using the different tools, including the results for Argus discussed in Section 4.6.1; checks indicate correct root cause diagnosis. All of the other tools diagnosed much fewer performance issues than Argus. `spindump` diagnosed at most five issues. It captures the state near the symptom point but cannot deduce how the execution reaches a problematic point, especially in the presence of highly concurrent and asynchronous execution across different entities. `Instruments` diagnosed at most four issues. It only outputs the most costly functions, which are helpful for performance optimizations but may not be for troubleshooting specific performance issues. Neither of the causal tracing tools did any better because the constructed trace graphs are highly inaccurate. AppInsight only diagnosed two issues while Panappticon diagnosed four issues.

4.6.3 Mitigation of Trace Graph Inaccuracies

We evaluated the effectiveness of Argus in mitigating trace graph inaccuracies in diagnosing the performance issues in Table 4.2. Table 4.4 shows the benefits of weak edges and Beam search. Argus diagnoses eight issues if it discards weak edges (no weak edges), and seven issues if it uses

	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12
`spind.@top1`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
`spind.@top3`	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗
`spind.@top5`	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✗
`spind.@top10`	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✓	✗
`Instr.@top1`	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
`Instr.@top3`	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
`Instr.@top5`	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
`Instr.@top10`	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗
AppInsight	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
Panappticon	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗
Argus	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
no weak edges	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
w/critical path	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗	✗	✗
Argus result validation	Src	**Dev**	Dyn	**Dev**	Dyn	Dyn	Src	Fix	Dyn	Src	Grd	Dyn

Table 4.4: Comparing Argus with other debugging tools.

	Events	Vertices	Edges		
			Total	Strong	Weak
Max	12.3M	1.68M	1.62M	751.3K	864.6K
Min	260.8K	15.1K	25.5K	17.5K	8.01K
Mean	3.31M	349.5K	358.4K	188.8K	169.6K
Med	1.02M	97.3K	172.6K	111.9K	60.71K

Table 4.5: Argus trace graph statistics.

traditional critical path analysis instead of Beam search (w/critical path). In both cases, Argus still performs better than other tools.

Table 4.5 shows that the Argus trace graphs include hundreds of thousands to millions of events, and on average have 350K vertices and up to 1.68M vertices. Graphs are in general dense, with an average of 358K edges. A significant percentage, 40% on average, of the edges are tagged as weak edges. To avoid abusing weak edges and overwhelming the diagnosis, Argus applies the optimizations discussed in Section 4.4. Figure 4.10 shows the percentages of potential weak edges that Argus excludes from the trace graph for different techniques: call stack similarity, wait on end

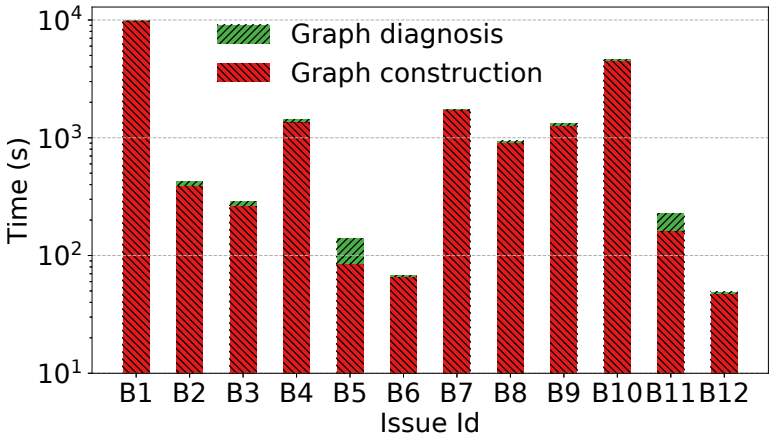


Figure 4.8: Argus diagnosis time.

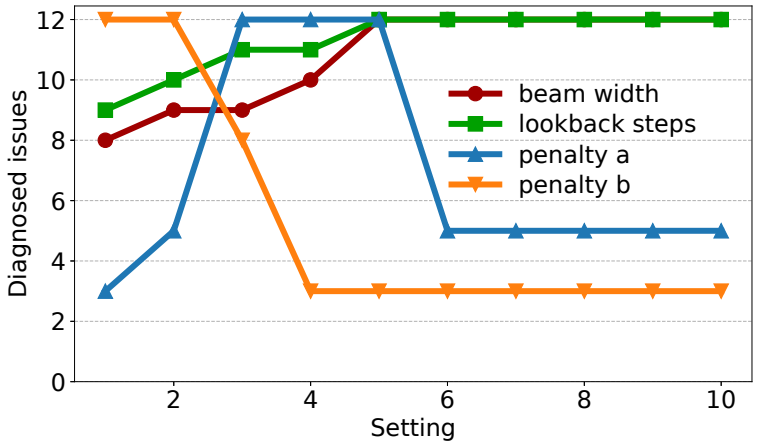


Figure 4.9: Sensitivity of beam search settings.

of task in a thread, acquire worker threads, and kernel task delegate. Call stack similarity was most effective in pruning potential weak edges.

We evaluated the sensitivity of Argus's beam search settings: beam width, lookback steps, and penalty function coefficients a and b. Figure 4.9 shows the number of diagnosed issues when changing one setting and leaving the rest at their defaults. The settings for beam width and lookback steps are robust. Larger settings increase the diagnosis effectiveness, but if they are too large, the Argus debugger could run out of memory or time out for large trace graphs. Changing penalty function coefficients can significantly change the number of diagnosed issues. In general, small coefficients from two to four are better. Overall, the results indicate that Argus is practical, and

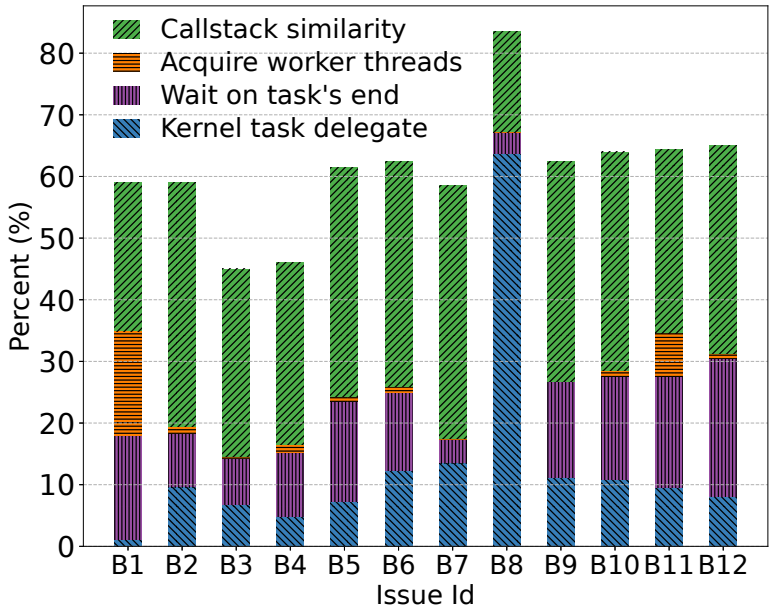


Figure 4.10: Potential weak edges pruned.

developers do not need to spend much effort to tune search settings.

4.6.4 Performance

We measured the time to run the Argus grapher and debugger for diagnosing each of the performance issues in Table 4.2. Figure 4.8 shows the time varies for different issues, ranging from 49 s (B12) to 9870 s (B1). Constructing the trace graph is the dominant cost. Running the beam search diagnosis algorithm on the graph is fast, taking at most 144 s (B10).

We also measured the overhead of the Argus tracer using various CPU, memory, and I/O benchmarks running on a live deployment of Argus on a MacBookPro9,2 with an Intel Core i5-3210M CPU, 10 GB RAM, and a 1 TB SSD. We first measured five runs of the iBench Cocoa benchmark [46], with and without Argus, to measure overall performance. The reported scores were 6.14 with 0.027 standard error without Argus tracing and 6.13 with 0.025 standard error with Argus tracing enabled. Argus only has a 0.16% performance degradation on average. In comparison, with `Instruments`, the reported score was 6.04, showing a 1.6% performance degradation. We next ran the Chromium Catapult benchmarks [14] to evaluate CPU performance, with and without

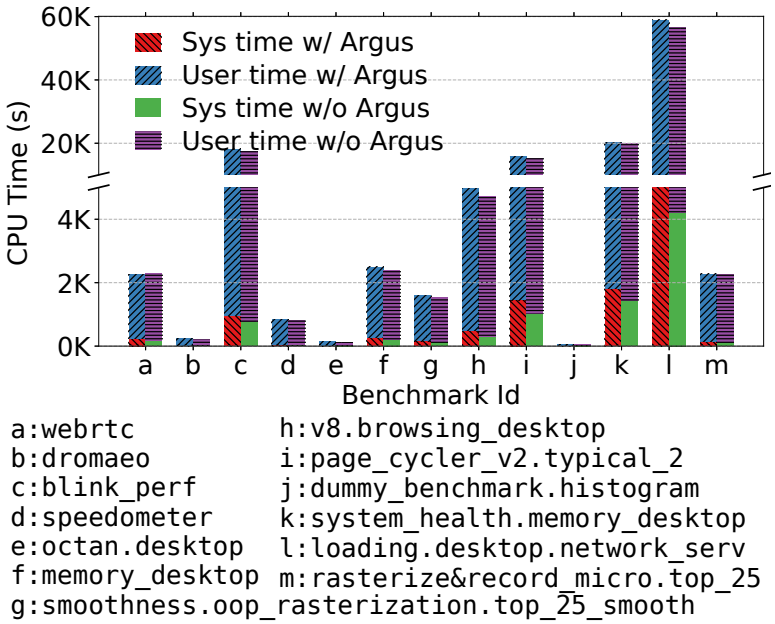


Figure 4.11: CPU overhead.

Argus tracing. Figure 4.11 shows that Argus overhead is less than 5%. The average overhead for `real` and `user` time was 3.36% and 2.15%, respectively. `sys` overhead was higher because Argus tracing in libraries involves crossing the user-kernel boundary. Finally, we ran Bonnie++ [20] and IOzone [13] I/O benchmarks to evaluate I/O performance, with and without Argus tracing. Figure 4.12 shows the I/O throughput measurements. Argus tracing has almost no overhead for sequential character read and write operations and less than 10% overhead for block read and write operations.

4.7 Conclusions

Argus is the first comprehensive causal tracing system to diagnose performance anomalies in complex desktop applications. We observe that although causal tracing is powerful and extensively studied in distributed systems, it is brittle when applied to desktop systems due to inherent tracing inaccuracies.

Argus addresses this problem by introducing annotated trace graphs with strong and weak edges to account for these inaccuracies. Argus pairs annotated trace graphs with a novel beam

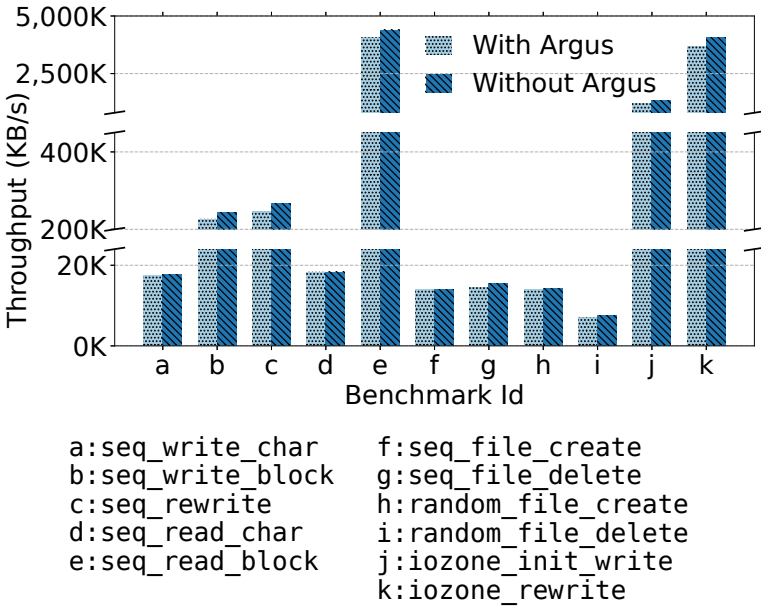


Figure 4.12: I/O overhead.

search diagnosis algorithm and subgraph comparison mechanism to determine causal paths in the presence of these inaccuracies. We have implemented Argus across multiple versions of macOS and evaluated its effectiveness on complex desktop applications. Argus successfully pinpoints the root causes for 12 real-world performance issues in these applications, many of which had remained open for several years. Argus imposes less than 5% CPU overhead, making it fast enough for regular use.

We believe Argus's strong and weak edge notions and inaccuracy-tolerant diagnosis algorithm may extend beyond the scope of desktop systems. In causal tracing of distributed systems, many solutions assume systems are perfectly instrumented, but in practice this is not the case. We plan to explore using Argus's techniques in the context of distributed systems as an area of future work.

Chapter 5: Limitations and Future Work

This chapter discusses the limitations of the performance diagnostic tools, namely Argus and vProf, which are implemented in this dissertation. In addition to examining their limitations, we also explore the future work aimed to overcome these challenges.

Reliable Bug Reproduction One significant limitation of these diagnostic tools is that they require performance issues to be reliably reproducible. However, reproducing certain performance bugs in real-world scenarios can be extremely challenging. A study conducted by Han *et al.* [30] further emphasizes the difficulties associated with the reproduction of performance bugs. The researchers randomly examined 98 performance bugs, all of which had been confirmed by developers with domain knowledge. Surprisingly, out of these bugs, only 17 could be consistently reproduced.

Hence, it becomes imperative to explore strategies aimed at facilitating easier and more reliable bug reproduction. Based on our experience, the continuous collection of supplementary runtime information can be a viable solution to help developers identify crucial operations for bug reproduction. For instance, user input event handlers can manifest the necessary user operations, such as a sequence of key presses, to reproduce the performance issue. Recording data flow, on the other hand, can aid developers in discovering the configuration settings. This lightweight information has the potential to significantly enhance the ease of reproducing performance issues.

Baseline Selection The diagnostic algorithms in both Argus and vProf require a comparison between a buggy execution and a normal baseline. The effectiveness of these tools depends on the choice of appropriate baselines. For example, if a performance bug consistently exists in modern desktop applications, Argus works effectively only when the bug is a LongRunning performance issue. In such cases, Argus extracts the most likely causal paths that lead to high CPU utilization. However, in LongWait cases involving extended waiting periods, the effectiveness of Argus di-

minishes because it cannot identify the missing events that should have awakened the long waiting thread in the absence of a baseline. Similarly, when there is no comparable use case available, vProf operates similarly to the traditional profiler `gprof`.

Fortunately, runtime information can provide valuable insights into the potential variations in the use cases that revolve around the performance anomaly. Focusing on leveraging these variations to automatically generate the appropriate baseline deserves our attention, as it can ensure the usefulness of the diagnostic tools.

Refine Runtime Information The diagnostic tools inevitably involve noisy data, as discussed in Chapter 2, and the runtime information collected to enhance these tools can be incomplete. This is because the availability of certain information is not guaranteed due to various optimization techniques in both source code and compilers. For example, the call stack collected in Argus depends on the existence of debugging symbols in the binary. Similarly, the locations where variables can be accessed at runtime are also determined by the debugging information entries in the binary. Furthermore, collecting all related data is not feasible due to the resulting overhead. For instance, vProf does not support dereferencing complex class pointers, as sanitizing these pointers incurs a considerable cost.

Noise and the incompleteness of runtime information can impact the diagnosis, potentially leading to both false positives and false negatives. Take Argus as an example. If Argus categorizes a weak connection as a strong edge due to incorrectly inferred semantics from call stacks, it may generate false positives. Conversely, missing a strong edge may result in false negatives. Although the beam search algorithm helps tolerate potential errors, it cannot completely eliminate the possibility of errors. Similarly, in vProf, different baselines can produce different value samples, leading to different inferences of bug patterns. While multiple bug patterns may all be reasonable, they provide developers with varying levels of insight into performance bugs.

Hence, it is worthwhile to explore approaches for mitigating erroneous data and determining whether the collected data is sufficient for diagnosing performance issues. For example, employing different baselines can offer developers a comprehensive understanding of performance issues from

various aspects. Aggregating the findings can help pinpoint root causes.

Implementations for Production Environment The prototypes of vProf and Argus, as implemented in this dissertation, demonstrate good performance in addressing the selected performance issues. However, their current implementations may be overly simplified to provide comprehensive support for production systems, particularly in terms of reliability, usability, and scalability.

To address these limitations, several areas of improvement need to be considered. Firstly, enhancing implementation details is essential to accommodate complex software. Currently, the vProf prototype restricts value sampling to primitive types, structure members, and pointers. To enhance comprehensiveness, it could be extended to reliably and efficiently dereference pointers from intricate data structures. Moreover, the static analysis in vProf remains incomplete due to the absence of analysis for function pointers. Integrating dynamic analysis can further improve the reliability of vProf in performance diagnosis.

Secondly, minimizing analysis overhead is crucial to improve their usability and scalability for production systems. While the overhead on the online system is manageable, offline analysis becomes costlier with larger data sizes. Given the potentially overwhelming nature of runtime information from production systems, incorporating advanced data processing techniques becomes vital to ensure scalability, thus enabling prompt diagnosis of performance issues.

Thirdly, to make the tools more useful, the implementation must accommodate modifications in production system designs. For instance, Argus necessitates developers to accurately identify and insert tracing points based on domain knowledge. Simplifying instrumentation for production systems is imperative, especially since different third-party libraries may be used over time. Feasible options include defining binary instrumentation templates or providing utilities for vendors to modify their source code, drawing from our experience.

Beyond Performance Issues on CPU Usage Both vProf and Argus primarily focus on enhancing diagnostic tools to address performance issues related to CPU usage. However, application performance is significantly impacted by many other factors. These factors include excessive memory

usage, I/O blocking, and energy leaks, among others.

Nonetheless, the fundamental concept underpinning vProf and Argus is the integration of supplementary runtime information into diagnostic tools. This foundational concept can be extended to address various other performance issues. For instance, integrating supplementary runtime data into profilers that measure wall-clock time enables the diagnosis of performance issues stemming from inappropriate I/O blocking, paging, or synchronization.

Furthermore, this fundamental concept can enhance the effectiveness of tools designed to detect excessive resource consumption, such as memory leaks. It aids in distinguishing between inherent intensive resource usage and wasteful resource consumption by comparing the buggy case to normal use cases.

Improve profiling strategy in vProf for multi-threading vProf relies on `gprof` for support in multi-threaded applications. In general, `gprof` delivers the SIGPROF signal to an arbitrary thread depending on the thread scheduling in the system. Specifically, it sets a timer and monitors the CPU time used by a process. When the timer runs out, it delivers the SIGPROF signal and saves the runtime information from the context of the running thread. Therefore, vProf might still experience sampling bias. Different scheduling algorithms for the threads can result in different profiling results. This bias could lead to the omission of specific thread executions, potentially weakening the effectiveness of vProf.

One potential area for future work is to adjust the profiling strategy for multi-threaded applications, taking into account the characteristics of workloads and the scheduling strategies in the system.

Generalize Argus to Other Systems Our evaluation of the annotated causal tracing tool, Argus, has been implemented and has demonstrated promising results on macOS. However, the generalizability of Argus to other systems has not yet been explored.

It is worth noting that inherent inaccuracies are common in diagnostic tools across various operating systems, as well as in distributed systems. The concept of annotating edge types based

on semantics and introducing error-tolerant diagnostic algorithms using these edge types shows promise in mitigating inaccuracies and enhancing effectiveness.

Nevertheless, the construction of the annotated trace graph relies on domain-specific knowledge, such as typical programming paradigms in operating systems. While it is possible that the techniques in Argus may not be universally applicable to other operating systems, modern operating systems often share many similarities and draw inspiration from each other's designs. For instance, they frequently offer tracing facilities like ETW [25] in Windows and LTTng in Linux [37], which could potentially support techniques similar to those used in Argus. Therefore, we are hopeful that our ideas are generally applicable.

In conclusion, there exist numerous potential avenues for further enhancing the effectiveness of diagnostic tools, in addition to the ones discussed above.

Conclusion

Despite the existence of various diagnostic tools, diagnosing performance issues often remains ineffective.

Profilers, although mature and commonly recommended for diagnosing performance issues, tend to produce inaccurate results, lack the capability to capture data flow, and determine why a function is slow. Causal tracing tools, designed to tackle performance issues with causal relationships spanning across components, encounter limitations due to inaccuracies in the tracing graphs they generate, thus undermining the effectiveness of performance diagnosis.

This dissertation critically examines the sources of these inherent inaccuracies and underscores the essential role of capturing supplementary runtime information to overcome them. To this end, it introduces a novel profiling methodology named value-assisted profiling (vProf), built upon the existing profiler `gprof`. vProf measures not only execution costs but also data-flow in a program to effectively pinpoint the root causes of performance issues. Similarly, in a bid to enhance the efficacy of causal tracing tools, the dissertation presents a pioneering causal tracing tool called Argus. This tool leverages semantic inferences drawn from supplementary runtime information to annotate the edges of tracing graphs. Utilizing a beam-search algorithm guided by these edge types, Argus mitigates and tolerate the inherent inaccuracies and improves the effectiveness of performance diagnosis.

Our evaluation indicates that both of them are practical for real-world performance bugs.

References

[1] *Activity Monitor User Guide: Run system diagnostics in Activity Monitor on Mac*, https://support.apple.com/guide/activity-monitor/run-system-diagnostics-actmntr2225/mac.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance Debugging for Distributed Systems of Black Boxes," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, New York, NY, USA, Oct. 2003, pp. 74–89.

[3] T. W. Anderson and D. A. Darling, "Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes," *The Annals of Mathematical Statistics*, vol. 23, no. 2, pp. 193 –212, 1952.

[4] Apache, *httpd: Apache Hypertext Transfer Protocol Server*, https://httpd.apache.org/.

[5] M. Attariyan, M. Chow, and J. Flinn, "X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, Hollywood, CA, USA, Oct. 2012, 307—320.

[6] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for Request Extraction and Workload Modelling," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, Dec. 2004, pp. 259 –272.

[7] E. Bendersky, *Parsing ELF and DWARF in Python*, https://github.com/eliben/pyelftools.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, May 2014, pp. 468–479.

[9] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining Temporal Invariants from Partially Ordered Logs," in *Workshop on Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, Cascais, Portugal, Oct. 2011.

[10] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models," in *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Szeged, Hungary, Sep. 2011, pp. 267–277.

[11] D. BRS and T. Balathandayuthapani, *Recovery Failure: Loop of Read Redo Log up to LSN*, https://jira.mariadb.org/browse/MDEV-21826.

[12] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the 2004 USENIX Symposium on Annual Technical Conference*, Boston, MA, USA, Jun. 2004.

[13] D. Capps, C. Capps, D. Sawyer, J. Lohr, G. Dowding, G. Little, C. Capps, R. Miller, S. Faibish, R. Wang, T. Waghmare, Y. Zhang, V. Miller, N. Principe, Z. Jones, U. Bapat, W. Norcott, I. Crawford, K. Collins, A. Slater, S. Rhine, M. Wisner, K. Goss, S. Landherr, B. Smith, M. Kelly, A. Dr. CYR, R. Dunlap, M. Montague, D. Million, G. Brebner, J.-M. Zucconi, J. Blomberg, H. Benny, D. Boone, E. Habbinga, K. Strecker, W. Wong, J. Root, F. Bacchella, Z. Xue, Q. Li, D. Sawyer, V. Bojaxhi, B. England, L. Vikentsi, and A. Skidanoy, *IOzone filesystem benchmark*, https://www.iozone.org/.

[14] *Catapult: Chromium Benchmark*, https://chromium.googlesource.com/catapult.

[15] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," in *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks*, Bethesda, MD, USA, Jun. 2002, pp. 595–604.

[16] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective Statistical Debugging via Efficient Path Profiling," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 34–44.

[17] *Chromium Issue 115920: Response Time can be Really Long with Some IMEs (e.g. Pinyin IME (Apple), Sogou Pinyin IME)*, https://bugs.chromium.org/p/chromium/issues/detail?id=115920.

[18] *Cocoa Fundamentals Guide*, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/Introduction.html.

[19] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, Dec. 2004, pp. 231–244.

[20] R. Coker, *Bonnie++ Benchmarking*, https://www.coker.com.au/bonnie++/.

[21] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-Sensitive Profiling," pp. 89–98, Jun. 2012.

[22] C. Curtsinger and E. D. Berger, "COZ: Finding Code that Counts with Causal Profiling," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, Monterey, CA, USA, Oct. 2015, pp. 184–197.

[23] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, TX, USA, Oct. 2017, pp. 1285–1298.

[24] M. J. Eager, "Introduction to the DWARF Debugging Format," pp. 1–11, 2012.

[25] *Event Tracing for Windows*, `https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-`, 2002.

[26] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A Pervasive Network Tracing Framework," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, Cambridge, MA, USA, Apr. 2007, pp. 271–284.

[27] T. Gleixner, I. Molnar, *et al.*, *perf: Linux Profiling with Performance Counters*, `https://perf.wiki.kernel.org/index.php/Main_Page`.

[28] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A Call Graph Execution Profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, Boston, MA, USA, Jun. 1982, pp. 120–126.

[29] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance Debugging in the Large via Mining Millions of Stack Traces," in *Proceedings of the 34th International Conference on Software Engineering*, Zurich, Switzerland, Jun. 2012, pp. 145–155.

[30] X. Han, D. Carroll, and T. Yu, "Reproducing Performance Bug Reports in Server Applications: The Researchers' Experiences," *Journal of Systems and Software*, vol. 156, pp. 268–282, 2019.

[31] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011, 71–83.

[32] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance Regression Testing Target Prioritization via Performance Risk Analysis," in *Proceedings of the 36th International Conference on Software Engineering*, May 2014, pp. 60–71.

[33] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," in *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, USA, Jul. 1999, pp. 135–143.

[34] *Instruments Overview*, `https://help.apple.com/instruments/mac/current/#/dev7b09c84f5`.

[35] Intel, *Intel VTune Profiler*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[36] B Jacob, P Larson, B. Leitao, and S. Silva, *SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems*, 2016.

[37] M. Jeanson, J. Galarneau, and P. Proulx, *LTTng: Linux Tracing Toolkit - next generation*, https://lttng.org.

[38] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and Detecting Real-World Performance Bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, Beijing, China, Jun. 2012, pp. 77–88.

[39] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Reno/Tahoe, Nevada, USA, Oct. 2010, 241–255.

[40] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An End-to-End Performance Tracing and Analysis System," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, Shanghai, China, Oct. 2017, pp. 34–50.

[41] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu, *Kernel Probes (Kprobes)*, https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[42] T. A. Khan, I. Neal, G. Pokam, B. Mozafari, and B. Kasikci, "DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, Jul. 2021, pp. 163–181.

[43] J. Levon, *OProfile: A System Profiler for Linux*, https://oprofile.sourceforge.io/about.

[44] S. J. Lewis, *A Performance Debugging Story*, https://twitter.com/SarahJamieLewis/status/1397313537538592769.

[45] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, 2014, 1013–1024.

[46] R. M. Llamas, *iBench: The Cocoa Benchmark*, https://ibench.sourceforge.io.

[47] LLVM, *Writing an LLVM Pass*, https://llvm.org/docs/WritingAnLLVMPass.html.

[48] *Mac OS X System Font Registration Manager*, https://www.unix.com/man-page/osx/8/fontd/.

[49] J. Mace and R. Fonseca, "Universal Context Propagation for Distributed System Instrumentation," in *Proceedings of the 13th European Conference on Computer Systems*, Porto, Portugal, Apr. 2018, pp. 1–18.

[50] J. Mace, R. Roelke, and R. Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, Monterey, CA, USA, Oct. 2015, pp. 378–393.

[51] MariaDB, *The Open Source Relational Database*, https://mariadb.org.

[52] M. Mayer, *In Search of... A better, faster, stronger Web*, Apr. 2019.

[53] Microsoft, *Office is now Microsoft 365*, https://office.com, 2020.

[54] D. Mosberger-Tang, A. Sharma, D. Watson, *et al.*, *The libunwind Project*, https://savannah.nongnu.org/projects/libunwind/.

[55] M. Nikulin, "Hellinger Distance," *Encyclopedia of Mathematics*, 2001.

[56] A. Nistor, T. Jiang, and L. Tan, "Discovering, Reporting, and Fixing Performance Bugs," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 237–246.

[57] PostgreSQL, *The World's Most Advanced Open Source Relational Database*, https://www.postgresql.org.

[58] *Quiver Crash When Applying Bullet Points on Multiple Lines of Text*, https://github.com/HappenApps/Quiver/issues/21.

[59] *Quiver: The Programmer's Notebook*, https://happenapps.com.

[60] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "AppInsight: Mobile App Performance Monitoring in the Wild," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, Hollywood, CA, USA, Oct. 2012, pp. 107–120.

[61] Redis, *A Vibrant, Open Source Database*, https://redis.io.

[62] X. J. Ren, S. Wang, Z. Jin, D. Lion, A. Chiu, T. Xu, and D. Yuan, "Relational Debugging — Pinpointing Root Causes of Performance Problems," in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, Jul. 2023, pp. 65–80.

[63] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the Unexpected in Distributed Systems," in *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, San Jose, CA, USA, May 2006, pp. 115–128.

[64] riverbed, *How Poor Application Performance Impacts the Enterprise in the Age of the Cloud*, `https://www.riverbed.com/resource/white-paper/how-poor-application-performance-impacts-enterprise-age-cloud/`, Apr. 2020.

[65] D. Rogora, A. Carzaniga, A. Diwan, M. Hauswirth, and R. Soulé, "Analyzing System Performance with Probabilistic Performance Annotations," in *Proceedings of the 15th European Conference on Computer Systems*, Heraklion, Greece, Apr. 2020, pp. 1–14.

[66] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, "Full-system Critical Path Analysis," in *Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, TX, USA, Apr. 2008, pp. 63–74.

[67] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger, "Principled Workflow-Centric Tracing of Distributed Systems," in *Proceedings of the 7th ACM Symposium on Cloud Computing*, Santa Clara, CA, USA, Oct. 2016, pp. 401–414.

[68] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google, Tech. Rep., Apr. 2010.

[69] L. Song and S. Lu, "Statistical Debugging for Real-World Performance Problems," in *Proceedings of the 2014 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, Portland, OR, USA, Oct. 2014, pp. 561–578.

[70] *TeXstudio Freezes When bib File is Updated in the Background*, `https://github.com/texstudio-org/texstudio/issues/288`.

[71] *The Chromium Projects*, `https://www.chromium.org`.

[72] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger, "Stardust: Tracking Activity in a Distributed Storage System," in *Proceedings of the 2006 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Saint Malo, France, Jun. 2006, pp. 3–14.

[73] *Trace: Configure, Record, and Display Kernel Trace Events*, `https://opensource.apple.com/source/system_cmds/system_cmds-671.10.3/trace.tproj`.

[74] L. Weng, *Sequel-Ace Fix Reconnect Timeout - Accept SSH Password after Network Connection Reset*, `https://github.com/Sequel-Ace/Sequel-Ace/pull/772`.

[75] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, "Ad Hoc Synchronization Considered Harmful," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010, pp. 163–176.

[76] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct. 2009, pp. 117–132.

[77] S. Zaman, B. Adams, and A. E. Hassan, "A Qualitative Study on Performance Bugs," in *2012 9th Working Conference on Mining Software Repositories (MSR)*, 2012, pp. 199–208.

[78] B. v. d. Zander, J. Sundermeyer, D. Braun, and T. Hoffmann, *TeXstudio: LaTeX Made Comfortable*, `https://www.texstudio.org`.

[79] D. Zaparanuks and M. Hauswirth, "Algorithmic Profiling," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, Beijing, China, Jun. 2012, pp. 67–76.

[80] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda, "Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance," in *Proceedings of 2013 International Conference on Hardware/Software Codesign and System Synthesis*, Montreal, QC, Canada, Sep. 2013.