Fall 2013

Enabling Richer Insight Into Runtime Executions Of Systems

Karthik Swaminathan Nagaraj
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By KARTHIK SWAMINATHAN NAGARAJ

Entitled
ENABLING RICHER INSIGHT INTO RUNTIME EXECUTIONS OF SYSTEMS

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

CHARLES E. KILLIAN Chair	PATRICK T. EUGSTER
JENNIFER L. NEVILLE	
DONGYAN XU	
RAMANA R. KOMPELLA	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): CHARLES E. KILLIAN

JENNIFER L. NEVILLE

Approved by: SUNIL PRABHAKAR 10/10/2013
Head of the Graduate Program Date

ENABLING RICHER INSIGHT INTO RUNTIME EXECUTIONS OF SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Karthik Swaminathan Nagaraj

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana

To my wife Ashwathi,
my parents Nagaraj & Rajeswari,
and sister Narmadha.

ACKNOWLEDGMENTS

There have been many people who have immensely supported me over the last five years, and it would be erroneous on my part to not mention them. To begin with, I thank my wife, Ashwathi, for all her love, encouragement, support and belief in me through this long journey. My parents and sister constantly motivated and pushed me toward scholastic excellence, for which I'll always be grateful.

My advisors Chip Killian and Jennifer Neville provided abundant encouragement and guidance, along with spirited discussions and hands-on experience, while persuading me to attend many conferences. Their recommendations never failed to bring a different perspective to my work. Working with them came with abundant freedom, making for a wonderful PhD experience with little stress. Their unceasing motivation kept my enthusiasm high to bring this dissertation to a successful completion.

I am also thankful to Ramana Kompella, Dongyan Xu and Patrick Eugster for their insights, suggestions and support. It would be remiss of me to not thank my collaborators at Microsoft Research – John (JD) Douceur and Bryan Parno who exposed me to a remarkably different style of research and a great work atmosphere, during my summer internship.

I had the good fortune to work with great individuals in my lab on various projects, and I appreciate the assistance from KC Sivaramakrishnan, Lukasz Ziarek, Hyojeong Lee, Sunghwan Yoo, Hitesh Khandelwal, Jayaram KR and Bo Sang. The CS software and hardware facilities have been particularly patient and helpful in managing our server machines, allowing me to focus on my research.

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

ABBREVIATIONS viii

ABSTRACT ix

1 INTRODUCTION 1
- 1.1 Problems with Managing and Maintaining Systems Software 2
  - 1.1.1 Development Problems 4
  - 1.1.2 Maintenance Problems 4
- 1.2 Instrumentation: Record Runtime State 5
- 1.3 Facilitating Large Scale Analysis Using Statistics and Machine Learning 6
- 1.4 Thesis Statement 7
- 1.5 Contributions 7
  - 1.5.1 Distalyzer: Diagnosing Performance Problems in Systems Software 8
  - 1.5.2 PerfDetect: Tracking Performance Changes of Systems in Code Repositories 9
- 1.6 Road Map 9

2 DISTALYZER: DIAGNOSING PERFORMANCE PROBLEMS IN SYSTEMS SOFTWARE 10
- 2.1 Instrumentation 12
- 2.2 Design 14
  - 2.2.1 Feature Creation 16
  - 2.2.2 Predictive Modeling 20
  - 2.2.3 Descriptive Modeling 22
  - 2.2.4 Attention Focusing 25
- 2.3 Implementation 27
  - 2.3.1 Processing Text Log Messages 27
  - 2.3.2 Distalyzer 28
- 2.4 Case Studies 30
  - 2.4.1 TritonSort 30
  - 2.4.2 HBase 33
  - 2.4.3 Transmission 36
- 2.5 Related Work 41

Page

2.6 Practical Implications 43
2.7 Summary 44

3 TRACKING PERFORMANCE CHANGES OF SYSTEMS IN CODE REPOSITORIES 46
3.1 Designing PerfDetect 49
3.1.1 Performance Metrics Features 52
3.1.2 Windows of Large Software Changes 54
3.1.3 Change Detection by Trend Estimation 55
3.1.4 Queue Extra Experiments 56
3.2 Measurement & Implementation 57
3.2.1 Transmission 59
3.2.2 V8 60
3.2.3 Hadoop 61
3.3 Experiences 62
3.3.1 Quantitative Evaluation 62
3.3.2 Practical Experiences 66
3.3.3 Transmission 66
3.3.4 V8 71
3.3.5 Hadoop 74
3.4 Related Work 75
3.5 Summary 77

4 SUMMARY 78
4.1 Contributions 78
4.2 Future Work 81

LIST OF REFERENCES 84

VITA 91

LIST OF TABLES

Table — Page

2.1 Event and State feature types extracted from the logs 17

2.2 Summary of performance issues diagnosed using DISTALYZER 29

2.3 DISTALYZER diagnosis: Reduction in the number of features for each diagnosed problem 30

3.1 Summary of software and benchmarks used in our measurement and evaluation 58

3.2 Descriptions of the important performance changes observed in Transmission 68

3.3 Descriptions of the important performance changes observed in Chrome V8 and Hadoop 72

LIST OF FIGURES

Figure Page

1.1 Performance management of systems software through automated detection and diagnosis of degradations 8

1.2 Framework for automated analysis of runtime instrumentation to aid in system management 9

2.1 Four-step log comparison process in DISTALYZER leading up to a visual interface 15

2.2 Example Dependency Network (DN). 24

2.3 Algorithm for feature scoring of dependency networks, in attention focusing 26

2.4 DISTALYZER logging API 28

2.5 TritonSort dependency graphs indicating the root cause of the slow runtime 31

2.6 DN for unmodified HBase events 34

2.7 DN for HBase after fixing lookups 34

2.8 Dependency graphs for unmodified Transmission 38

2.9 Dependency graphs for BitTorrent after fixing the NAT problem 40

3.1 Performance measurement study of Transmission and V8 software repositories 50

3.2 Design of PERFDETECT for detecting deviations of performance metrics from their expected behavior 51

3.3 Comparisons of prediction accuracy of PERFDETECT to other methods 64

3.4 Box-plot of Transmission file download performance indicating three performance issues 67

3.5 Three performance problems identified in the V8 JavaScript engine across two benchmarks 73

3.6 Performance improvement on the Hadoop TeraSort benchmark 75

ABBREVIATIONS

API	Application Programming Interface
CDF	Cumulative Distribution Function
CPD	Conditional Probability Distribution
DN	Dependency Network
IP	Internet Protocol
SLA	Service Level Agreement
TCP	Transport Control Protocol
UDP	User Datagram Protocol

ABSTRACT

Nagaraj, Karthik Swaminathan Ph.D., Purdue University, December 2013. Enabling Richer Insight into Runtime Executions of Systems. Major Professors: Charles E. Killian and Jennifer L. Neville.

Systems software of very large scales are being heavily used today in various important scenarios such as online retail, banking, content services, web search and social networks. As the scale of functionality and complexity grows in these software, managing the implementations becomes a considerable challenge for developers, designers and maintainers. Software needs to be constantly monitored and tuned for optimal efficiency and user satisfaction. With large scale, these systems incorporate significant degrees of asynchrony, parallelism and distributed executions, reducing the manageability of software including performance management. Adding to the complexity, developers are under pressure between developing new functionality for customers and maintaining existing programs. This dissertation argues that the manual effort currently required to manage performance of these systems is very high, and can be automated to both reduce the likelihood of problems and quickly fix them once identified. The execution logs from these systems are easily available and provide rich information about the internals at runtime for diagnosis purposes, but the volume of logs is simply too large for today's techniques. Developers hence spend many human hours observing and investigating executions of their systems during development and diagnosis of software, for performance management. This dissertation proposes the application of machine learning techniques to automatically analyze logs from executions, to challenging tasks in different phases of the software lifecycle. It is shown that the careful application of statistical techniques to features extracted from

instrumentation, can distill the rich log data into easily comprehensible forms for the developers.

1 INTRODUCTION

Systems implementations are ubiquitous in today's computing platforms, from operating systems running on end-user machines to large distributed systems running over thousands of machines across the planet. Distributed systems implementations operating in data centers power the bulk critical functions of businesses, with a wide range of services such as banking, serve news content, online retail, web search and social networks to name a few, making them indispensable to humans. All this responsibility falls in the hands of developers and maintainers, who need to keep systems operating continuously without loss of correctness or efficiency. Systems software are no different from any other mechanical machine, in requiring careful and constant monitoring for proper functioning. Today, software developers are frequently under pressure to provide new features in their products for customers, rather than support older versions, to stay ahead of the competition. This commonly leads to growing software projects that accumulate hidden or subtle flaws.

The management of systems is a continuous process in the software development lifecycle, happening at the design, development and maintenance stages. These stages induce changes in the software code or execution environment, which must be validated against developer expectations of behavior. Such validations are typically made on the two externally visible metrics of software – correctness and performance.

Correctness in the functionality of systems programs is paramount in enabling reliability and reputation. An error in the *business logic* of the program can cause trouble such as initiating an unauthorized wire transfer to a foreign bank account, authorize online retailer inventory to be sent to the wrong address, return irrelevant results to a web search request, or even leak a user's sensitive photos to the public. System developers take precautionary measures at all stages to avoid correctness problems including the use of language primitives, unit tests, integration tests and nightly regression tests, to ensure the *safety* of the runtime execution. These tests return a boolean result, which can be used to monitor correctness. Research in software engineering has made great strides in creating tools and techniques to aid developers in maintaining functionally correct software [1–6]. Performance of systems, although commonly misconceived to be secondary or insignificant has been shown to greatly induce customer satisfaction and increased revenues [7, 8].

Performance encompasses a variety of runtime metrics, such as latency, throughput, resource footprint (CPU, Memory, Disk, etc.) or others defined by developers. Since systems software is usually large and complex with many components, developers create multiple benchmarks and workloads that exercise different aspects of the software *jointly* in a comprehensive evaluation. Unfortunately, unlike correctness and functionality, performance is much harder to evaluate and assess. In the case of correctness and functionality, tests essentially ask a binary question about whether the software is behaving correctly. But with performance, there is no simple and straightforward measure to evaluate the software completely. Additionally complicating the performance evaluation is that performance measurements are typically different every time they are evaluated, arising from external factors, inputs, background load, traffic, etc.

The investigation into the runtime properties of systems creates many challenges for developers in managing their software, arising from the scale and complexity of today's software.

1.1 Problems with Managing and Maintaining Systems Software

Today's systems software are commonly large scale and developed over many years of active development, to attain rich functionality and high performance. They utilize asynchrony and parallelism to efficiently leverage resources such as multi-core CPUs and time consuming I/O operations. To scale to large scale data and computation, systems utilize distributed executions to leverage the power of multiple machines, frequently into the hundreds or even thousands of machines. This leads to complex software executions that are hard to predict and understand. The complexity arises from internal and external non-determinism, software randomness, unpredictable network delays and machine failures, to name a few. For example, if a process receives messages from two other remote processes on different channels at the same time, the messages could be delivered in any order deemed appropriate by the OS transport layer.

Compounding this complexity is the fact that software is typically developed by teams of programmers, often relying on external components and libraries developed independently, such that generally no one developer is fully aware of the complex interactions of the sub-components.

Large software projects are separated into components, often clearly defined by functionality such as business logic, networking, fault tolerance, etc. Component interactions are facilitated through thin interfaces designed with pre-defined semantics. Inter-component interactions can often be delicate and subtle, requiring careful use to avoid adverse outcomes. For instance, consider the interface between an application and the transport, with the implicit assumption that the application timeouts are larger. Here, an unplanned increase in the transport layer's timeout could result in unexpected behavior in the application. In distributed executions, such interplay also happens across distributed sites over the network, requiring software to be more robust. Most performance problems in systems originate from faulty interactions between components, that the developers had not already anticipated.

With the greatest trouble in managing systems software being bugs or deficiencies in the software code, we next discuss the kinds of problems that occur at each stage of development, maintenance and design.

1.1.1 Development Problems

At time of software development, many concurrent software changes go into a common repository resulting from quick development of both related and independent functionality. Since software is developed in groups of tens or hundreds of programmers who may not be aware of code changes from their peer programmers, it is not uncommon for correctness or performance bugs to manifest. A rigorous and comprehensive set of benchmarks and nightly tests are necessary to avoid unexpected surprises. Nevertheless, the complexity of systems software triggers intricate execution paths during integration tests that result in faulty or inefficient behavior, that don't necessarily flag as clear failures.

1.1.2 Maintenance Problems

Even well built systems occasionally fail or perform poorly in deployment due to unexpected changes in the workload characteristics, background load, system updates, machine failures, etc. Swift diagnosis of problems in deployed systems is of principal importance to maintainers. A bug in a particular component could trigger erroneous (possibly non-crash) behavior, that traverses multiple component boundaries and reflect as faulty behavior at a remote component. Thus diagnosis needs to extract both the symptom and the root cause. This needs to be performed by systematically extracting the reverse sequence of actions from the observed fault to the actual root cause, easily becoming a tedious process amidst thousands of concurrent operations. This is especially made difficult by opaque component executions and inter-leavings of concurrent operations.

To manage software execution and investigate problems, developers need a mechanism to look into the execution state of programs. Software runtime instrumentation is the most popular, ubiquitous and comprehensive method for recording program state, for purposes of monitoring or diagnosing software behavior. With a wealth of information already available within existing systems software in this form, it is only rational to leverage this to provide richer insight into system behaviors.

1.2 Instrumentation: Record Runtime State

Instrumentation is the process of monitoring the runtime aspects of a system, collected and stored as a trace of the execution. The scope of instrumentation can include values of program variables/states, signals received, system calls performed and resource utilization. Instrumentation data is often unstructured and labeled with the recording time, verbosity of the message, severity of the occurrence, followed by a software-specific message. Instrumentation forms the common substrate across different libraries, versions and even software implementations allowing developers to seamlessly inspect runtime executions of programs. Developers collect instrumentation data for a variety of purposes such as monitoring resource usage [9], identifying occurrence of fatal actions [4, 10], modeling components usage [11], tracing execution paths [12–14], etc.

Instrumentation is ubiquitous, because many developers already instrument their programs [15], using one of their preferred logging frameworks (printf, log4j [16] or others [12,13,17]). Moreover, instrumentation data is frequently recorded in a method that separates it from the execution (e.g. text logs), and hence allows for decoupled analysis.

Instrumentation is primarily used to diagnose faults in the software, using error or warning messages. However, existing instrumentation of program state and/or performance metrics also aids in diagnosis of performance degradations, by identifying bottleneck components. Unfortunately, as software continues to grow in size and complexity, the volume of instrumentation data also increases. Today's systems frequently instrument GBs worth of data every hour [18, 19]. The contributions of logging to debugging are so deeply ingrained that systems typically are not successful without a significant amount of effort expended in logging infrastructures. As data volume grows, the search for root causes of problems becomes notably challenging.

Manual analysis of large volumes of instrumentation data is *hard*, and is a big burden on developers [14, 20]. Apart from the sheer volume of information instrumented from the programs, one difficulty is the unpredictable ordering of instrumented events, stemming from complexity of parallel and distributed executions, that cause large degrees of non-determinism in executions. Currently, most developers naïvely manually browse through this data in an attempt to find useful pieces of information [4]. Unfortunately, most developers are equipped with brittle and unsophisticated tools such as `sed`, `awk`, `grep`, etc. together with visualization tools [21–23]. This results in very little automation and limits the extent and efficiency of such analysis.

1.3 Facilitating Large Scale Analysis Using Statistics and Machine Learning

Statistical and machine learning methods are useful for summarization, categorization and prediction tasks on large volumes of data. By extracting higher order statistics from runtime values of programs, the unpredictability of systems executions can be quantified. More specifically, by observing runtime values such as program values or timing information of a large collection of unpredictable executions, summary statistics can distinguish patterns from truly random events. As opposed to manual techniques for analyzing logs, machine learning techniques can be easily automated to scale to complex and repetitive tasks.

As deployed systems grow to become hard to manage, there is an emergence of analyzing instrumentation logs using machine learning. These applications range from analysis of raw system logs [4, 14], diagnosing faults by tracking component dependencies [24, 25], summarizing execution behavior for profiling and diagnosis [11, 20] and anomaly detection [26]. However, these applications have only grazed the boundary of a large space of problems.

One of the biggest challenge in this space is identifying the right modeling of the input dataset – program execution state, into an appropriate and amenable form, combined with using suitable machine learning methods. One needs to transform the instrumentation logs (frequently in text form) into tabular views of numerical or categorical data. Naturally, identifying such a transformation can be challenging, as it is heavily dependent on both the software management and machine learning task.

1.4 Thesis Statement

Manual techniques for investigating runtime executions of large systems are cumbersome, inefficient and not scalable. Machine learning can be used to automatically detect and diagnose performance degradations of software systems, by analyzing their instrumented runtime executions.

Specifically this dissertation posits the following:

- Performance problems in systems implementations can be automatically diagnosed by comparing instrumentation logs from a set of executions with unacceptable performance, to logs from executions with acceptable performance.
- Significant changes in performance characteristics of software during development, that diverge from expected behavior can be automatically detected.

1.5 Contributions

This thesis explores the application of machine learning to automate the management of performance degradations in systems software as shown in Fig. 1.1. Firstly, we designed Distalyzer for automated *diagnosis* of performance problems in systems implementations by automatically analyzing unstructured logs. Next, we *identify* problems in software at an early stage in the development environment, by enhancing software repositories with the intelligence to track and manage performance metrics.

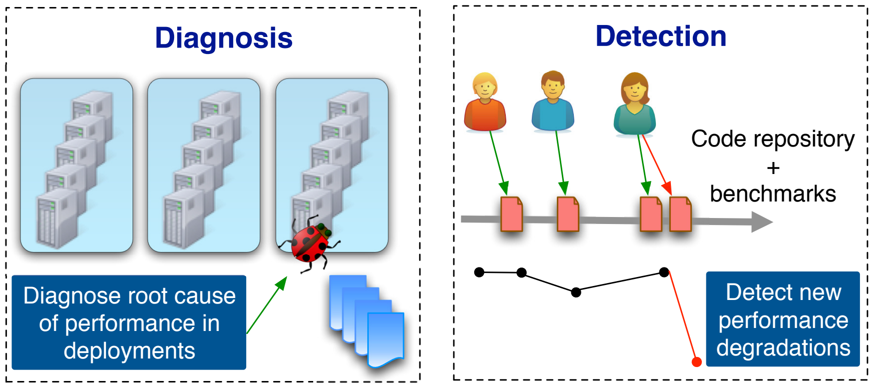

Diagnosis
Diagnose root cause of performance in deployments
Detection
Code repository + benchmarks
Detect new performance degradations

Figure 1.1.: Performance management of systems software through automated detection and diagnosis of degradations

The common foundation to these machine learning analysis is the automated analysis of program instrumentation. Fig. 1.2 describes the framework for such analyses, from the instrumentation of programs, to extracting meaningful metrics from the data, to applying statistical and mathematical models (machine learning) on the transformed data. Finally the developer is presented with a simplified description of the analyzed deteriorations. We describe the two dimensions briefly below:

1.5.1 Distalyzer: Diagnosing Performance Problems in Systems Software

Distalyzer diagnoses the root cause of performance problems from existing systems logs by differentiating the given execution with bad performance from another comparable execution with acceptable performance. It can work with little or no extra instrumentation added to existing logs, and achieves automated diagnosis by supplementing the logs with minimal structure. Distalyzer uses event occurrences and program values extracted from the logs to compare sets of logs and determines the root cause that most affects overall performance. It is specifically designed for use by non-expert developers who are not familiar with the whole system design or code base. The effectiveness of these techniques is demonstrated through the diagnosis of 6 real performance issues in 3 systems.

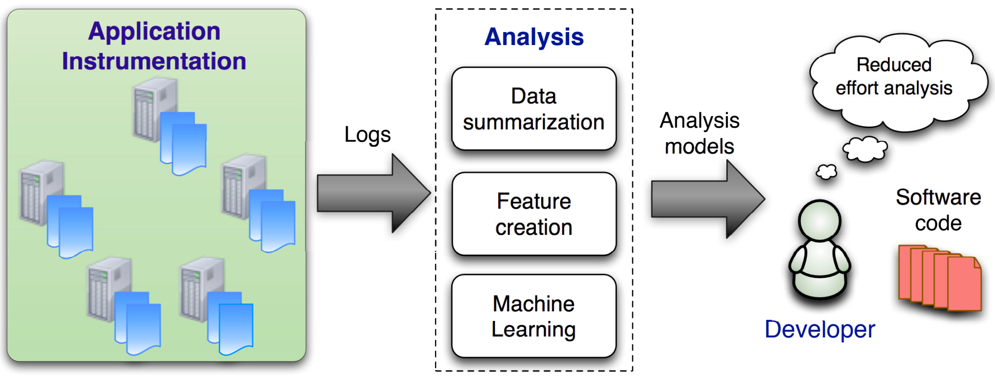


Figure 1.2.: Framework for automated analysis of runtime instrumentation to aid in system management

1.5.2 PerfDetect: Tracking Performance Changes of Systems in Code Repositories

Software undergoes significant changes on a daily basis in code repositories, many of which impact the performance of one or more benchmark metrics. We conduct a detailed measurement study of three software repositories to highlight classes of performance issues and their impact on the software. We design PerfDetect to manage daily performance and detect code changes that cause significant divergence in performance metrics. We demonstrate its effectiveness through qualitative evaluations of performance issues, and quantitative comparisons with state-of-the-art detection techniques.

1.6 Road Map

Chapter 2 describes Distalyzer and our novel techniques in leveraging logs to diagnose performance problems, together with a demonstration of Distalyzer on three real distributed systems. Next, Chapter 3 discusses PerfDetect, a tool for detecting significant changes of performance in software repositories, along with an evaluation over three systems repositories. Finally, Chapter 4 summarizes this dissertation and indicates avenues for future work.

2 DISTALYZER: DIAGNOSING PERFORMANCE PROBLEMS IN SYSTEMS SOFTWARE

Performance problems are common and frequently discovered in many stable and mature systems implementations. The diagnosis of performance problems is however quite hard, because of the scale and complexity of the implementations, heavy non-determinism in executions and unavailability of comprehensive diagnosis tools.

Transmission [27] and HBase [28] exemplify the scale of this type of software development. Transmission is an open-source implementation of BitTorrent. In 2008, after three years of development, it became the default BitTorrent client for Ubuntu and Fedora, the two most popular Linux distributions. In the last two years alone, 15 developers committed changes to the codebase, not counting patches/bugs submitted by external developers. HBase is an open-source implementation of BigTable [29], depending on the Hadoop [30] implementation of the Google File System [31]. HBase has grown very popular and is in production use at Facebook, Yahoo!, StumbleUpon, and Twitter. HBase's subversion repository has over a million revisions, with 21 developers from multiple companies contributing over the last two years.

Given the activity in these projects, it not surprising that, in our experiments, we observed performance problems, despite their mature status. In systems with many independent developers, large user-bases with differing commercial interests, and a long history, diagnosis and correction of performance issues can be a daunting task—since no one developer is likely to be completely familiar with the entire system. In the absence of clear error conditions, manual inspection of undesirable behaviors remains a primary approach, but is limited by the experience of the tester—a developer is more likely to ignore occasional undesirable behavior if they do not have intimate knowledge of the responsible subsystems.

Recent research on distributed systems has produced several methods to aid debugging of these complex systems, such as execution tracing [12, 13, 32], replay debugging [33], model checking [3,34,35], live property testing [36], and execution steering [37]. However, these methods either require either extensive manual effort, or are automated search techniques focused on discovering specific *error* conditions.

To address the challenge of debugging undesirable behaviors (i.e., *performance* issues), the problem can be scoped toward comparing a set of baseline logs with acceptable performance to another set with unacceptable behavior. This approach aims to leverage the vast log data available from complex, large scale systems, while reducing the level of knowledge required for a developer to use the tool. The state-of-the-art in debugging the performance of request flows [10, 11, 14, 20] also utilizes log data; however, in contrast with this previous work, the focus of this work is analyzing a wider range of system behaviors extracted from logs. This has enabled the development of an analysis tool applicable to more than simply request processing applications. Other work in identifying problems in distributed systems from logs [4] is restricted to identifying anomalous local problems, while it is argued here that poor performance commonly manifests from larger implementation issues.

Distalyzer is a tool to analyze logs of distributed systems automatically through comparison and identify components causing degraded performance. More specifically, given two sets of logs with differing performance (that were expected to have equivalent performance), Distalyzer outputs a summary of event occurrences and variable values that (i) most diverge across the sets of logs, and (ii) most affect *overall* system performance. Distalyzer uses *machine learning* techniques to automatically infer the strongest associations between system components and performance. Contributions of this dissertation include:

- An assistive tool, Distalyzer, for the developer to investigate performance variations in distributed systems, requiring minimal additional log statements and post processing.

- A novel algorithm for automatically analyzing system behavior, identifying statistical dependencies, and highlighting a set of interrelated components likely to explain poor performance. In addition to the highlighted results, Distalyzer also provides interactive exploration of the extended analysis.
- A successful demonstration of the application of Distalyzer to three popular, large scale distributed systems–TritonSort [38], HBase & Transmission–identifying the root causes of six performance problems. In TritonSort, a recently identified performance variation is analyzed – the TritonSort developers surmised Distalyzer could have saved them 1.5 days of debugging time. In follow-up experiments on Transmission and HBase, once fixed the identified problems were fixed, their performance was boosted by 45% and 25% respectively.

2.1 Instrumentation

Distalyzer derives its analysis based on the data extracted from logs of distributed systems executions. Hence, we describe the process of obtaining and preparing the logs for analysis, before the actual design in Section 2.2. Applying our modeling to the logs of systems requires that some amount of its meaning is provided to Distalyzer. Inherently, this is because we are not seeking to provide natural language processing, but instead to analyze the structure the logs represent. Xu et al. [4] have considered the automatic matching of log statements to source code, which requires tight coupling with programming languages to construct abstract syntax trees. In contrast, Distalyzer aims to stay agnostic to the source code by abstracting the useful information in the logs. We describe this in more detail below.

The contributions of logging to debugging are so deeply ingrained that systems typically are not successful without a significant amount of effort expended in logging infrastructures. Distalyzer assumes that the collection of logs has not affected the performance behaviors of interest in the system. This is a standard problem with logging, requiring developers to spend much effort toward efficient logging infrastructures. Logging infrastructures range from free text loggers like *log4j* [16], to fully structured and meaningful logs such as Pip [12] and XTrace [13]. Unfortunately, the common denominator across logging infrastructures is not a precise structure indicating the meaning of the logs.

Consider Pip [12], a logging infrastructure which provides log annotations indicating the beginning and ending of a task, sending and receiving of messages, and a separate log just as an FYI (a kind of catch-all log). Every log also indicates a path identifier that the log belongs to, thus it is possible to construct path trees showing dependencies between tasks within paths. This kind of instrumentation has been leveraged by Sambasivan et al. [20] to compare the path trees in systems logs. Unfortunately, this detail of logging is neither sufficient (it does not capture the instances of value logging, and does not adequately handle tasks which belong to multiple flows), nor is it widely available. A more commonly used logging infrastructure, *log4j*, provides a much more basic scheme - logs are associated with a "type," timestamp, priority, and free text string. It then remains to the developer to make sense of the logs, commonly using a brittle set of log-processing scripts.

As a compromise between fully meaningful logs and free-text logs, we work to find a middle-ground, which can be applied to existing logs without onerous modifications to the system being investigated. Our insight is that logs generally serve one of two purposes: *event log messages* and *state log messages*.

Event log message. An event log message indicates that some event happened at the *time* the log message was generated. Most of the logging for Pip falls into this category, in particular the start or end of tasks or messages. Other examples of such logs include logs that a particular method was called, branch of code taken, etc. These logs are often most helpful for tracing the flow of control with time between different components of the system.

State log message. A state log message indicates that at the time the log message was generated, the *value* of some system variable is as recorded. Typically, a state log message does not imply that the value just became the particular value (that would instead be an event log message), but merely that at the present time it holds the given value. State log messages are often printed out by periodically executed code, or as debugging output called from several places in the code. State log messages are often most helpful for capturing snapshots of system state to develop a picture of the evolution of a system.

Distinguishing state and event log messages is an important step, that allows us to tailor our modeling techniques to treat each in kind. We will commonly refer to these values as the *Event Variables* and the *State Variables*. A practical artifact of this approach is that we can simply use the system's existing infrastructure and logging code to generate logs, and then write a simple script to translate logs into state and event log messages in a post-processing step (§ 2.3.1). Adopting this approach makes it much easier to apply Distalyzer to a wide range of existing systems, and avoids extra logging overheads at runtime. Additionally, it is possible to integrate our logging library into other logging infrastructures or code-generating toolkits that provide a distinction between state and event log messages, so that no post-processing phase would be required. Furthermore, this strategy allows the incorporation of external system activity monitoring logs for a richer analysis.

2.2 Design

This section presents the design of Distalyzer, an operational tool capable of identifying salient differences between sets of logs, with the aim of focusing the attention of the developer on aspects of the system that affect overall performance and significantly contribute to the observed differences in behavior. Distalyzer involves a multi-step analysis process as shown in Figure 2.1. The input to the workflow is a set of logs with tagged event and/or state log messages, separated by the developer

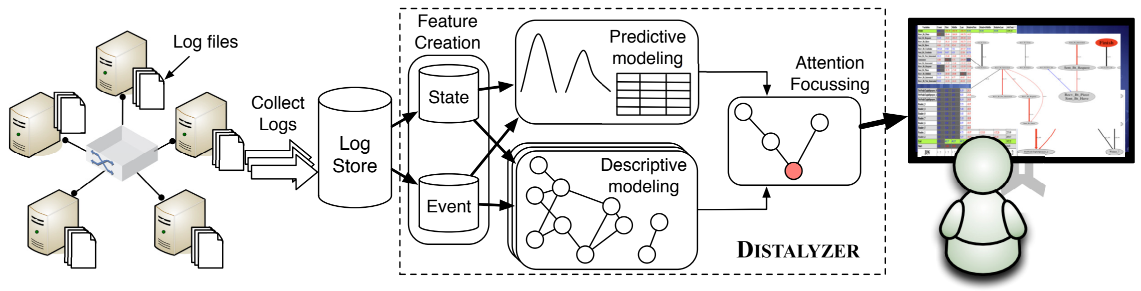


Figure 2.1.: Four-step log comparison process in DISTALYZER leading up to a visual interface

into two classes C_0 and C_1 with different behavior on some performance metric P (*e.g.*, runtime). The choice of performance metric can be easily determined from Service Level Agreement (SLA) metrics. Some example choices for separation of classes are as follows:

- Different *versions* of the same system
- Different *requests* in the same system
- Different *implementations* of the same protocol
- Different *nodes* in the same run

DISTALYZER uses machine learning methods to automatically analyze the input data and *learn* the salient differences between the two sets of logs, as well as the relationships among the system components. Further, DISTALYZER identifies and presents to the developer (for investigation) the most notable aspects of the system likely to contain the *root cause* of the observed performance difference. Specifically the system involves the following four components:

1. **Feature Creation**: A small set of event/state *features* are extracted from each log instance (file) in both classes to make the data more amenable for automated analysis.

2. **Predictive Modeling**: The event/state variables are analyzed with statistical tests to identify which features *distinguish* the two classes of logs. This step

directs attention to the system components that are the most likely causes of performance difference.

3. **Descriptive Modeling**: Within a single class of logs (*e.g.*, C_0), the relationships among event/state variables are learned with dependency networks [39]. The learned models enhance the developer's understanding of how aspects of the system interact and helps to discard less relevant characteristics (*e.g.*, background operations, randomness).

4. **Attention Focusing**: The outputs of steps 2 and 3 are combined to automatically identify a set of interrelated variables that most diverge across the logs and most affect overall performance (*i.e.*, P). The results are graphically presented to the developer for investigation, not only indicating where to look for performance bugs, but also insight into the system itself, obviating the need for the developer to be an expert at all system interactions.

We describe each of these components in more detail below. We note that the user need not be aware of the internals of the statistical or machine learning techniques, and is given an understandable graphical representation of the variables likely to contain the root cause of performance differences. With a clear understanding of the root cause, the developer can spend more time on finding a good fix for the performance bug. In Section 2.4 we present results of using Distalyzer to analyze TritonSort (different versions), BigTable (different requests), and BitTorrent (different implementations).

2.2.1 Feature Creation

The workflow starts with extracting a handful of feature summaries from the logs. The input is two sets of logs C_0 and C_1, classified by the developer according to a performance metric of interest P. For example, the developer may be interested in diagnosing the difference between slow (C_0) and fast (C_1) nodes based on total runtime

Table 2.1: Event and State feature types extracted from the logs

Event times
{First, Median, Last} × {*Absolute*, *Relative*} occurrences
{Count}
State values
{Minimum, Mean, Maximum, Final}
{One-fourth, Half, Three-fourth} × {*Absolute*, *Relative*} snapshots

P. Distalyzer performs offline analysis on the logs, after they have been extracted from the system execution. We assume that a similar test environment was maintained for both sets of logs, including workloads, physical node setup, etc. However, it is not necessary that both classes contain the same number of occurrences of a variable. Also, the two classes are not required to have disjoint non-overlapping values for P. The necessity for similarity can be relaxed further under certain conditions as discussed in Section 2.6.

Distalyzer begins by calculating features from variables extracted from the log instances. We refer to each log instance as an *instance*. Each instance i contains many event and state log messages, which first need to be summarized into a smaller set of summary statistics before analysis. The intuition behind summarizing is that this reduces the complexity of the system execution to a handful of *features* $\mathbf{X_i}$, that are much less prone to outliers, randomness and small localized discrepancies in log statements. Since Distalyzer aims to find the source of *overall* performance problems (and not localized problems as in [4]), a coarse-grained set of features provides a better representative of each instance than every single value within that instance. A smaller set of features are a lesser burden on the developer, but a richer set provides better coverage for different types of problems. Distalyzer aims at striking the right balance between these objectives through our experiences and intuition debugging distributed systems. Distalyzer constructs a set of summary statistics $\mathbf{X}$ from the timestamps of event log messages and the values of numeric variables for state log messages, as described below.

Event Features

The timing of system events is often closely related to overall performance, as it can identify the progress of system components, the presence or absence of noteworthy events, or the occurrence of race conditions between components. We consider a set of event variables Y^e that are recorded in the log instances with timestamps. For example, an instance may refer to a node downloading a file in BitTorrent, where the event log may contain several recv_bt_piece events over time.

To summarize the timing information associated with a particular type of event Y^e in instance i, Distalyzer constructs features that record the time associated with the *first*, *median* and *last* occurrence in Y^e_i. (All timestamps within a log instance i are normalized based on the start time). Specifically, $X^e_{i.1} = min(\mathbf{Y^e_i}[\mathbf{t}])$, $X^e_{i.2} = median(\mathbf{Y^e_i}[\mathbf{t}])$, $X^e_{i.3} = max(\mathbf{Y^e_i}[\mathbf{t}])$. In addition, a fourth feature is constructed that counts the total number of occurrences of Y^e, $X^e_{i.4} = |\mathbf{Y^e_i}|$. Our experience debugging systems suggests that these occurrences capture some of the most useful, yet easily comprehensible, characteristics of system progress. They most commonly indicate issues including but not limited to startup delays, overall slowdown and straggling finishes.

In addition to the above features, which consider the *absolute* timing in instances, we consider the same set of features for *relative* times. Since the instances from C_0 and C_1 may have different total times, normalizing the times within each instance to the range $[0, 1]$ before computing the features will yield a different perspective on event timings. For example, in BitTorrent, it is useful to know that the last outgoing connection was made at 300sec, but for debugging it may be more important to know that it occurred at 99% of the runtime when comparing to another instance where the last connection was made at 305sec, but earlier at 70% of its total runtime. In this case, the divergence in the relative event times is more distinguishing. The top half of Table 2.1 outlines the set of event feature types considered by Distalyzer.

State Features

It is common for some system state variables to be directly or inversely proportional to the performance, and their divergence could be equally important for diagnosis. We consider a set of state variables Y^s that maps to a list of values with their logged timestamps in an instance. For example, in BitTorrent, one of the state variables logged is the download speed of a node, which is inversely proportional to the total runtime performance. DISTALYZER does not attempt to understand the meaning behind the variables or their names, but systematically searches for patterns in the values.

To summarize the information about a particular state variable Y^s in log i, we construct features that record the *minimum*, *average* and *maximum* value in Y_i^s. Specifically, $X_{i.1}^s = min(\mathbf{Y_i^s})$, $X_{i.2}^s = mean(\mathbf{Y_i^s})$, $X_{i.3}^s = max(\mathbf{Y_i^e})$. In addition, to understand the variable values as the system progresses and also give the values context, DISTALYZER constructs features that record the variable values at one-fourth, half and three-fourth of the run. Similar to the events, the relative versions of these snapshots are also considered as feature types. The complete list of state feature types is listed in Table 2.1.

Cost of Performance Differences

Our analysis focuses on leveraging the characteristics of the *average* performance difference between the two classes, thus naïve use of the instances in statistical techniques will fail to distinguish performance in the *tails* of the distribution. For example, in a class of bad performance, there may be 2-3% of instances that suffer from significantly worse performance. Although these cases are relatively infrequent, the *high cost* of incurring such extreme bad performance makes analysis of these instances more important. DISTALYZER automatically detects a significant number of abnormally high/low values of the performance metric, and flags this to the developer for consideration before further analysis. Specifically, DISTALYZER identifies a "heavy" tail for P when the fraction of P_i outside $\overline{P} \pm 3\sigma_P$ is larger than 1.1% (*i.e.*, 4× the expected fraction in a normal distribution). To more explicitly consider these instances in the modeling, we can re-*weight* the instances according to a *cost* function (see e.g., [40]) that reflects the increased importance of the instances in the tail. Section 2.4.2 discusses this further.

2.2.2 Predictive Modeling

In the next stage of the workflow, Distalyzer uses statistical tests to identify the features that most distinguish the two sets of logs C_0 and C_1. Specifically, for each event and state feature X described above (*e.g.*, *first*(recv_bt_piece)), we consider the distribution of feature values for the instances in each class: X_{C_0} and X_{C_1}. Distalyzer uses t-tests to compare the two distributions and determine whether the observed differences are *significantly* different than what would be expected if the random variables were drawn from the same underlying distribution (*i.e.*, the means of X_{C_0} and X_{C_1} are equal). If the t-test rejects the null hypothesis that the $\overline{X}_{C_0} = \overline{X}_{C_1}$, then we conclude that the variable X is *predictive*, *i.e.*, able to distinguish between the two classes of interest. Specifically, we use *Welch's t-test* [41], which is defined for comparison of unpaired distributions of unequal variances. The t-value and significance probability (p) are computed as follows:

$$t = \frac{\overline{X}_{C_0} - \overline{X}_{C_1}}{\sqrt{\frac{\sigma^2_{C_0}}{N_{C_0}} + \frac{\sigma^2_{C_1}}{N_{C_1}}}} \tag{2.1}$$

$$v = \frac{\left(\frac{\sigma^2_{C_0}}{N_{C_0}} + \frac{\sigma^2_{C_1}}{N_{C_1}}\right)^2}{\frac{\sigma^4_{C_0}}{N^2_{C_0}(N_{C_0}-1)} + \frac{\sigma^4_{C_1}}{N^2_{C_1}(N_{C_1}-1)}} \tag{2.2}$$

$$p = 2(1 - CDF(TDist(v), t)) \tag{2.3}$$

where $\sigma^2_{C_0}, \sigma^2_{C_1}$ are the variances, N_{C_0}, N_{C_1} are the sample sizes, v is the degrees of freedom, $TDist$ is the t-distribution. We use a critical value of $p < 0.05$ to reject the null hypothesis and assess significance. An adjustment for multiple comparisons must be made for tests made on the same data source, and is performed with a Bonferroni correction [42] based on the total number of features evaluated (*i.e.,* $NumFeatures \times NumVariables$).

Our use of t-tests is motivated by the fact that we want to identify variables that distinguish the two classes on *average* across many instances from the system. Previous work [20] has used Kolmogorov-Smirnov (KS) tests to distinguish between two distributions of request flows. In that work, the bulk of the two distributions are the same and the KS test is used to determine whether there are *anomalous* values in one of the two distributions. In contrast, our work assumes that the log instances have been categorized into two distinct classes based on developer domain knowledge. Thus the overlap between distributions will be minimal if we can identify a variable that is related to performance degradation in one of the classes. In this circumstance, KS tests are too sensitive (*i.e.*, they will always reject the null hypothesis), and t-tests are more suitable form of statistical test.

Given the features that are determined to be *significant*, the magnitude of the t-statistic indicates the difference between the two distributions—a larger t-statistic can be due to a larger difference in the means and/or smaller variance in the two distributions (which implies greater separation between the two classes). The sign of the t-statistic indicates which distribution had a bigger mean. Among the significant t-tests, we return a list of significant variables ranked in descending order based on the absolute sum of t-statistic over all features. This facilitates prioritized exploration on the variables that best differentiate the two classes.

2.2.3 Descriptive Modeling

In the third component of the workflow, Distalyzer learns the relationships among feature values for each class of logs separately. The goal of this component is to identify salient dependencies among the variables within a single class (*i.e.*, C_0)—to help the developer understand the relationships among aspects of the system for diagnosis and debugging, and to highlight the impact of divergent variables on overall performance P. It is often difficult to manually discover these relationships from the code, because of large code bases. It is also possible that observed variation across the classes for a feature is not necessarily related to performance. For example, a timer period may have changed between the classes without affecting the performance, and such a change can be quickly ignored if the dependencies are understood.

Since we are interested in the overall associations between the features in one class, we move beyond pairwise correlations and instead estimate the *joint distribution* among the set of features variables. Specifically, we use dependency networks (DNs) [39] to automatically learn the joint distribution among the summary statistics $\mathbf{X}$ and the performance variable P. This is useful to understand which sets of variables are inter-related based on the feature values. We construct DNs for the event and state features separately, and within each we construct two DNs for each feature type (*e.g.*, *First.Absolute*), one for instances of class C_0 and one for instances of C_1.

Dependency Network (DN)

Dependency Networks [39] are a graphical model that represents a joint distribution over a set of variables. The primary distinction between Bayesian networks, Markov networks, and dependency networks is that dependency networks are an approximate representation of the joint distribution, which uses with a set of conditional probability distributions (CPDs) that are learned independently.

Consider the set of variables $\mathbf{X} = (X_1, ..., X_n)$ over which we would like to model the joint distribution $p(\mathbf{X}) = p(X_1, ..., X_n)$. Dependencies among variables are represented with a directed graph $G = (V, E)$ and conditional independence is interpreted using graph separation. Dependencies are quantified with a set of conditional probability distributions $\mathcal{P}$. Each node $v_i \in V$ corresponds to an $X_i \in \mathbf{X}$ and is associated with a probability distribution conditioned on the other variables, $p(x_i|\mathbf{x}-\{x_i\})$. The parents of node i are the set of variables that render X_i conditionally independent of the other variables ($p(x_i|pa_i) = p(x_i|\mathbf{x} - \{x_i\})$), and G contains a directed edge from each parent node v_j to each child node v_i ($(v_j, v_i) \in E$ iff $X_j \in pa_i$).

The CPDs in P do not necessarily factor the joint distribution so we cannot compute the joint probability for a set of values $\mathbf{x}$ directly. However, given G and P, a joint distribution can be recovered through Gibbs sampling (see [39] for details). From the joint distribution, we can extract any probabilities of interest.

For example, the DN in Figure 2.2 models the set of variables:

$$\mathbf{X} = \{X_1, X_2, X_3, X_4, X_5\} \tag{2.4}$$

Each node is conditionally independent of the other nodes in the graph given its immediate neighbors (*e.g.*, X_1 is conditionally independent of $\{X_2, X_4\}$ given $\{X_3, X_5\}$). Each node contains a CPD, which specifies a probability distribution over its possible values, given the values of its parents.

Both the structure and parameters of DNs are determined through learning the local CPDs. The DN learning algorithm learns a CPD for each variable X_i, conditioned on the other variables in the data (*i.e.*, $\mathbf{X}-\{X_i\}$). Any conditional learner can be used for this task (*e.g.*, logistic regression, decision trees). The CPD is included in the model as $\mathcal{P}(v_i)$ and the variables selected by the conditional learner form the parents of X_i (*e.g.*, if $p(x_i|\{\mathbf{x} - x_i\}) = \alpha x_j + \beta x_k$ then $PA_i = \{x_j, x_k\}$). The parents are then reflected in the edges of G appropriately. If the conditional learner is not selective (*i.e.*, the algorithm does not select a subset of the features), the DN will be fully connected. (*i.e.*, $PA_i = \mathbf{x} - \{x_i\}$) To build understandable DNs, it is thus desirable to use a selective learner. Since event and state features have continuous values, we use Regression Trees [43, 44] as the conditional learner for the DNs. Their

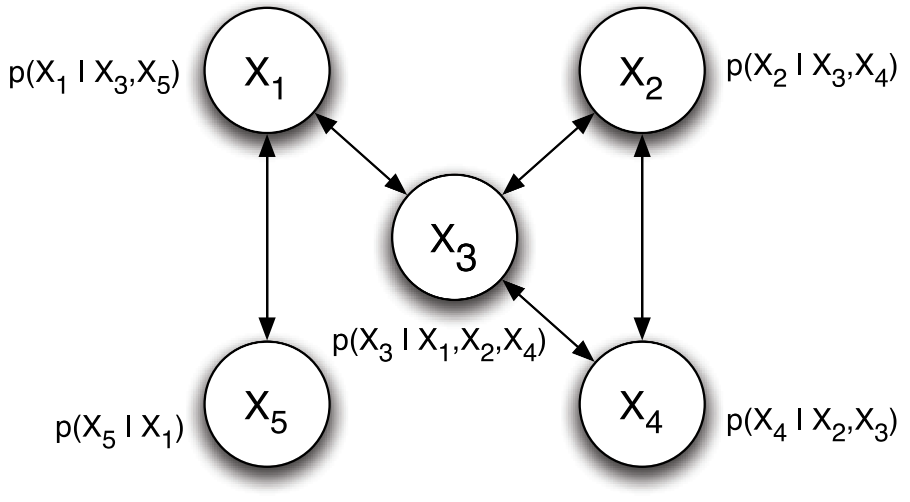


Figure 2.2.: Example Dependency Network (DN).

advantage over standard regression models is that they are selective models, so the features selected for inclusion in the tree will determine the structure of the DN.

Improvements

The graphical visualization of the learned DN are enhanced to highlight to the developer (1) the divergence across classes (sizes of the nodes), (2) the strength of associations among features (thickness of edges), and (3) temporal dependencies among features (direction of edges). Specifically, each feature (node) in the DN is matched with its corresponding statistical t-test value. Since the t-statistics reflect the amount of divergence in the feature, across the two classes of logs, they are used to size the nodes of the graph. Next, for the assessment of relationship strength, we use an input parameter m for the regression tree that controls the minimum number of training samples required to split a leaf node in the tree and continue growing (*i.e.*, a large value of m leads to *shorter* trees because tree growth is stopped prematurely). The dependencies identified in a shorter tree are *stronger* because such variables are most correlated with the target variable and affect a larger number of instances. Thus, we weigh each edge by the value of m for which the relationship is still included in the DN. Finally, we augment the DN graphical representation to include *happens-before* relationships among the features. If a feature value X_i occurs before feature value X_j in all log instances, the edge between X_i and X_j is drawn as directed in the DN.

2.2.4 Attention Focusing

The final component of the workflow automatically identifies the most notable results to present to the user. The goal of this component is to focus the developers attention on the most likely causes of the observed performance differences. The predictive modeling component identifies and presents a ranked list of features that show significant divergences between the two classes of logs. The divergence of a single feature is usually not enough to understand both the root cause of performance problems and their impact on performance—because performance problems often manifest as a causal chain, much like the domino effect. The root cause feature initiates the divergence and forces associated features (down the causal chain) to diverge as well, eventually leading to overall performance degradation.

Moreover, we noticed that divergences tend to increase along a chain of interrelated features, thus the root cause may not have the largest divergence (*i.e.*, it may not appear at the top of the ranking). The descriptive modeling component, on the other hand, identifies the associations among features within a single class of logs. These dependencies can highlight the features that are associated with the performance measure P. To identify likely *causes* for the performance difference, DISTALYZER searches for a small set of features that are *both* highly divergent and have strong dependencies with P. The search procedure for finding the DN that highlights this set is detailed below.

The set of DNs vary across three dimensions: (1) event *vs.* state features, (2) feature type, *e.g.*, *First.Absolute*, and (3) the parameter value m_{min} used to learn the DN. In our experiments, we set m_{min} to one-third of the instances. The aim was to

```
**Input:** Log type: $t$ (State / Event)
**Input:** Log class: $c$, Number of instances: $N$
**Input:** T-tests for all random variables in $(t, c)$
**Input:** DNs for all random variables in $(t, c)$
**Input:** Performance metric: $P$
  $feature\_graphs = \{\}$
  for Feature $f$: $\mathsf{feature\_types}(t)$ do
    $dn = \mathsf{DN}_f(m_{min} = N/3)$
    $cc$ = Connected-component in $dn$ containing $P$
    $tree = \mathsf{maxSpanningTree}(cc)$ rooted at $P$
    $score = 0$
    for Node $n$: $tree$ do
      $score \mathrel{+}= \mathsf{T}_f(n) * dn.\mathsf{weight}(\mathsf{parentEdge}(n))$
    end for
    Append $(score, cc)$ to $feature\_graphs$
  end for
  return $feature\_graphs$ sorted by score
```

Figure 2.3.: Algorithm for feature scoring of dependency networks, in attention focusing

focus on the sufficiently strong relationships among features, and this choice of m_{min} consistently proved effective in all our case studies. However, m_{min} is included as a tunable parameter in the system for the developer to vary and observe the impact on the learned models. DISTALYZER identifies the most notable DN graph for the state and event features separately. Within a particular set, the attention-focusing algorithm automatically selects the feature type with the "best" scoring DN subgraph. To score the DN graphs, they are first pruned for the smallest connected component containing the node P, and then the selected components are scored using the algorithm shown in Fig. 2.3.

The intuition behind the DN subgraph *score* function is that it should increase proportionally with both the node weights (divergence across classes) and the edge weights (strength of association). The node and edge weights are normalized before computing this score. If the developer is interested in biasing the search toward features with larger divergences or toward stronger dependencies, a parameter α can be used to moderate their relative contributions in the score. The feature type with the highest scoring connected component is selected and returned to the developer for inspection.

Section 2.4 describes the outputs of Distalyzer for real systems with observed performance problems. Apart from the final output of the attention focusing algorithm, the developer can also access a table of all the t-test values and dependency graphs for both the state and event logs. This is shown as the final stage in Fig. 2.1.

2.3 Implementation

We describe some implementation details for transforming text logs and developing Distalyzer.

2.3.1 Processing Text Log Messages

The BitTorrent implementations we considered were implemented in C (Transmission [27]) and Java (Azureus [45]), whereas HBase [28] was implemented in Java. The Java implementations used Log4j [16] as their logger. Transmission however used hand-coded log statements. HBase also used Log4j, but did not have any logs in the request path.

For each implementation, we tailored a simple Perl script to translate the text logs into a standard format that Distalyzer accepts. We maintained a simple internal format for Distalyzer. This format captures the timestamp, type of log, and the name of the log. For state logs, the format additionally includes the value of the log. We advocate adopting a similar procedure for analyzing any new system implementation. A developer with domain knowledge on the system should be able to write simple one-time text parsers to translate the most important components of the log instances. To support the translation, we provide a simple library API for logging in a format accepted by Distalyzer (shown in Fig. 2.4). At the beginning of each log instance, the translator calls `setInstance`, which indicates the instance id and

```
setInstance(class, instance_id)
logStateValue(timestamp, name, value)
logEventTime(timestamp, name)
```

Figure 2.4.: Distalyzer logging API

class label for subsequent log messages. It specifically requires marking log messages as event or state logs at translation time by calling one of the two log methods.

2.3.2 Distalyzer

We implemented Distalyzer in Python and C++ (4000 lines of code) using the scientific computing libraries Numpy and Scipy. Distalyzer is publicly available for download [46]. The design allows adding or tweaking any of the event or state features if required by the developer. The Orange data mining library [43] provides regression tree construction, and we implemented dependency networks and Algorithm 2.3 over that functionality. The DOT language is used to represent the graphs, and Graphviz generates their visualizations. The implementation of Distalyzer comprises of many embarrassingly parallel sub-tasks and can easily scale on multiple cores and machines enabling quick processing.

An interactive JavaScript based HTML interface is presented to the developer along with the final output. This immensely helps in trudging through the individual distributions of variables, and also to view the dependency graphs of all features. This has been useful in the post-root cause debugging process of finding a possible fix for the issue. To a good extent, this also helps in understanding some of the non-performance related behavioral differences between the logs. For example, in one case of comparing different implementations, we noticed that either system was preferring the use of different protocol messages to achieve similar goals.

Table 2.2: Summary of performance issues diagnosed using DISTALYZER

System Implementation	**Types of Logs**	**Volume**	**Variables**	**Issues**	**Performance gain**	**New issues**
TritonSort	State, Event	2.4 GB	227	1	n/a	$\times$
HBase (BigTable)	Event	2.5 GB	10	3	22%	$\surd$
Transmission (BitTorrent)	State, Event	5.6 GB	40	2	45%	$\surd$

Table 2.3: Distalyzer diagnosis: Reduction in the number of features for each diagnosed problem

System Implementation	**Input Features**		**Output Features**	
	State	**Event**	**State**	**Event**
TritonSort	1990	217	8	3
HBase (BigTable)	–	70	–	5, 3, 3
Transmission (BitTorrent)	110	203	7, 5	2, 6

2.4 Case Studies

Our goal in these case studies is to demonstrate that Distalyzer can be applied simply and effectively to a broad range of existing systems, and that it simplifies the otherwise complex process of diagnosing the root cause of significant performance problems. We therefore applied Distalyzer across three real, mature and popular distributed systems implementations. Table 2.2 captures the overview of the systems we considered. These systems represent different types of distributed system applications: distributed sorting, databases, and file transfers. We identified previously unknown performance problems with two of these systems, and worked with an external developer to evaluate usefulness of Distalyzer in rediscovering a known performance bug with another. In all cases, Distalyzer significantly narrowed down the space of possibilities without the developer having to understand all components. Due to space constraints, we are unable to describe each action taken by the developer leading to fixes for problems. A user need not be aware of how the tool computes divergences and dependencies to understand Distalyzer's outputs. We describe the outputs of Distalyzer and henceforth straightforward debugging process.

2.4.1 TritonSort

TritonSort is a large scale distributed sorting system [38] designed to sort up to 100TB of data, and holds four 2011 world records for 100TB sorting. We demonstrate the effectiveness of Distalyzer by applying it over logs from a known bug.

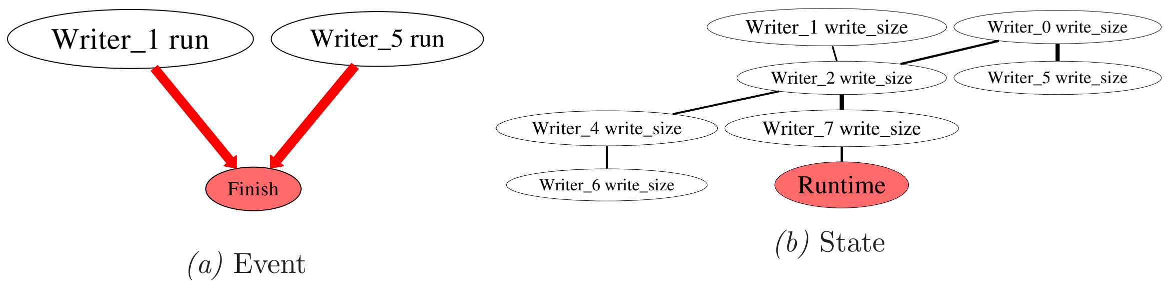


Figure 2.5.: TritonSort dependency graphs indicating the root cause of the slow runtime

We obtained the logs of TritonSort from the authors, taken from a run that suddenly exhibited 74% slower performance on a day. After systematically and painstakingly exploring all stages of the sort pipeline and running micro-benchmarks to verify experimental scenarios, the authors finally fixed the problem. They said that it took "the better part of two days to diagnose". The debugging process for the same bug took about 3-4hrs using DISTALYZER, which includes the implementation time of a log parser in 100 lines of Python code. A detailed analysis of the output of DISTALYZER and the debugging process on these logs follows.

We had access to logs from a 34 node experiment from the slow run that took 383 sec, and also a separate run with the same workload that had a smaller runtime of 220 sec. These naturally fit into two classes of logs with one instance per node, which could be compared to identify the reason for the slowdown. These logs were collected as a part of normal daily testing, meaning no additional overhead for log collection. The logs contained both event and state log messages that represented 8 different stages of the system (Table 2.2). The performance metrics were identified as Finish and runtime for the event and state logs respectively, both indicating the time to completion. Fig. 2.5 shows the final dependency sub-graphs output by DISTALYZER for both event and state logs.

To briefly explain the visualization generated by DISTALYZER, nodes shown to be colored indicate the performance metric and the font size is proportional to the magnitude of the divergence. Edge thickness represents the strength of the dependencies between variables. Directed edges in event graphs indicate that a *happens-before* relationship was identified between the two bounding variables, as described in Section 2.2.4.

The best dependency graph picked for events (*Last* feature type) is shown in Fig. 2.5a, indicating that variables Writer_1 run and Writer_5 run are both significant causes of Finish's divergence. The final stage of TritonSort's pipeline is the writer which basically handles writing the sorted data to the disk. Each stage in TritonSort is executed by multiple thread workers, denoted by the number in the variable. This analysis attributes the root cause of slow runs to highly divergent last occurrences of the writer workers. A quick look at our distribution comparison of the two sets of logs in both the writers indicated that the slow run showed a difference of 90 sec. The performance metric and the writer run distributions also showed an outlier with a larger time than the rest.

Similarly, the DN picked for the states is shown in Fig. 2.5b, where the performance metric Runtime is connected to the subgraph consisting of the write queue size of different writer workers. Although the figure was scaled down for space constraints, it is clear that all the nodes are highly divergent like the total performance. To understand the reason of this divergence, we looked at distributions for Absolute Half (best feature) to learn that writers in the slow run were writing 83% more data. Thus, we concluded the root cause as *slow writers*.

The actual bug had been narrowed down to the disk writing stage, found to be slowing down earlier stages of the pipeline. It was further noticed that a single node was causing most of this delay, which eventually led the authors to discover that the cache battery on that node had disconnected. This resulted in the disks defaulting to write-through and hence the poor performance. Both the top ranked DNs output by DISTALYZER were useful in identifying the bug. We shared these DNs and interactive t-test tables with the author of the TritonSort paper, who had manually debugged this problem. The output root cause was immediately clear to him, and he surmised "had we had this tool when we encountered this problem, it would have been a lot easier to isolate the difference between the bad run and a prior good one".

2.4.2 HBase

BigTable [29] is a large-scale storage system developed by Google, holds structured data based on rows and columns, and can scale efficiently to a very large number of rows and column content. HBase [28] is an open source implementation of BigTable being developed by the Apache foundation. It runs on top of Hadoop Distributed Filesystem (HDFS), and has been tuned and tested for large scales and performance.

In our experiments, we noticed that "Workload D" from the Yahoo Cloud Storage Benchmark (YCSB) [47] had a notable heavy tail distribution of read request latencies. The minimum and median latencies were 0 and 2 msec respectively. However the mean latency was 5.25 msec and the highest latency was as high as 1 second, which is 3 orders of magnitude over the median. Moreover, more than 1000 requests have a latency greater than 100ms. To debug this performance bottleneck in HBase, we would like to be able to compare these slow requests to the huge bulk of fast ones. This task is infeasible manually because these issues manifest only in large experiments (1 million requests), and a sufficiently large number of requests exhibit this behavior. We used Distalyzer to identify and debug three performance bugs in HBase, two of which are described below in detail.

Experimental setup Our testbed consisted of 10 machines with 2.33GHz Intel Xeon CPUs, 8GB RAM and 1Gbps Ethernet connections running Linux 2.6.35.11. Our HBase setup used a single master on a dedicated machine, and 9 region servers (equivalent to BigTable tablet servers), and 1 Million rows of 30kB each were preloaded into the database. The YCSB client was run on the same machine as the master (which was otherwise lightly loaded), with 10 threads issuing parallel requests. Each request is either a read or write for a single row across all columns. "Workload D" consisted of 1 Million operations out of which 5% were writes.

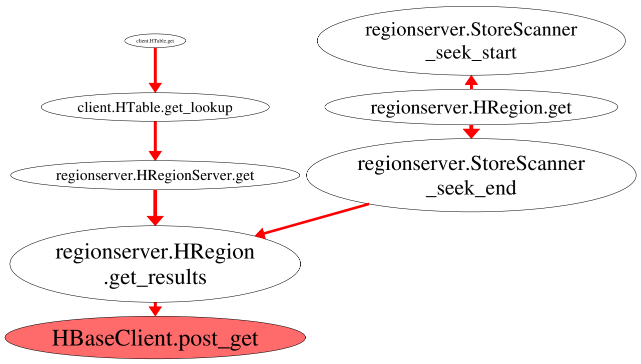


Figure 2.6.: DN for unmodified HBase events

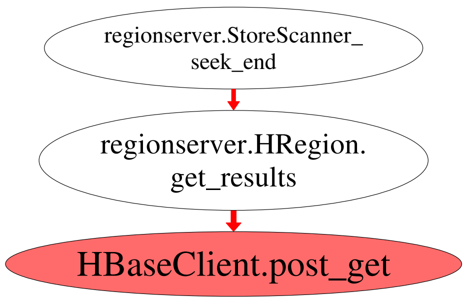


Figure 2.7.: DN for HBase after fixing lookups

The HBase implementation had no log statements in the request flow path, in spite of using the log4j logging library that supports log levels. Therefore, we manually added 10 event logs to the read request path, using the request row key as the identifier. The request logs from the different machines were gathered at the end of the run and bucketed by request ID. The performance metric is the event that signifies the last step in request processing – HBaseClient.post_get.

Fixing the slowest outliers

On applying DISTALYZER to the logs, it detected the presence of a heavy tail in the performance metric (§ 2.2.1) and suggested re-weighting the instances. The weight function used to boost the instances with a large latency was $\lfloor w^{latency} \rfloor$. This is an exponential weight function and we chose a value of $w = 2^{(1/150)}$, with the intuition that instances with $P < 150ms$ will have a weight of 1. Fig. 2.6 shows the best DN of the root cause divergence. All dependency edges are directed because all requests follow the same flow path through the system. We identified two strong associations with large divergences leading up to the performance metric. Each of the chains is considered independently, and we first chose to follow the path leading from client.HTable.get_lookup (the second chain is discussed in § 2.4.2). This chain starts at client.HTable.get which indicates that the HBase client library received the request from YCSB, followed by client.HTable.get_lookup after completion of lookup for the region server handling the given key.

This particular edge leads from a tiny variable to a variable with significant divergence, and domain knowledge indicates that no other event occur between them. client.HTable.get is drawn small because it does not differ considerably between the two classes of logs. As it is connected by a strong directed edge to the larger variable, this indicates the two classes consistently *differ* between these two variables. In this context, the edge represents the operation where the client needs to lookup the particular region server that manages the row, and this is achieved by contacting the master who maintains the mapping. The distributions of this particular event in the t-test table shows that this event created gaps in the request flow of the order of 1000 ms.

When we looked at the logs of the regionserver at the same time these requests were being delayed, we noticed that the server was throwing a NotServingRegionException. This is given by the server when it does not serve a region that was specifically requested. This happens when a region was moved to another server for load balancing. The client possesses a stale cache entry for the region, and hence receives this exception. The client was catching this exception as an IOException, and treated it as a server failure. This triggers an exponential back off procedure that starts at 1 sec. According to the Bigtable description [29], the client immediately recognizes a stale cache and retries with the master leading to an overhead of just 2RTTs. We came up with a fix for this issue, by treating the exceptions correctly and extracting the NotServingRegionException, and retrying immediately. This fixed the requests with latencies over 1 second.

Operating System effects

Distalyzer was used again to analyze the new logs to find the cause of the other delays. Since the distribution skew was lesser than the threshold, the weighting function was not used anymore. The best DN is shown in Fig. 2.7, and closely resembles the right chain of Fig. 2.6. In fact, this root cause was also identified in the initial step as a second significant root cause, but was not chosen for inspection. Here, the variables regionserver.StoreScanner_seek_end and regionserver.HRegion.get_results chain up as the root cause.

The default Linux I/O scheduler since version 2.6.18 is Completely Fair Queuing (CFQ), and it attempts to provide fairness between disk accesses from multiple processes. It also batches requests to the disk controller based on the priority, but it does not guarantee any completion times on disk requests. Since only the HBase process was accessing the disk on these machines, we believed that this scheduling policy was not well suited to random block reads requested by HBase. Another available I/O scheduler in Linux is the deadline scheduler, which tries to guarantee a start service time for requests. Hence the deadline scheduler would be more suited toward latency sensitive operations.

After we applied the I/O scheduler change, we ran the same experiment again to understand if this improved the latencies of the slow requests. The number of slow requests ($\geq$100ms) reduced from 1200 to just under 500 – a 60% reduction. Also, the mean latency for the workload dropped from 5.3ms to 4ms, which is a 25% overall improvement in the read latency, confirming deadline is appropriate for these workloads. Both the reported root cause DNs were helpful in debugging HBase.

Further, we identified a problem with HBase's TCP networking code which affected latencies of requests, but we do not discuss it here for brevity.

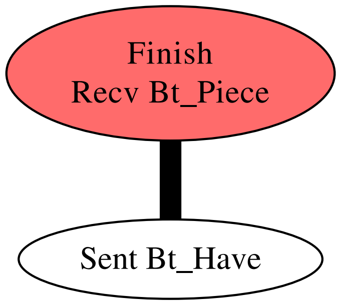
2.4.3 Transmission

Transmission implements the BitTorrent protocol, a distributed file sharing mechanism that downloads different pieces of a file from multiple peers. The protocol works by requesting a set of active peers for the file from a *tracker*, then directly requests file pieces for download from them. By downloading from multiple peers simultaneously, clients can more easily download at large speeds limited only by its bandwidth. Azureus is another BitTorrent implementation, that we used for comparison. In some basic experiments, Transmission had a much worse download time compared to Azureus (552 sec vs. 288 sec).

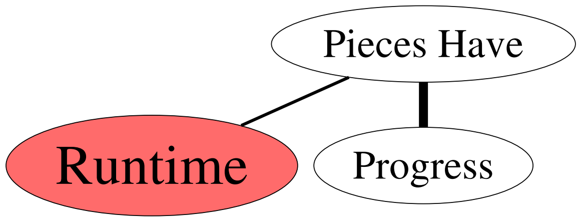
Transmission [27] is a light-weight C implementation, and among all the free clients, it is known for its minimal resource footprint. Azureus [45] is one of the most popular free implementations of the protocol, developed in Java. It is an older and more mature implementation of the protocol and well known for its excellent performance. Unlike Transmission, it extends the basic BitTorrent messaging protocol for extra minor optimizations in communicating with supporting peers. Both are serious implementations of the protocol, and we expect a well tuned C implementation should perform no worse than a Java implementation. Using DISTALYZER, we were able to identify two performance bugs in Transmission that eliminated the download time difference completely.

Experimental setup Experiments consisted of 180 BitTorrent clients (30 clients per machine) attempting to download a 50MB file, providing ample interaction complexity in the system. They used the same machines as described in Sec. 2.4.2. The swarm was bootstrapped with a single seeder, and each client was limited to an upload bandwidth of 250KB/s which is similar to common Internet bandwidths and makes ample room for running 30 clients on a single machine. Experiments were conducted with each implementation in isolation.

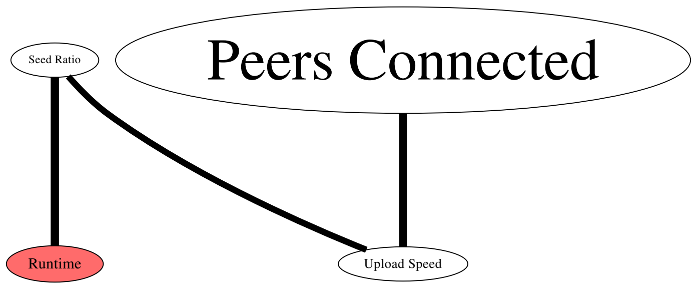
We built Azureus from its repository at rev. 25602 (v4504). Azureus had a detailed log of BitTorrent protocol messages during a download, and we added some state logs. The experiments used the HotSpot Server JVM build 1.6.0_20. We used version 2.03 of Transmission in our experiments, which contained debugging logs, and we simply activated the ones pertaining to the BitTorrent protocol. We identified the event and state performance metrics Finish and Runtime, respectively.



(a) Event DN



(b) State DN: 1st with score 0.89



(c) State DN: 2nd with score 0.84

Figure 2.8.: Dependency graphs for unmodified Transmission

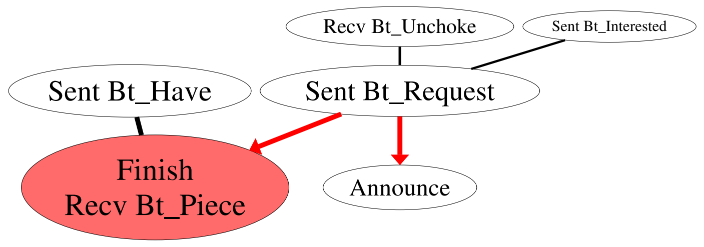
Faulty component affecting performance

The best DNs output by DISTALYZER for both event and state shown in Fig. 2.8a and Fig. 2.8b, were dependencies between trivial divergences. These are in a sense false positives to the automatic root cause detection. More specifically, Fig. 2.8a was picked from the *Last* event-feature and shows the performance metric coalesced with the last piece receipt. The strong dependency to Sent_Bt_Have is justified by the fact that implementations send out piece advertisements to peers, as soon as they receive one more piece. Similarly, the state dependency graph in Fig. 2.8b shows strong dependencies between download completion time and the number of pieces download in half the run, and also the progress (which is in fact a factor of Pieces Have). We discard these DNs and move to lesser ranks.

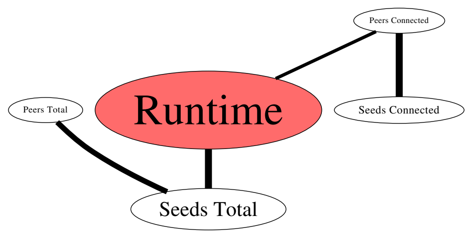
This led to considering the second ranked state graph in Fig. 2.8c, which in fact had a very close score to the highest rank. This DN was constructed from snapshots of the state variables at three-fourth of Transmission's runtime. Runtime is connected to divergent Peers Connected through a chain of variables. The chain involves the amount of data seeded and upload speed, both affirming the symbiotic nature of BitTorrent. This immediately takes us to the distributions of the number of peers, where we noticed that all nodes reported 6 peers in Transmission, as against 50 for Azureus. We also verified these values for the *Maximum* feature.

Fixing the bug To find the problem that limited Transmission's peer connectivity, we considered a single node's logs and fetched the set of unique IP:port pairs, and on looking at the values, we immediately realized that each peer had a different IP address. In our experimental setup with 6 physical machines, different nodes on the same physical machine were setup to listen on different ports and coexist peacefully. The bug was traced to the internal set that holds peers, whose comparison function completely ignored port numbers. When a node obtains a new peer from the tracker, and it is already connected to a peer with the same IP address, it is simply dropped.

On looking through forums and bug management software, we found that this inconsistency had actually been identified 13 months back, but the bug was incorrectly closed. We verified the authenticity of this bug and reopened it. The developers deemed this bug to be hard to fix, in terms of requiring changes to many modules. We argue that this is an important bug that limits Transmission from connecting to multiple peers behind a NAT box. In cases where multiple peers are situated behind a NAT box in an ISP, they would definitely want to download from each other and avoid the slow ISP link. This bug would prevent local connections, thus forcing them to connect to peers on the Internet.



(a) Event



(b) State

Figure 2.9.: Dependency graphs for BitTorrent after fixing the NAT problem

Tuning the performance

Since the fix for the first bug was too tedious, we decided to circumvent the problem by assigning unique virtual IP addresses to each of the nodes. This did indeed solve the problem and made Transmission faster to an average download time of 342 sec, which was still much higher than 288 sec. DISTALYZER was used again with the new set of logs which produced the dependency graph output shown in Fig. 2.9. Considering the event DN in Fig. 2.9a, showing the highly divergent performance metric for the *Last* feature. Some of the features of this DN are similar to Fig. 2.8a that were discussed earlier.

The dependency between finishing and sending requests fits well with the protocol specifications, that a request for a piece must be sent in order to receive one. The Announce event happens *after* sending out requests, and hence de-values its possibility for root cause. The interested messages were a more probable cause of the differences (compared to un-choke) because one must first express interest in another peer after connection establishment. Only after this step does the remote peer un-choke it, thus opening up the connection to piece requests. This hypothesis was verified by viewing the distributions of Sent_Bt_Interested across all features. After knowing the root cause, the distribution for the offending variable in the *First* feature showed gaps of the order of 10 sec on Transmission, but was very small for Azureus.

We traced the code from the message generator to fix these large gaps, and found a timer (called rechokeTimer) that fired every 10 sec. For comparison, we found that

Azureus had a similar timer set at 1 sec, thus giving it a quicker download start. The large divergence in sending interested messages could be fixed by shortening the timer value from 10sec to 1sec. Fig. 2.9b shows the state DN for the same logs for completeness, but it does not indicate a highly divergent root cause.

Performance gains We were able to apply a quick fix for this problem and the download times of Transmission were much better than earlier, dropping the mean completion time to 288 sec. The performance was up to 45% better than the first experiment. It should be noted that the more frequent timer did not affect the resource utilization of Transmission, still using far fewer CPU cycles and memory than Azureus. Neither of these issues affected correctness, nor threw any sort of exceptions, and present themselves as subtle challenges to the developers. Overall, 5 DNs were reported for the two issues in Transmission, out of which 3 indicated trivial relationships between the components, but the other two were immensely helpful in understanding the root causes.

2.5 Related Work

One of the underpinnings of systems software development is model checking. Model checking aims to provide guarantees on program code against pre-specified properties. A number of techniques [1–3] have described different methods to assert program correctness. However, traditional model checking attempts to discover violations of clear failure conditions, which is convenient for correctness problems. There is also research in applying machine learning to logs of faulty executions, to categorize them [11, 48] and also predict the root cause [10]. Conditions of performance degradation cannot be accurately modeled using these approaches, because it is rarely possible to specify performance as definite runtime predicates.

The formulation of debugging as an anomaly detection task has been applied in a variety of contexts, with the hypothesis that bad behavior is occasional and diverges as an anomaly from normal behavior. Magpie [11] and Pinpoint [10] model request paths in the system to cluster performance behaviors, and identify root causes of failures and anomalous performance. Fu et al. [26] propose the use of a Finite State Automaton to learn the structure of a normal execution, and use it to detect anomalies in performance of new input log files. Xu et al. [4] propose a mechanism to encode logs into state ratio vectors and message count vectors, and apply Principal Component Analysis to identify anomalous patterns within an execution. However, they completely ignore timestamps in logs and use the value logged, to identify localized problems within a single log file. On the other hand, DISTALYZER finds the root cause of the most significant performance problem that affects the *overall* performance. In contrast to all these systems, DISTALYZER aims to find the cause of performance problems in a major portion of the log instances, and hence uses t-tests to compare the average performance.

Request flows are a specific type of distributed processing, with a pre-defined set of execution path *events* in the system. Spectroscope [20] aims to find structural and performance anomalies in request flows that are induced by code changes, by analyzing instrumented latencies on request paths. Their approach of comparing different requests bears some similarity to our technique. X-ray [49] also tracks the paths of possible root causes to observed performance degradations. It intelligently instruments application binaries through dynamic instrumentation at runtime, and computes the likelihood of specific root causes impacting measured performance metrics. Similar to Spectroscope, X-ray performs performs performance differentiation to identify faulty components on code paths. However, as illustrated through the case studies, DISTALYZER can be applied to request flow systems (HBase, § 2.4.2), as well as other types of distributed systems, by abstracting the logs into states and events. Although these specific applications of machine learning (including [9–11,14]) can leverage path structures, DISTALYZER can show the *most impacting* root cause among many performance problems.

Cohen et al. [9] use instrumentation data from servers to correlate bad performance and resource usage using tree-augmented Bayesian networks. Similarly, DISTALYZER can utilize system monitoring data as outlined in Section 2.1 to identify performance slowdowns due to resource contention using DNs. NetMedic [24] and Giza [25] use machine learning to construct dependency graphs of networked components, to diagnose faults and performance problems. WISE [50] uses network packet statistics to predict changes to CDN response times on configuration changes, using causal Bayesian networks. In contrast, the use of distributed system logs allows DISTALYZER to identify software bugs by marking specific components in the code. Our novel use of dependency networks to learn associations between code components alleviates the need for an expert developer.

Splunk [21] is an enterprise software for monitoring and analyzing system logs, with an impressive feature set. Although it provides a good visual interface for manually scanning through logs and finding patterns, it does not provide tools for rich statistical analysis on the data. Furthermore, there is no support for comparing two sets of logs automatically. We believe that Splunk is complementary to our work, and the concepts embodied in DISTALYZER could serve as a great addition to Splunk.

2.6 Practical Implications

While DISTALYZER has proven to be useful at finding issues in real systems implementations, we now discuss some of the practical implications of our approach, to illustrate when it is a good fit for use.

First, DISTALYZER is based on comparing many log instances using statistical approaches. To be effective, there must exist enough samples of a particular behavior for the tool to determine that a behavior is not just a statistical anomaly. The use of weights is a partial solution to this problem. Similarly, however, the tool cannot find problems which are not exercised by the logs at all, either originating from an external black box component or insufficient logging within the system. In the former case, there is hope that existing logs would capture artifacts of the external problem and hence point to that component. The ideal approach would be combining logs from the external component or network with the existing logs, to paint the complete picture. With insufficient logging, DISTALYZER would fail to find feature(s) that describe the performance problem. This can be alleviated with additional instrumentation followed by iterative use of DISTALYZER to diagnose the issue.

Second, we assume similar execution environments for generating the logs, leaving situations of differing machine architectures, network setups or node count in obscurity. This is a tricky process because a subset of features can be dependent on the environment, and hence their divergence would be trivial leading to futile DNs. As a counter measure, these features can either be removed or transformed into a comparable form with domain knowledge. The specific case of *relative* times for event features highlights such a transformation. In future work, we imagine support for a mapping technique provided by the user for converting the features into comparable forms, allowing DISTALYZER to be used even to compare different environments.

Finally, the system inherently requires log data. If it is impractical to collect logs, either due to the overhead imposed or the manual effort required to instrument uninstrumented systems, our tool will not be a good choice. Similarly, it is important when using DISTALYZER to verify that the user-provided classifying distribution is not adversely affected by the instrumentation. Indeed, one "problem" we tracked down using DISTALYZER identified that some poor performance was actually caused by the system's logging infrastructure flushing to disk after every log call. This is observed by seeing performance variations with and without logging.

2.7 Summary

This chapter proposed a technique for comparing distributed systems logs with the aim of diagnosing performance problems. By abstracting simple structure from the logs, the machine learning techniques described here can analyze the behavior of poorly performing logs, as divergence from a given baseline. We design and implement DISTALYZER, which can consume log files from multiple nodes, implementations, runs and requests and visually output the most significant root cause of the performance variation. The analysis of three mature and popular distributed systems demonstrates the generality, utility, and significance of the tool, and the reality that even mature systems can have undiagnosed performance issues that impact the overhead, cost, or health of these systems. Distalyzer can help to find and solve these problems when manual analysis is unsuccessful.

3 TRACKING PERFORMANCE CHANGES OF SYSTEMS IN CODE REPOSITORIES

Many software projects consider performance as a vital component of development. While many tools [48,51,52] exist to help developers optimize and profile the resource utilization of the code they are currently running, there are fewer tools available to help developers understand how code changes will impact the overall performance of the system in the real world.

In ongoing development, continuous integration tests (e.g., unit tests) are run after nightly builds to monitor correctness and ensure that local code changes do not have long-range unintended consequences [6, 53]. However, unlike correctness and functionality, performance is much more difficult to evaluate and assess.

To address this, developers typically create multiple performance benchmarks that exercise various components of the code, and can be run on specific revisions in the repository to measure and evaluate performance. One or more of these benchmarks may be affected due to changes in the code, and any significant changes in the behavior of the benchmarks need to be brought to the attention of the developers. Identifying such changes is beneficial to developers in two ways: (i) *expected changes* in performance can be confirmed with domain knowledge expectations of the code change; (ii) *unexpected changes* can be investigated and diagnosed early, to avoid the persistence of performance bugs.

The primary goal of a performance management framework is to identify important changes in the performance metrics, so that a developer may investigate its cause. Previous work in the context of monitoring systems performance [9, 54] has assumed unchanging systems with varying workload, and use thresholds to identify changes. To our best effort, we explored the performance and software engineering literature in the past 10 years, but surprisingly found little work on systems to automatically monitor the performance of changing software. Performance modeling of complex software systems has been explored in the context of predicting performance on configuration changes or significantly different workloads [55–57], but such models cannot be easily applied to changing software code.

The current *state-of-the-practice* for automatically flagging significant performance changes use variants of Service Level Agreements (SLAs) [9, 58], thresholds [59] or moving windows [54]. Each of these require significant developer assistance and configuration. These techniques are tedious and error-prone for changing software applications for the following reasons:

- Requires the specification of SLA bounds using domain knowledge for each metric,
- Requires developers to choose parameter settings such as thresholds or window sizes,
- Performs inaccurately unless *carefully* tuned by the developer (§ 3.3.1).

These factors severely limit the scope of these techniques for use in performance testing of code repositories. To address this, in this dissertation we develop an automated tool that can help developers detect when changes in the code produce an important impact on performance (Section 3.1).

To begin, we conducted a performance measurement study of three popular software projects over years of their commits, all of which highlight the existence of abrupt performance changes. The performance of Transmission BitTorrent (Fig. 3.1a), Google Chrome V8 JavaScript engine (Fig. 3.1b) and Hadoop (not shown here due to lack of space) have their own unique characteristics. Each benchmark was executed multiple times to obtain these measurements. From this study, we derived the key insight that application performance behavior can be characterized as periods of predictable performance, punctuated by significant code changes and anomalies. We believe that this natural model of performance is not properly accounted for, in *state-of-the-practice* techniques, thus being the primary reason for their low accuracy.

Our analysis shows evidence of many possible performance problems over the period of study. We discovered cases where commits trigger changes in both the *average* and *variance* of the performance of the executions. Interestingly, some instances of bad performance changes have persisted for many weeks, showing evidence of unidentified performance issues. The impact of these unidentified, and uncorrected, performance problems should not be overlooked. For example, a slow JavaScript engine (V8) in Chrome could seriously impede the rendering and responsiveness of pages, impacting a substantial userbase [60]. Furthermore, these plots do not show the features other performance metrics that these benchmarks yield including software-specific metrics and machine resource metrics (CPU, memory, etc.), all of which are likely to change independently.

Since we identified many instances of code changes that have visible impact on different metrics, but sometimes without an effect on the main metric itself, we conjecture that important changes could be more easily detected by looking at all relevant changes in unison. However, due to the large number of metrics and complex relationships between them, it would be difficult to manually identify correlated changes across metrics on a *daily* basis.

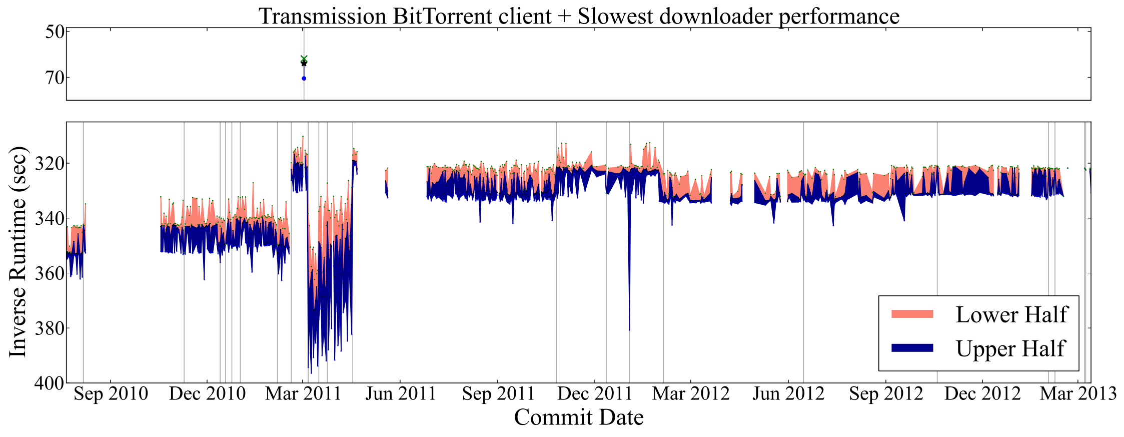
To address this, we aim to develop a system which does not only consider whether performance increases or decreases, but instead detects when the performance of a system is abnormally (i.e., significantly) different from expectations in one or more of several dimensions, be it average performance, performance stability, or the correlation of performance to other measures. The contributions of this dissertation are:

- A measurement study of how software performance changes over time in three popular software repositories – Transmission, V8 JavaScript engine and Hadoop MapReduce in Sec 3.2.
- The design of PerfDetect, a framework for managing daily performance and detecting code changes that cause significant changes in one or more performance metrics, in Sec 3.1.

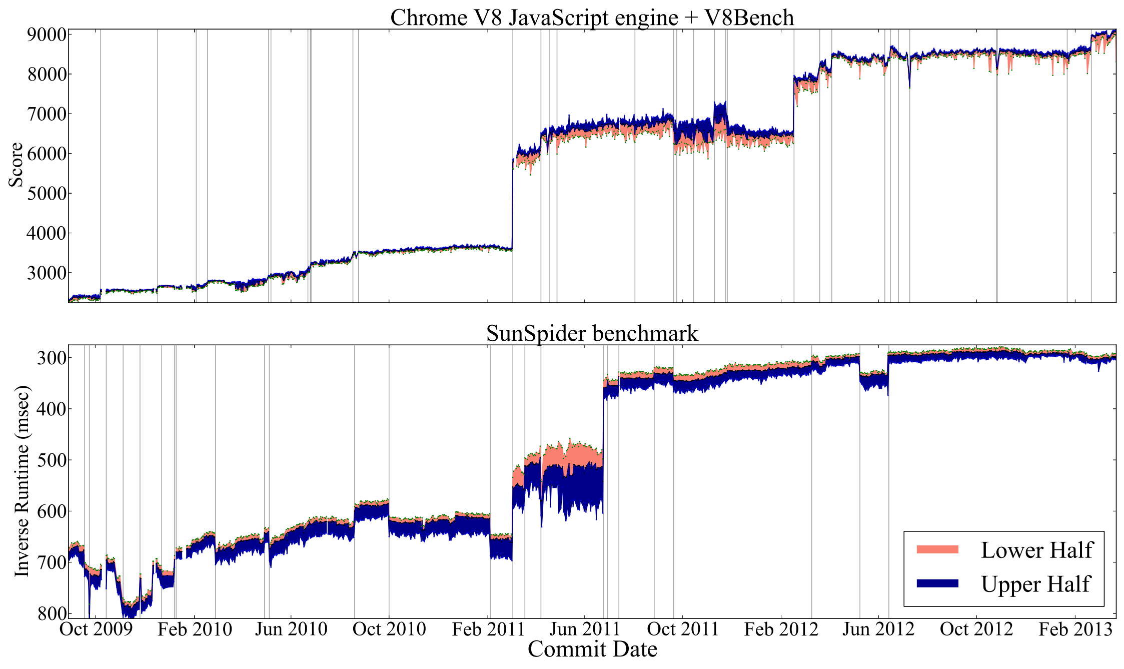
- A quantitative evaluation of PerfDetect against state-of-the-practice techniques, together with experiences in diagnosing these performance changes in Sec 3.3. We find that threshold-based techniques are very sensitive to the choice of the threshold, that even a ±0.6% change from the "ideal" threshold can cause upto a 100% increase in false positives, or a 20% reduction in true positives. When compared to the only publicly available implementation of a detection scheme, used by Mozilla Firefox [61], PerfDetect detects an additional 10–125% true changes, and reduces misdetections as much as 40–70%.

3.1 Designing PerfDetect

This section presents the design of PerfDetect, a framework for identifying changes in the performance characteristics of code repositories, that concurrently manages the execution of daily performance benchmarks and monitors divergence of metrics from their historical trend. Fig. 3.2 shows the different stages of PerfDetect starting with the execution of daily performance benchmarks on code revisions, and identifying revisions that require developer investigation on unexpected performance. At its input are the code repository coupled with a build system, and the set of performance benchmarks developed and configured for the repository. The developer also configures the framework for nightly testing with the choice of night builds, most commonly chosen as the last commit nightly (*e.g.* 4 AM). In our evaluation of the three code repositories, we empirically identified their respective night times as the time of day with least commits (Sec. 3.3). The developer also indicates the performance metrics of interest collected during benchmark execution, including a primary performance metric P (*e.g.* latency), other benchmark-specific metrics and resource usage metrics. Examples of benchmark-specific metrics include scores of individual sub-benchmarks, throughput, goodput, etc. Each execution of a benchmark produces a single value of each of these metrics.



(a) Performance trend of Transmission revisions on a file download benchmark on 100 nodes. Shaded regions indicate the upper and lower half of the execution distributions. Gray lines show *true* changes as identified by the authors.



(b) Performance scores of Chrome V8 JavaScript engine under the V8Bench and SunSpider benchmarks. Although using the same code, the benchmarks are quite contrasting. Gray lines show *true* changes across benchmarks as identified by the authors.

Figure 3.1.: Performance measurement study of Transmission and V8 software repositories

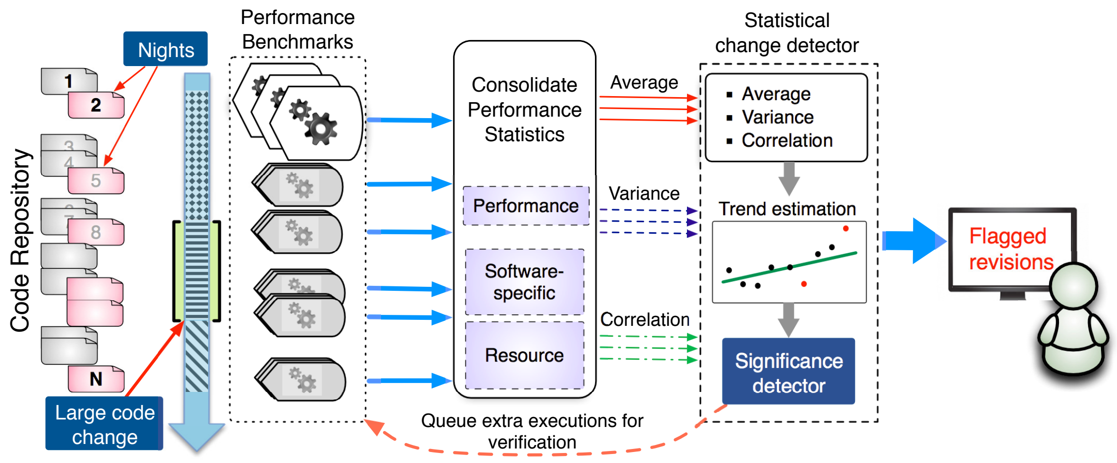


Figure 3.2.: Design of PERFDETECT for detecting deviations of performance metrics from their expected behavior

PERFDETECT uses statistical techniques to identify if today's benchmark metrics exhibits changes in mean, variability or correlation, that are considered unexpected deviations from the current trend. The components that make up PERFDETECT are:

- **Mean, Variance and Correlation**: Each performance metric is assessed for changes in these dimensions, to gather not just the average behavior, but the *variability* of the metric, and association with the primary performance metric.
- **Large code changes**: To improve the accuracy of change detection against large variance, we argue for the demarcations of specific revisions as having significant impact on future performance characteristics of the system.
- **Detecting changes**: By estimating the trend of a performance metric since the last large code change, this flags revisions as having significant impact on one or more performance metrics.
- **Queue extra experiments**: If a day is identified to have caused a significant change, it is crucial to identify if the executions are exhibiting a real shift in performance or simply large variability. Therefore, extra benchmark executions are automatically scheduled to improve the accuracy of the prediction.

We evaluate the output of these flagged days in Section 3.3 by comparing them with other known techniques, and describing the issues we diagnosed as a result. Each of these components is described in greater detail below.

3.1.1 Performance Metrics Features

Performance metrics typically often vary from execution to execution, because of non-determinism that exists both within and outside the software. Therefore, a single benchmark execution of a day's code snapshot is insufficient for characterizing output behavior. PerfDetect summarizes the scores from multiple executions in the following dimensions:

Mean. The mean performance is the mean of the performance values from all executions of a given day. This aims to capture changes in the overall performance of the software, and is also the most frequently used metric to track performance. For instance, the *mean* runtime performance of Transmission improved from 350 sec to 330 sec in Feb 2011 shifting the whole distribution, with the 20 sec difference having a magnitude larger than nearby noise (Fig. 3.1a).

Variance. Variance is natural to systems and measurements of performance, primarily attributed to randomness in code, timing measurements, scheduling, etc. and developers strive to reduce the variability of their systems [62,63]. Even small changes in the code can have significant impact on the variability, as we observe in our measurement. Surprisingly, we identified instances of increase in variance going unnoticed for many weeks, suggesting that developers easily miss variability although it is well appreciated to be important. For example, Transmission exhibited high variance for a month in Mar 2011 (Fig. 3.1a) that went undetected§ 3.3.3. PerfDetect considers the variance of all performance metrics for estimation of changes. Estimating variance accurately requires a large number of samples, and it may not be feasible for all benchmarks due to limited time. Sec. 3.1.4 discusses methods for improving accuracy.

Correlation Analysis. As the performance changes over time, other performance and resource metrics are expected to change synchronously. These can be metrics such as CPU usage, memory usage, or scores of individual benchmarks that constitute a benchmark suite. For instance both V8Bench and SunSpider benchmark suites aggregate scores from many sub-benchmarks. These associated metrics share some characteristics with performance metrics such as variability, unpredictability and often change together with the software in its lifetime. For the purposes of diagnosis, a naïve detection technique would be identifying correlated changes in other metrics when performance changes. Today, this can be easily achieved using standard correlation techniques. However, we quickly realized that metrics are very often correlated as one would expect, leading to very little new knowledge about the revisions. On the contrary, PerfDetect searches for *changes to the correlation behavior*.

For instance, it is trivial to infer that faster download times are correlated with larger bandwidth usage on the machines. However, it is much more useful to know that *increased correlation* between UDP throughput and performance happens on the *same day* performance got worse (§ 3.3.3). We use a standard Pearson correlation coefficient of the performance metric and the secondary metric. This value ranges from -1 to $+1$, indicating strong negative and strong positive correlations respectively. A value of 0 indicates no correlation. A small caveat to correlating with scores from sub-benchmarks for both V8Bench and SunSpider is that the total score is *necessarily correlated* with the sub-benchmark scores, as the total score is a mathematic function of the individual scores. To remove this dependence before correlation, we correlate a sub-benchmark score against a new aggregate of the *other scores*. Sec. 3.3.4 describes our experiences in using the correlation predictor to identify the cause of a performance degradation to a single sub-benchmark.

The mean, variance and correlation of all performance metrics are independently analyzed for significant changes. In identifying a technique for estimating change, it is intuitive to see that *recent trends* in the performance are more representative of the near future, than performance values from (say) last year. This is because performance trends in code repositories do not follow any periodicity. Therefore, an early prototype of PerfDetect used an exponential moving average (EMA) and variance (EMV) of a performance statistic since beginning of time. With this technique, we were actually able to successfully detect many of the significant performance changes in all the benchmarks, but were consistently missing changes following a *large change*. Right after a large change in the performance metric, the EMA and EMV consume a few days warming up to the new performance values, and hence temporarily estimates large variability. This causes it to miss relatively smaller changes that would otherwise be considered significant.

3.1.2 Windows of Large Software Changes

By taking the position of a developer of these systems, we realized that all large performance changes were the result of code changes that were well intended. Hence the repercussions in the form of performance improvement or degradation, were *expected*. Deriving from this observation, we posit that software development can be characterized as regions of predictable performance, punctuated by *large code changes* or *anomalies*. Large code changes are recognized by the developers as important changes to the software history, and alter one or more performance metrics considerably. Anomalies are commonly due to trivial flaws in the software or the experiment infrastructure, and do not contribute to the performance trend. This model of software performance behavior allows us to construct a realistic model of performance, whose scope is *bounded* by the large changes.

Determining a large change. Large code changes are not frequent in software repositories, and in our experience with analyzing the performance changes of Transmission and V8, we noticed only 4 large code changes in V8 and Transmission, and 2 in Hadoop, over years of commits. Revisions are manually tagged by the developer as a large change, but it is worth noting that these are easy to identify and are some of the largest changes in the software's lifetime. As elaborated in the evaluation(§ 3.3), PerfDetect manages to identify *all* large code changes as significantly divergences. Being non-developers of these systems, we were easily able to associate commit messages with observed performance trends to determine these changes.

3.1.3 Change Detection by Trend Estimation

Given a set of historical values X of a performance metric, the goal is to identify if today's value of the performance metric x_t is similar or divergent. To correctly model the trend in performance values, we use a *linear regression predictor* to predict divergences. This is achieved by fitting a trend line on the historical values, and estimating the distance of x_t from this line. The predictor outputs two values – a binary value indicating the significance of the new data against the history, and a score indicating the strength of the prediction.

Linear Regression Predictor After fitting a line to the historical performance values, the significance of today's value is determined by comparing today's divergence from the line to the historical divergence. The comparison is made using a normality test, that determines the probability that a value came from a given normal distribution, and in this case, the normal distribution of divergences. Given the history contains $t-1$ days of performance values $(x_1, x_2, \cdots, x_{t-1})$, this predictor is constructed out of the two dimensional data points $(1, x_1), (2, x_2), \cdots, (t, x_{t-1})$. The least squares function is used as the minimization function for fitting a line to these points. Once the line is computed, the error in today's value is simply the distance between the predicted value of $pred(X, t)$ and x_t (say $err(x_t)$). To estimate the significance of x_t, we compute the distribution of all $err(x_i); i < t$, and then compare this error distribution to $err(x_t)$. Using the distribution mean (μ) and variance (σ^2), the normality test is used to estimate the significance of the new value $err(x_t)$ using a confidence threshold of 0.05 or 5%. In other words, the new value is *insignificant* if it lies in the symmetric 95% window around the mean of the normal distribution, and *significant* otherwise. This choice of threshold is standard in statistics and well accepted.

The *strength* of the significance is determined using a standard one-sample student T-Test. The T-value is computed as:

$$t = (\mu - x)/(\sqrt{\frac{\sigma^2}{n}})$$

The strength of the significance is reported to the user, only if it has already been determined to be significant. It should be noted that the linear regression is recomputed on each day with the addition of every data point to the history, allowing the regressor to attempt finding a better line to fit the data on each day, *i.e.,* the trend changes daily. As hypothesized, we observed linear trends in performance metrics for all benchmarks, within the scope of large code change windows, but little evidence of such trends for the whole lifetime. This allows this predictor to have a high accuracy for predicting software performance trends. Moreover, we did not observe a non-linear trend in any of the metrics, and therefore higher order regression (polynomial of 2^{nd} degree or higher) was unnecessary.

3.1.4 Queue Extra Experiments

The physical resources available for running performance benchmarks is limited, because these benchmarks tend to be large and time consuming, and moreover need multiple executions for precision. For instance, our benchmarks for Transmission and Hadoop consume between 7 and 10 cluster minutes for a single execution(§ 3.3.3, 3.3.5), and it is practical to run only a handful of executions within a practical benchmarking quota of 1–2 hrs daily. With such a constrained environment, PerfDetect needs to efficiently use the given time across executions. Currently, we run each benchmark daily with a standard repetition (n_{std}) and use these performance metric samples to assess if the day's values are significantly different from the trend. If evidence of such changes are found, PerfDetect successively gathers more precision by queuing more execution repetitions of the benchmark (n_{extra}), and repeating the predictions.

To summarize, PerfDetect estimates the divergences in the mean, variance and correlation behaviors of the performance metric on every new data point. Positive predictions are reported to the developer for further investigation and diagnosis. We describe some of our experiences in using the predictions made by PerfDetect to determine real performance issues in these systems in Section 3.3.

3.2 Measurement & Implementation

We systematically collected performance data from each night's commits from years worth of code change in three software repositories – Transmission BitTorrent [27], Google Chrome V8 JavaScript engine [64] and Hadoop MapReduce [30]. These represent high-performance, popular systems implementations that are actively developed and maintained, with both distributed and non-distributed semantics. Table 3.1 summarizes the benchmarks and code activity for these repositories. Performance is an important concern for these systems and is frequently an objective of code changes, which is also apparent from the performance measurements (Fig. 3.1a, 3.1b). To the best of our knowledge, these systems do not have a systematic performance testing framework or haven't publicized their performance data. In this section, we describe each software repository briefly along with details on compilation and benchmarks execution. For all these software, the benchmarks were inter-operable with all revisions of the code selected for this measurement.

Experimental setup All our experiments were run on 8-core 2.33GHz Intel machines with 8GB of RAM, configured with Gentoo GNU/Linux 3.0.18 x86_64. One concern for a long term use of PerfDetect is the change in infrastructure at in-

Table 3.1: Summary of software and benchmarks used in our measurement and evaluation

Software	**Commits**	**Nights**	**Benchmark**	**Notes**
Transmission	2800	618	100-node 50MB torrent	Slowest node runtime with 250KBps upload cap, run on 5 machines.
Chrome V8	8617	937	V8Bench	8 sub-benchmarks: Crypto, EarleyBoyer, RayTrace, NavierStokes, etc.
			SunSpider	26 sub-benchmarks in 9 classes: 3D, Bitops, Crypto, Math, RegExp, String, etc.
Hadoop	3447	566	TeraSort	Sort 3GB/machine, on 10 machines
			Pi	Compute π to large precision, on 10 machines

termediate times. A possible workaround is to consider such changes as large code changes, thereby ignoring the performance of earlier revisions. A more intelligent workaround could execute a small number of historic revisions in the new infrastructure to estimate a small portion of the trend and improve accuracy.

3.2.1 Transmission

Transmission [27] is a client implementation of the BitTorrent protocol [65] being developed since 2005, and is the default BitTorrent client in the Fedora and Ubuntu Linux operating systems. The *trunk* branch had its lowest commit activity at 0900hrs UTC, and hence picked to be the time of the nightly testing.

Compilation. Transmission uses the GNU build system for compilation, and all our binaries were compiled for GNU/Linux and x86_64 target architecture. Specifically, the Command Line Interface (CLI) target is built, skipping all GUI related code. It should be noted that the choice of building and testing the CLI specifically limits the scope of bugs we find to this code. For instance, this would not catch performance issues that are resident in the GUI code. A library dependency issue was encountered for `libevent`, where the dependency changed from pre-v2.0 to v2.0 or higher in December 2010.

Benchmark. Transmission's code base is only bundled with a set of unit tests that verify correctness of a few components, but no performance tests. Therefore, we designed a simple benchmark that would exercise most of the core features, by evaluating the efficiency of downloads. The experiment consists of 100 peers downloading a 50MB file in a single torrent with a single seeder. All peers run the same binary, and are specified an upload bandwidth cap of 200KBps. These nodes are equally distributed on 5 physical machines connected through 3×1Gbps links, and we ensured that none of CPU, memory, network or disk was a bottleneck during the experiments.

Performance Metric. The primary performance metric is the time to completion of download by *all* nodes, *i.e.,* runtime of the slowest node. The ideal runtime for downloading this torrent file is 256sec (50MB on a 200KBps link), and the fastest observed runtime for Transmission was 309sec. Fig. 3.1a shows the performance of all revisions ranging between 60sec and 400sec. It is interesting to note that the average runtime dropped from 350sec to 330sec overall, with a 2 month region in the middle exhibiting high variance. Around Nov 2011, the 10sec runtime gap between executions almost completely vanishes.

3.2.2 V8

The V8 JavaScript engine [64] is an interpreter and runtime for the JavaScript language, designed and developed by Google using C++. It powers the Google Chrome web browser, that is used by over 34% of all Internet users [60]. We picked 937 night revisions since August 2009 for measurement, after empirically determining the night test time as 0200hrs UTC.

Compilation. V8 went through two different build systems in three years of commits, that we systematically automated. SCons was the build system used between Mar 2009 and Dec 2010, then Generate Your Projects (GYP) generator and standard GNU make files between Dec 2010 and Oct 2012. We used the 64-bit build target – *shell*, that takes in a single JavaScript program and executes it completely.

Sun Spider. The Sun Spider benchmark suite is one of the oldest and most popular JavaScript benchmark, comprising of 26 sub-benchmarks in 9 broad classes as indicated in Table 3.1. The *performance metric* for this benchmark is the elapsed time. The default behavior of this benchmark is to run all the benchmarks once for warmup (and discarding these timings), and running the benchmarks a second time for the true scores. A single run of the benchmark takes only 2 seconds, and therefore we run 150 executions nightly. Fig. 3.1b captures the overall performance of this benchmark, with more than a 2X improvement in 3 years. There are multiple large performance improvements such as the one in Jul 2011, with a general trend in better performance overall. There are also a number of deteriorations such as Jul 2012, and also a period of high variance in May 2011.

V8Bench. The V8 project also maintains its own performance benchmarks suite consisting of 8 sub-benchmarks (version 7), designed as macro experiments when compared to Sun Spider. Each sub-benchmark score represents the ratio between its pre-determined *ideal* runtime to the observed runtime, where higher values are better. Each benchmark is run at least 32 iterations for at least one second. The *performance metric* is the geometric mean of the individual benchmark scores, thus equally weighting all sub-benchmarks. Each run of the benchmark takes about 23sec on average on our machines, and we run about 8–10 runs each night. Fig. 3.1b shows the performance of this benchmark on the V8 code, with over a 4X improvement in performance on the same code. Similar to the other systems and benchmarks, there are periods of large performance improvements, high variance, low variance and gradual performance growth.

3.2.3 Hadoop

Hadoop Map-Reduce [30] is a popular framework for large scale data processing, that leverages distributed execution of `map` and `reduce` tasks. The framework comprises of the Hadoop Distributed Filesystem (HDFS), upon which the Map-Reduce framework is layered. Revisions from the *trunk* repository from 2009 and 2010 were picked for analysis. To the best of our knowledge, Hadoop does not have a principled approach to performance detection although it maintains a Jenkins continuous integration system. We measured performance of two benchmarks that come packaged with Hadoop and are considered relevant [66] – TeraSort which is the world record holder sorting benchmark, Pi which computes an approximation of π. All benchmarks were run on 10 machines, with 4 mappers and 4 reducers per machine. The *performance metric* for all Hadoop benchmarks is the total runtime, and we designed the experiments to run for approximately 10 minutes. Due to lack of space, we do not show the complete plots for these benchmarks.

TeraSort benchmark. We configured this benchmark to sort 3GB of data/machine using 100-byte rows. An input generator MapReduce program called TeraGen generates the input for TeraSort, and is run for each instance of the benchmark. The performance metric only accounts for sorting time.

Pi benchmark. We configured the precision of this benchmark so that it runs for 10 minutes (input $n = 10^{10}$). As described later, the runtime for this benchmark was double (20 min) for periods in early 2009.

3.3 Experiences

We applied PerfDetect to the historic performance data collected from the three systems that we measured, to both verify if the important changes are detected and also to understand the types of performance issues present in systems software. Firstly, we provide a quantitative evaluation of PerfDetect by comparing it with known and used methods to detect performance changes, namely thresholds, moving windows and exponential weighting as used by Mozilla Firefox. Next, we present experiences from diagnosing the performance issues that were flagged and describe what this measurement data means for systems developers.

3.3.1 Quantitative Evaluation

To develop a baseline for evaluation, we simulated the decision-making behavior of analysts and identified interesting behavior via manual inspection of the full time series. We have marked the revisions that were flagged as important using vertical lines in Fig. 3.1a and Fig. 3.1b for the reader. We use these points as "ground truth" to quantitatively evaluate the methods, but future work is needed to more carefully assess the impact of each revision on system performance. Section 3.3.2 reports on the bugs that we uncovered after delving deeper into several of the “anomalous” revisions flagged by PerfDetect.

For each comparison, we measure standard metrics including *True positives*, *False positives* and *False negatives* of the predictor as compared with the “truth”, as shown in Fig. 3.3. The quantitative evaluation of one statistic, mean performance, is presented across all *state-of-the-practice* techniques, due to lack of space. However, this evaluation is representative of change prediction that is performed on any other performance metric and statistic (variance, correlation, etc.).

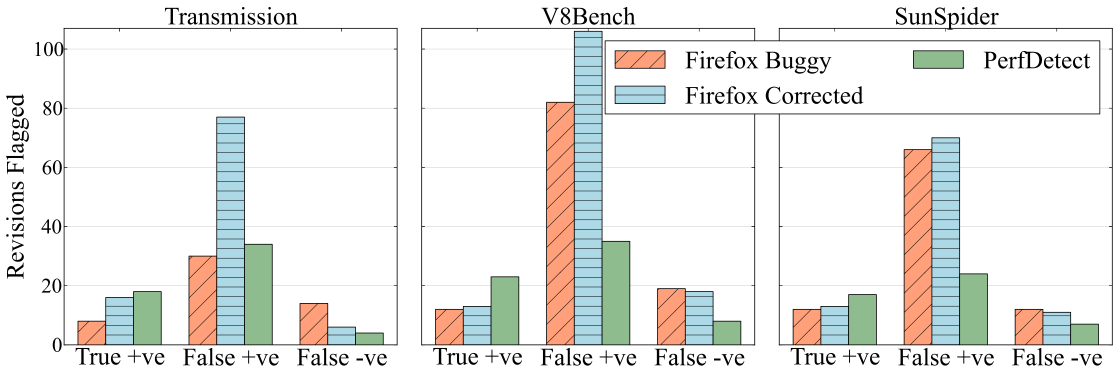
Comparison with Mozilla Firefox’s methods

Mozilla Firefox maintains a daily performance testing system – Talos, that uses an exponentially weighted moving distribution [61] to estimate the historical trend. Their method computes the exponential moving average (EMA) of the count, mean and variance of the performance to estimate the “exponentially-weighted distribution”, and then performs a T-Test to compare it with today’s values [67].

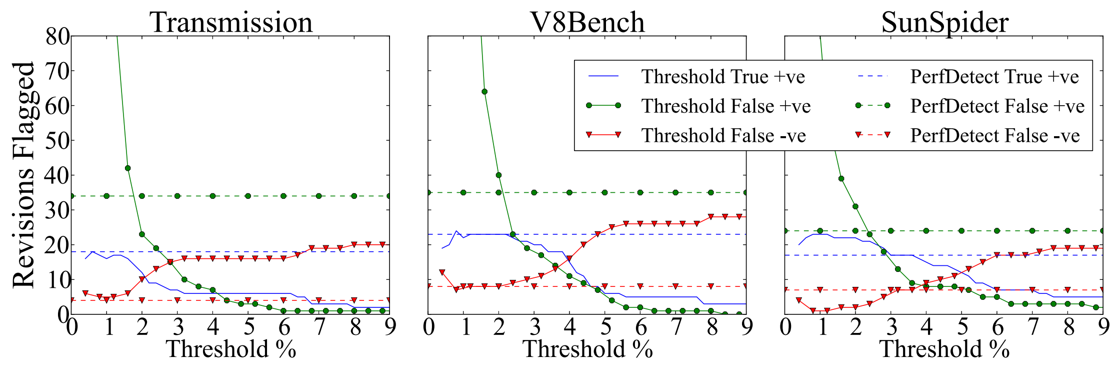
We have identified this to be an erroneous estimation of an exponentially weighted distribution, because $EMA(variance)$ is not the correct evaluation of the variance of an exponential distribution $var(ExpDistr)$ [68]. We fixed this computation using the presentation in [68]. Fig. 3.3a shows the evaluation of both the original (buggy) detection technique and our fixed version as compared with PerfDetect. It is clear from these graphs that PerfDetect performs better across the board with higher true positives and lower in both false positives and negatives.

Comparison with Thresholds

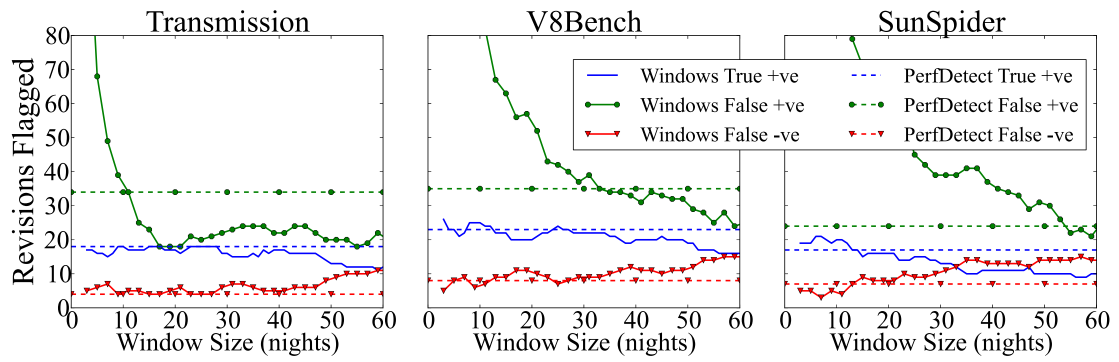
Thresholds are one of the most popular techniques for change estimation, and works by classifying the performance difference between two successive performance



(a) Firefox's exponential detection technique compared with PERFDETECT on prediction accuracy



(b) Different Thresholds applied on the previous day's performance compared with PERFDETECT on prediction accuracy



(c) Different moving window sizes compared with PERFDETECT on prediction accuracy

Figure 3.3.: Comparisons of prediction accuracy of PERFDETECT to other methods

values based on a pre-specified threshold. We originally expected the sensitive of the threshold value will be linear and 5% threshold would work reasonably. However, 5% threshold performed too poor and it led us to a finer grained evaluation. The comparison is shown in Fig. 3.3b where PerfDetect is independent of the threshold (horizontal lines).

As shown in Fig. 3.3b, at a very low threshold value, this technique is very sensitive and predicts almost all changes as significant resulting in high false positives, that grows exponentially. When the rates of false positives are equal (at about 2% threshold), the true positives starts dropping across all benchmarks.

Our observation shows the biggest drawback of threshold methods, which is the choice of the threshold is very sensitive to the accuracy. Moreover, the optimal threshold value depends on the benchmark, even when two benchmarks are for the same system. Thus, despite the importance, selecting a right threshold is a very challenging task. PerfDetect on the contrary is not based on a parameter input and achieves good accuracy with a reasonable number of false positives.

Comparison with Windows

We constructed the moving window predictor as follows: compare the distribution of performance values in the moving window to the new performance, to assess if it belongs to the same distribution. We used a normality test with a confidence interval of 5% similar to the significance technique used by PerfDetect (§ 3.1.3). The accuracy of this technique is shown in Fig. 3.3c.

Similar to what we observed in the threshold method, a lower window size produces a high true positive rate, yet with a huge false positive rate that grows exponentially. The ideal choice of window size would minimize false positives and maintain high truth, however this does not happen in unison for both V8Bench and SunSpider benchmarks, and can be identified for a small range of values for Transmission. Moving windows predicts Transmission's performance more effectively because of its rather flat performance trend (Fig. 3.1a). Moreover, the window sizes at which moving windows are equal the false positive rate of PerfDetect (intersection of the green lines) for three benchmarks are completely different. This indicates the downside of this scheme: while choice of an appropriate window size is critical for the prediction accuracy, selecting the right size of window is impossible or very challenging and it depends on the target system.

3.3.2 Practical Experiences

By tracking the performance of these systems using methodical automated techniques, we identified many instances of performance changes at the time of their inception and for problems, their eventual fix. We describe a few of these issues for each software system in detail, to highlight the common types of performance changes. Note that PerfDetect identifies the existence of a performance change in one or more performance metrics and dimensions (mean, variance, correlation), and we *manually* determined the root causes of those issues. The issues were most commonly identified by going through the commit logs and execution logs to identify the offending change. Some issues are persistent for prolonged periods, and in such cases we patched together commits from the initiation and termination to diagnose them.

3.3.3 Transmission

Transmission exhibited instances of all of mean, variance and correlation changes in our measurement time frame from July 2010 to March2013, of which we describe 3 in detail here. Note that Transmission has been quite stable in its overall performance, given that it has already been adopted by many users and OS distributions, and its release notes mainly indicated new features and associated bug fixes. Fig. 3.4 summarizes the issues that include a 10% average performance improvement, performance degradation in variance and eventual fix, and a reduction in runtime variance

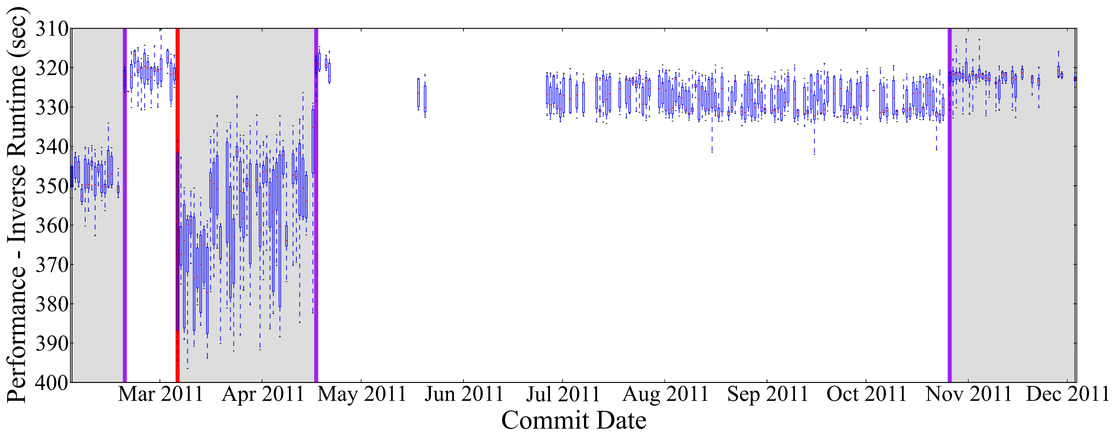


Figure 3.4.: Box-plot of Transmission file download performance indicating three performance issues. Shaded regions show the large code change windows. Other performance issues are not shown here for clarity.

of 10 sec. Tab. 3.2 briefly describes these performance issues, and we pick the most important to detail below.

Improvement to Piece Prioritization Policy

This is the first improvement in performance, that happens in February 2011 when the average runtime improves by 10% to around 320 sec in Fig. 3.4. Although we found it obvious that an improvement to the piece policy would improve performance in a sufficiently complex benchmark, we were surprised to find no mention of performance on the bug ticket. This change was predicted on the primary performance metric, and is indicated using a blue vertical line due to the direction of the change.

Prioritizing download on the rarest pieces is an important component of the BitTorrent protocol, and is the recommended technique by the specification – the standard does not force any piece download policy on implementations. By the virtue of the fact that the rare pieces are picked to be replicated across nodes, these pieces more easily available, thus increasing the number of available sources to download this piece. Transmission used a random piece policy until Nov 2010 when the lack of

Table 3.2: Descriptions of the important performance changes observed in Transmission

System	Prediction Type	Change	Description	Ref.
Transmission	Average, Window	10% improvement	Incorporated its first version of the rarest piece prioritization policy in Feb 2011, from its earlier scheme of random pieces.	§ 3.3.3
	Average, Anomaly	5X faster	Extreme performance improvement caused due to violation of specified upload bandwidth (March 2011).	
	Variance, Average, Window (2)	Large degradation and instability	For 40 days starting March 2011, instability caused by enabling new μTP wire protocol implementation to replace TCP. Root cause diagnosed to a faulty bandwidth allocator, and not in μTP code.	§ 3.3.3
	Broken *partially*	Assertion failures	Partial failure of nodes (83 among 99) in the middle of the execution, observed from late April 2011 to late June. Caused by a segmentation fault on a specific code path exercised by the nodes.	
	Variance, Window	Perennial 10 sec gap vanishes	In Nov 2011, a bimodal performance trend vanished almost completely, caused by a random generator bug failing to allocate equal bandwidth at a small granularity.	§ 3.3.3

this property was first reported by an external user, and was eventually incorporated into the code in Feb 2011. Due to the nature of this code change, this fix was later marked as the start of a new change window.

Lessons. It appeared from the bug ticket that performance was not a major criteria of this improvement. Firstly, the external reporter of this ticket does not mention a reason, and there was no reference of performance tests or improvement. More interestingly, on the same day that the final patch for this bug was committed, 70 other unrelated changes were committed. Within our measurement period this was the highest number of changes on a day, with the next highest falling short of 25. Such performance changes could happen at any day of development, and must be properly analyzed and investigated.

Enable μTP Wire Protocol Implementation

A new wire protocol called μTP meant to replace TCP as the default, was enabled in March 2011 causing a big performance degradation in both the mean and variance of the benchmark. This protocol constructs a *less aggressive* reliable in-order transport using UDP, meant to react to congestion sooner and back-off. In an uncongested network such as ours however, this is expected to provide throughput comparable to TCP and hence this change was not intended to drastically reduce performance. The goal of using μTP is to reduce the effect of BitTorrent traffic on ISP networks, by being *nice* to other TCP traffic.

Fig. 3.4 shows both the start and end of this degradation in March and April 2011 which lasted for 40 days. The mean performance worsened from 320 sec to 360 sec, and the variance spiked. It should be noted that PerfDetect also flagged changes in the UDP datagram and TCP packet volume, and further identified a large change in correlation between UDP and performance. These signals would quickly allow a developer to narrow down the space of performance metrics, to diagnose the problem. The actual root cause was a problem in the bandwidth allocator that was triggered by the switch to μTP. Moreover, the fix did not lie in newly written code for μTP but rather in a function that was last modified 2 years ago.

Lessons. Most of the implementation and testing of μTP was done earlier and simply enabled at this time. Since Transmission lacked a solid performance management system, such an important performance degradation was missed by the developers for a prolonged period. This bug clearly demonstrates repercussions of newly triggered executions that can cause older code to perform badly. Such problems are hard to predict and prevent during development due to the large number of components, but can be reactively identified and fixed.

Bi-Modal Performance Distribution

One characteristic of Transmission's performance graph was a fairly stable gap of 10-15 seconds between the slowest and fastest executions (across multiple executions) for most part of our measurements. This gap tapered off around November 2011 although not completely, with a few outlier executions as seen in the right end of Fig. 3.4. PerfDetect detected this change in the mean and variance of performance, but unfortunately none of the other performance metrics were flagged indicating the subtle nature of the change. The box-plots clearly show the disappearing bi-modality with the reduction in the inter-quartile range.

This improvement is important to Transmission's development as it removed a really old instability leading up to the beginning of our measurements. The root cause of the bi-modality was traced again to the bandwidth allocator, to the same function that was erroneous in Sec. 3.3.3. The bandwidth allocator was using a faulty random generator causing it to mis-allocate bandwidth across peers albeit at a very small granularity. In the overall execution of the system, this led to a predictable half of the executions to run longer.

Lessons. The bug ticket was reported externally and marked as minor, although we thought that it was a significant performance issue that ought to have been identified earlier. This shows that performance problems can go unfixed for *years*, and simply treated as cases of randomness or ignored because of their small magnitude.

3.3.4 V8

V8's performance had a tremendous gain since the beginning of the measurement period, as opposed to Transmission which had smaller and less frequent changes. Although both V8Bench and SunSpider benchmarks operate on the same V8 JavaScript engine code base, they exhibit very different performance characteristics (Fig. 3.1b), representing common challenges in measuring the performance of complex systems. Both benchmarks exhibited performance improvements and deteriorations, with some of the biggest changes being large performance boosts.

Here we describe two of the 6 performance changes (Table 3.3) that induce the most interesting performance fluctuations, as indicated by the vertical lines in Fig. 3.5. The whole performance history contains many more interesting behaviors that are also flagged by PerfDetect, but we have not diagnosed all of them.

Crankshaft JIT Compiler

In March 2011, both V8Bench and SunSpider saw one of the biggest performance improvements in V8, as a consequence of enabling a new component called Crankshaft in the JavaScript engine. This is shown in Fig. 3.5 as a large improvement in the score V8Bench from 3500 to 6000, and a smaller decrease in the runtime of SunSpider from 650 msec to 550 msec. Crankshaft is a compilation infrastructure for V8 and it brought significant improvements to compute-intensive JavaScript, by incorporating aggressive optimizations only on the most popular parts of the program (hot code). Note that JavaScript is an interpreted language and hence any compilation performed

Table 3.3: Descriptions of the important performance changes observed in Chrome V8 and Hadoop

System	Prediction Type	Change	Description	Ref.
Chrome V8	Average, Window	7% improvement	Dec 2009, fixed an overly-aggressive optimization earlier made in Nov 2009.	
	Average, Variance, Window	Improvement of 65% for V8Bench, 15% for SunSpider	March 2011, enabling Crankshaft – a new JIT compilation infrastructure. Significant performance improvements on compute-intensive JavaScript, with larger variability.	§ 3.3.4
	Average, Variance	36% improvement on SunSpider	Jun 2011 reducing overhead of optimizations. Real Crankshaft benefits for SunSpider.	
	Average, Correlation	13% slowdown	A performance problem caused by aggressive de-optimization, fixed by careful optimization. Appears in a single sub-benchmark, but affects the overall score.	§ 3.3.4
	Variance, Average, Window	20% improvement on V8Bench	Feb 2012, achieved by an aggressive optimization for arrays containing only small integers.	
	Variance, Average		In Nov 2011, successive improvement and deterioration was made due to changes in garbage collection.	
Hadoop	Average	10% improvement	Change inducing automated configuration of in-memory sort buffer, improving the TeraSort benchmark which exercises the shuffle phase significantly.	§ 3.3.5
	Average	2X improvement	The Pi benchmark implementation was updated to use a more precise computation of π, incidentally also improving its performance.	

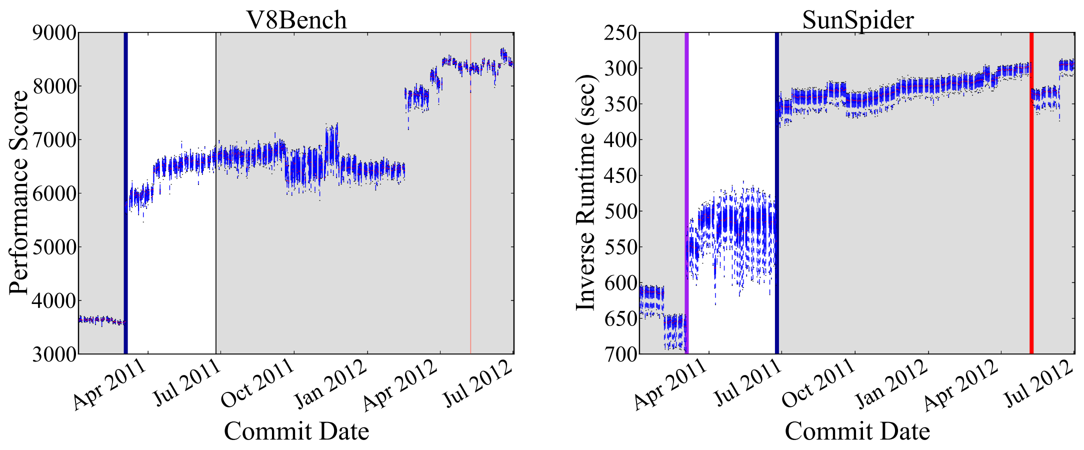


Figure 3.5.: Three performance problems identified in the V8 JavaScript engine across two benchmarks. Shaded regions show the large code change windows, common to all benchmarks sharing the codebase. Other performance issues are not shown here for clarity.

by the engine is Just-In-Time (JIT) compilation. By applying heavy optimizations only on the hot code, it avoids increasing runtime overheads due to compilation.

Crankshaft was enabled for the x64 platform in March 2011 which is our testing platform, although it appears to have been enabled earlier in December 2010 for the 32-bit builds. PerfDetect detected a change in both the mean and variance scores of V8Bench, and the box-plots clearly show the increase in variance. The instability in performance triggered by this change remains for a year until March 2012, before decreasing slightly. A similar variance trend was also caused in the SunSpider benchmark at this change, but this trend ends sooner in June 2011.

Lessons. Different benchmarks can have very different performance changes as a result of the same change in the code, and developers should carefully extract and analyze all such interesting changes. For instance, the V8 developers identified that the improvement to SunSpider is not as significant as in V8Bench and therefore needs other tweaks, that come later in June 2011. Also, a large magnitude of average performance change might easily mask secondary effects to the human eye, such as the variance in this case. In fact, we were unable to find any references to the high variance caused by Crankshaft in the commit logs, indicating a case of masked variance effects.

Misbehaving Sub-Benchmark: 3d-raytrace

This performance degradation is seen only in the SunSpider benchmark score in May 2012 as flagged with the red line in Fig. 3.5. The runtime of the benchmark increases from 300 msec to 350 msec. Along with the performance benchmarks, PerfDetect also predicted a change in one of the benchmark-specific scores of SunSpider – `3d-raytrace`. This performance degradation persists for a month in SunSpider. The computation of correlation factors out the 3d-raytrace component from the total score as described in Sec. 3.1.1. As a result of this change, the correlation score for 3d-raytrace starts a negative trend toward -1, *i.e.,* tending to negative correlation.

This bug was caused by overly aggressive de-optimization, with significant effect on the code exercised by this benchmark. As a result, this benchmark suffered the biggest penalty causing the sum of the runtimes of the sub-benchmarks to go up for SunSpider.

3.3.5 Hadoop

The performance of the Hadoop benchmarks were in general highly variable as compared to the other systems and benchmarks we measured. Fig. 3.6 shows a 10% performance improvement in the mean performance of the TeraSort benchmark as predicted by PerfDetect shown with box-plots. This change follows a performance commit made to automatically configure the sort buffer that is used during the *shuffle* phase. Prior to the commit, a configured fraction of the in-memory sort buffer was allocated for meta-data. The developers realized that such a hard-configuration was bad for workloads such as this sort benchmark which generates more meta-data per data, and added auto-configuration of this threshold. Naturally this caused an

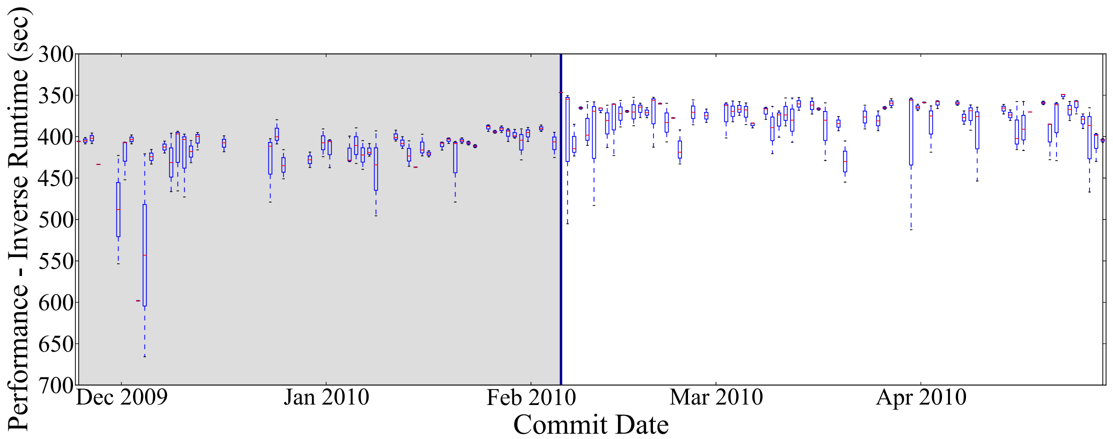


Figure 3.6.: Performance improvement on the Hadoop TeraSort benchmark

improvement in the MapReduce shuffle phase which is crucial for the sort benchmark, resulting in lower runtime.

Having described our experiences in designing and evaluating a principled approach to detecting performance changes in software repositories, we now turn to placing this in the context of published work for other similar problems in related areas.

3.4 Related Work

Performance detection is orthogonal to performance diagnosis, which is explored by Distalyzer (Chapter 2). Performance diagnosis usually follows the detection of a new problem.

Mathematical models of system behavior can be used to estimate performance characteristics. System modeling aims to model the performance behavior of complex software systems in a target environment, commonly when it is not easy or feasible to execute the system repeatedly. There exist a variety of system modeling techniques for target systems ranging from white to black-box modeling [55–57]. These modeling techniques have been proven effective for complex systems under uncertain environments, and could potentially be used to predict the impact of code changes. However, in the context of an automated test infrastructure, it is much less cumbersome to execute the target system for multiple executions than perform costly and uncertain performance modeling, and is the most commonly adopted approach [54, 61].

Mining software repositories (MSR) is a branch of software engineering that explores various aspects of software as it evolves in a revision control system (RCS) [69]. Apart from exploring the nature of the committed patches [70–72] and people who commit them [73, 74], it also explores choice of daily regression tests [5, 6]. We argue that the ability to manage performance measurements of systems collected regularly is orthogonal to the choice of the tests and infrastructure to facilitate their automatic execution. To the best of our knowledge, we conduct the first large scale study of the evolution of performance over years of development and thousands of code changes.

Many large software projects (including open-source ones) are known to perform daily tests to avoid performance regressions, but to the best of our effort we were able to find a publicly available system only at Mozilla Firefox [61]. Most *state-of-the-practice* techniques demand domain knowledge and careful configuration ability from the developers to make Service Level Agreements (SLAs) [9, 58], thresholds [59] or moving windows [54] work accurately, as clearly shown by the quantitative evaluation(§ 3.3.1). This places a huge burden on the developers, especially with the abundance of benchmarks and their associated performance metrics.

The automated correlation analysis of metrics data to performance of evolving software [75, 76] is perhaps the closest work to this PerfDetect w.r.t. correlations. Although they also apply machine learning techniques to find changes in correlation between the performance metric and other system metrics, it only computes correlation on discretized version of the input metrics. Moreover, the paper expects all the historical metrics to be generally *similar* to assess the general historical behavior. As seen for both Transmission and V8, the expected behavior of many metrics frequently changes with the system itself. Detecting significant changes from expected performance behavior has been explored in the context of software rejuvenation [77] with the goal of identifying changes *during* the execution of a system. This technique assumes the availability of a normal behavior as performance average and standard deviation, to identify specific kinds of deviations from this distribution based on a model of the input workload. We argue that our analysis is more powerful by being robust to non-discrete metrics and automatically estimates the *recent normal behavior* in history.

3.5 Summary

Regular performance measurements of software systems vary frequently with code changes, benchmarks, randomness in the environment, etc. Moreover large systems have many metrics and dimensions that must each verified with developer expectations, making performance tracking hard. Automatically tracking the performance of system benchmarks is essential for software whose performance is critical, so that performance deteriorations in the average, variance and correlations are identified early. This chapter motivates the need for automated performance management through a measurement study of three real systems over multiple years, and then builds PerfDetect that can automatically identify diverging performance behaviors. We develop on the insight that performance can be described as large code changes separating regions of predictable performance trends. The effectiveness of PerfDetect as compared with known techniques is demonstrated using both quantitative and qualitative evaluations.

4 SUMMARY

Software systems are becoming complex and harder to build and understand, especially in the context of parallel and distributed executions. With this shift in the complexity, it is getting harder for developers to manually manage the performance of their systems, hence inhibiting faster improvements to the software development life cycle. Performance issues often go undetected and pose as time consuming challenges to diagnose, due to unpredictable platforms, software randomness and the scale of executions. Many developer hours can be saved if such deteriorations can be detected at their early stages and fixed rapidly.

Machine learning techniques are useful to analyze the large volumes of rich runtime instrumentation data that are generated by these systems, to automate the management of performance. This thesis explored the application of machine learning techniques to aid software management in performance *detection* and *diagnosis* using widely available instrumentation logs.

4.1 Contributions

This thesis improves the state of performance management for developers by devising new machine learning techniques to detect and diagnose degradations in large scale systems software. We designed Distalyzer to diagnose performance problems using existing unstructured logs from systems software, to highlight the root cause of performance through event and state logs. As the precursor step toward diagnosis is the detection of new incoming problems during software development, we designed PerfDetect to automatically flag code changes that trigger non-trivial changes in one or more performance metrics. In developing these tools for developers, we have contributed to this field in many dimensions:

- **Semi-structured logs:** Most existing logs from large systems are unstructured yet rich with systems internal details, making them hard to use for automated analysis. Although there exist logging frameworks for generating highly structured logs, they are rarely used in practice. Inducing a small amount of structure through *State* and *Event* logs, allows for automated techniques to leverage the wealth of information.

- **Differentiate non-determinism from execution patterns:** Non-determinism is a major contributor to internal system event timings, as is intended program logic such as timers. During analysis, performing this distinction is necessary so that automated methods can search for true patterns in event signals as opposed to random event sequences. This can be resolved by requiring a sufficiently large sample set of event sequences (logs), and using robust statistical functions on observed values.

- **Graphical output of bug root causes:** As human developers are the consumers of our automated tools, it was necessary to develop outputs that are easy to comprehend and interpret as opposed to large tables of statistical computations. The use of visual graphs to represent component dependencies in DISTALYZER helped developers decipher the root cause, without previously having to know all dependencies. In addition, it required robust techniques for pruning the entities (or features) in the output, so that the developer is not overwhelmed by a bigger problem. Such visual output representations of machine learning analysis on systems logs would in general prove very useful to developers, who are commonly non-experts in statistics or data analysis.

- **Trends in software performance:** Identification of the trends in each performance metric delineated by large code changes was critical to the techniques of PERFDETECT. Each of the possible hundreds of performance metrics establishes its own trend over code commits, and it would help software managers to understand these trends.

This thesis has demonstrated that the common practice of debugging performance in systems through manual analysis is cumbersome and does not scale, and develops techniques for two common developer tasks. Our measurement study of three software repositories showed that daily performance characteristics are hard to track, especially in the presence of a large number of metrics, when these metrics are constantly impacted by different code commits. The insight into existence of metric trends, large code changes and anomalies led to the design of PerfDetect that estimates large metric divergences with high accuracy. Apart from the average trend in metrics, variance and correlation are shown to be helpful in detecting important changes to the software. On comparison with current practices in such detection that require heavy tuning, PerfDetect's detection scheme outperformed others with higher true positives and lesser false positives.

The natural successor to detection is diagnosis of the root cause, which is particularly hard for performance given the absence of clear error conditions, amidst large volumes of suspect log events. We identified that a common use case for developers is reasoning about bad performance against another set of executions with better performance, gathered from other implementations, versions, nodes or requests. We further recognized that existing logging in implementations contained enough information to perform a reasonable diagnosis task. This led to the design of Distalyzer to automate the comparison of semi-structured systems logs to identify a small subset of log events that best describes the performance problem. These graphical representations capture a balance between divergence in log features across the set of logs, and the statistical dependencies between possible root causes and the performance metric. This resulted in the successful diagnosis of six performance problems in three deployed software implementations.

The exploration of these problems in systems performance unraveled various other challenges that would help advance the state-of-the-art in automated management of systems. Next we identify a few closely related problems.

4.2 Future Work

In the era of big data, the growth of large systems is creating avenues for mining information from instrumentation logs. This research area is still at its infancy, and the work in this dissertation raises more problems and questions. Below, we list a few open problems.

Integration of software engineering with analysis techniques. Automated techniques for detection and diagnosis of bugs in software need good mechanisms and representations for output to human developers. Developers are most comfortable with the code and/or associated configuration parameters, and it would be ideal for automated techniques to associate directly to these, much like executable debuggers. Software engineering techniques such as static and dynamic program analysis, program tracing can enable automated tracing of program paths and interaction of program components. The combination of machine learning techniques that extract root cause chains with code analysis toolchains would be beneficial, by ideally outputting specific parts of the code corresponding to the root cause. For instance, the output dependency graphs of DISTALYZER (Chapter 2) refer to names of log instances found in the instrumentation. With an incorporation of software engineering that associates the locations in the code that generates the log statements, such software engineering techniques can scale much easily for diagnosis, and generate easier outputs for the developers. The scope of PERFDETECT is more restricted to changes in the code as a result of repository commits (Chapter 3). When performance changes are observed as an outcome of a specific changeset, they can also be associated with specific lines of change. The knowledge of the components affected by a changeset, together with a model for inter-component behavior can be used to diagnose new performance problems precisely. Additionally, repository maintainers can also choose to analyze software metrics such as code coverage, lines of change and other code metrics for change detection.

Diagnosis of performance problems caused by hardware faults. Hardware is equally likely compared to software, to fail or perform poorly. Faults are exacerbated in large scale deployments, where the mean-time-to-failure of individual components compound, resulting in higher probabilities of single component failure. Hardware problems are hard to detect, and developers frequently attempt to mask these in software as poor performance. In systems with large numbers of inter-dependent components, this can manifest as concealed performance problems in the overall or worst-case performance. Detecting root causes that exist in hardware is quite challenging when compared to software-induced problems, because the instrumentation only captures the software manifestation of the *true* problem. In our diagnosis of the hardware bug in TritonSort using Distalyzer (§ 2.4.1), we gathered the insight that hardware performance issues are caused by failure of a small number of components, as opposed to widespread deployments of buggy software. This creates challenges for the machine learning techniques such as the availability of a small number of samples, and misrepresented instrumentation data (*i.e.*, one or more log events display some postmortem evidence of the hardware problem). We believe that this space of hardware-induced performance problems is quite prevalent in large system establishments, and machine learning techniques can be specially tailored toward hardware nuances.

Analyzing gradual changes in component dependencies. Software undergoes many types of changes through continued development commits in the repository. Chapter 3 detects changes in the performance and related performance metrics due to code changes. Additionally, code changes trigger behavioral changes in inter-component dependencies. Systems have rich dependencies between components (Chapter 2) and understanding the nature and structure of dependencies is essential to understand the overall characteristics of the system behavior. Similar to changes in performance characteristics, gradual changes in dependency structures signify important behavioral changes. In complex systems with tens or hundreds of components and complex dependencies, developers struggle to manage such dependencies on a regular basis. We have explored the use of automated techniques such as dependency networks to learn these complex dependencies. The scope of these applications should be expanded to include analysis of dependency structures and their continuous evolution with code repositories.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[2] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation – Volume 5*, OSDI '02, Berkeley, CA, USA, December 2002. USENIX Association.

[3] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Detecting Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI '07, page 18, Berkeley, CA, USA, 2007. USENIX Association.

[4] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[5] Thomas Ball. On the Limit of Control Flow Analysis for Regression Test Selection. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 134–142, New York, NY, USA, 1998. ACM.

[6] John Bible, Gregg Rothermel, and David S. Rosenblum. A Comparative Study of Coarse- and Fine-Grained Safe Regression Test-Selection Techniques. *ACM Transactions Software Engineering Methodologies*, 10(2):149–183, April 2001.

[7] Latency is Everywhere and it Costs You Sales – How to Crush it. `http://goo.gl/35092y`.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[9] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation – Volume 6*, OSDI '04, page 16, Berkeley, CA, USA, 2004. USENIX Association.

[10] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation – Volume 1*, NSDI '04, page 23, Berkeley, CA, USA, 2004. USENIX Association.

[11] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation – Volume 6*, OSDI '04, page 18, Berkeley, CA, USA, 2004. USENIX Association.

[12] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting The Unexpected In Distributed Systems. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation – Volume 3*, NSDI '06, page 9, Berkeley, CA, USA, 2006. USENIX Association.

[13] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI '07, page 20, Berkeley, CA, USA, 2007. USENIX Association.

[14] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89, New York, NY, USA, 2003. ACM.

[15] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing Logging Practices in Open-Source Software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE '12, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.

[16] Apache Log4j. `http://logging.apache.org/log4j`.

[17] Sun. Solaris Dynamic Tracing Guide, 2009.

[18] Benjamin H. Sigelman, Luiz Andr Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[19] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Experience Mining Google's Production Console Logs. In *Proceedings of the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, SLAML '10, page 5, Berkeley, CA, USA, 2010. USENIX Association.

[20] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, page 4, Berkeley, CA, USA, 2011. USENIX Association.

[21] Splunk. `http://www.splunk.com/`.

[22] Elmer Garduno, Soila P. Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Theia: Visual Signatures for Problem Diagnosis in Large Hadoop Clusters. In *Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques*, LISA '12, pages 33–42, Berkeley, CA, USA, 2012. USENIX Association.

[23] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 795–806, Washington, DC, USA, 2010. IEEE Computer Society.

[24] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed Diagnosis in Enterprise Networks. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 243–254, New York, NY, USA, 2009. ACM.

[25] Ajay Anil Mahimkar, Zihui Ge, Aman Shaikh, Jia Wang, Jennifer Yates, Yin Zhang, and Qi Zhao. Towards Automated Performance Diagnosis in a Large IPTV Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 231–242, New York, NY, USA, 2009. ACM.

[26] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining*, ICDM '09, pages 149–158, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Transmission BitTorrent Client. `http://www.transmissionbt.com/`.

[28] HBase. `http://hbase.apache.org/`.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation – Volume 7*, OSDI '06, page 15, Berkeley, CA, USA, 2006. USENIX Association.

[30] Apache Hadoop Project. `http://hadoop.apache.org/`.

[31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[32] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '06, page 27, Berkeley, CA, USA, 2006. USENIX Association.

[33] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension For Distributed Replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI '07, page 21, Berkeley, CA, USA, 2007. USENIX Association.

[34] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM.

[35] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[36] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^{3}S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association.

[37] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

[38] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, page 3, Berkeley, CA, USA, 2011. USENIX Association.

[39] David Heckerman, David Maxwell Chickering, Christopher Meek, Robert Rounthwaite, and Carl Kadie. Dependency Networks For Inference, Collaborative Filtering, and Data Visualization. *Journal of Machine Learning Research*, pages 49–75, 2001.

[40] Charles Elkan. The Foundations of Cost-Sensitive Learning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence – Volume 2*, IJCAI '01, pages 973–978, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[41] B. L. Welch. The Generalization of Student's Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1-2):28–35, 1947.

[42] Olive J. Dunn. Multiple Comparisons Among Means. *Journal of the American Statistical Association*, 56(293):52–64, 1961.

[43] Janez Demšar, Blaž Zupan, Gregor Leban, and Tomaz Curk. Orange: From Experimental Machine Learning to Interactive Data Mining. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD '04, pages 537–539. Springer-Verlag New York, Inc., New York, NY, USA, 2004.

[44] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[45] Azureus BitTorrent Client. `http://azureus.sourceforge.net/`.

[46] Distalyzer download. `http://www.macesystems.org/distalyzer/`.

[47] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[48] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 105–118, New York, NY, USA, 2005. ACM.

[49] Mona Attariyan, Michael Chow, and Jason Flinn. X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[50] Mukarram Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. Answering What-If Deployment and Configuration Questions with WISE. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 99–110, New York, NY, USA, 2008. ACM.

[51] SAR: System Activity Report. `http://www.linuxmanpages.com/man1/sar.1.php`.

[52] Gprof. `http://sourceware.org/binutils/docs/gprof/`.

[53] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[54] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic Resource Allocation for Shared Data Centers using Online Measurements. *SIGMETRICS Performance Evaluation Review*, 31(1):300–301, June 2003.

[55] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter Nobel. Practical Performance Models for Complex, Popular Applications. In *Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 1–12, New York, NY, USA, 2010. ACM.

[56] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Alice X. Zheng, and Gregory R. Ganger. Modeling the Relative Fitness of Storage. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 37–48, New York, NY, USA, 2007. ACM.

[57] Eno Thereska and Gregory R. Ganger. IRONModel: Robust Performance Models in the Wild. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 253–264, New York, NY, USA, 2008. ACM.

[58] Jin Chen, G. Soundararajan, and C. Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing*, ICAC '06, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society.

[59] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A Cost-Aware Elasticity Provisioning System for the Cloud. In *Proceedings of the 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 559–570, Washington, DC, USA, 2011. IEEE Computer Society.

[60] Top 5 Browsers from April to September 2012. `http://gs.statcounter.com/#browser-ww-monthly-201204-201209`.

[61] Talos Statistical Analysis Writeup. `https://wiki.mozilla.org/Auto-tools/Projects/Signal_From_Noise`.

[62] Charlie Curtsinger and Emery D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 219–228, New York, NY, USA, 2013. ACM.

[63] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[64] V8 JavaScript Engine. `http://code.google.com/p/v8/`.

[65] The BitTorrent Protocol Specification, 2008. `http://www.bittorrent.org/beps/bep%2F0003.html`.

[66] A Benchmarking Case Study of Virtualized Hadoop Performance on VMware vSphere 5. White Paper, 2011.

[67] Firefox Datazilla. `https://github.com/mozilla/datazilla`.

[68] Tony Finch. Incremental Calculation of Weighted Mean and Variance. February 2009. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.147.5233`.

[69] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, March 2007.

[70] David Kawrykow and Martin P. Robillard. Non-Essential Changes in Version Histories. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 351–360, New York, NY, USA, 2011. ACM.

[71] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do Faster Releases Improve Software Quality? An Empirical Case Study of Mozilla Firefox. In *9th Working Conference on Mining Software Repositories*, MSR '12.

[72] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM.

[73] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning Bug Reports using a Vocabulary-Based Expertise Model of Developers. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 131–140, Washington, DC, USA, 2009. IEEE Computer Society.

[74] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring Developer Contribution from Software Repository Data. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 129–132, New York, NY, USA, 2008. ACM.

[75] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 32–41, Washington, DC, USA, 2010. IEEE Computer Society.

[76] Zhen Ming Jiang, A.E. Hassan, G. Hamann, and P. Flora. Automated Performance Analysis of Load Tests. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '09, pages 125 –134, Washington, DC, USA, September 2009. IEEE.

[77] Alberto Avritzer, Andre Bondi, Michael Grottke, Kishor S. Trivedi, and Elaine J. Weyuker. Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 435–444, Washington, DC, USA, 2006. IEEE Computer Society.

VITA

VITA

Karthik Swaminathan Nagaraj was born in Chennai, India where he did most of his schooling. He obtained his B.Tech. in computer science with distinction in 2008 from National Institute of Technology, Trichy, India. He continued in computer science to get his M.S. in May 2011 and PhD in December 2013 from Purdue University, West Lafayette, Indiana, USA. His research interests span distributed systems, networking and machine learning. During the course of his PhD, he interned at Google, Mountain View in the summer of 2009 and at Microsoft Research, Redmond in the summer of 2012. After his PhD, he joined Google, Mountain View, California, as a software engineer in the Platforms Networking Group.

PUBLICATIONS

Karthik Nagaraj, Charles Killian and Jennifer Neville. "Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems". In proceedings of *USENIX Symposium on Networked Systems Design and Implementation.* April 2012.

KC Sivaramakrishnan, Mohammad Qudeisat, Lukasz Ziarek, Karthik Nagaraj and Patrick Eugster. "Efficient Sessions". In proceedings of *Science of Computer Programming.* 2012.

Karthik Nagaraj, Hitesh Khandelwal, Charles Killian and Ramana Rao Kompella. "Hierarchy-Aware Distributed Overlays in Data Centers using DC2". In proceedings of *International Conference on Communication Systems and Networks.* January 2012.

Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson and Ranjit Jhala. "Finding Latent Performance Bugs in Systems Implementations". In proceedings of *International Symposium on the Foundations of Software Engineering.* November 2010.

KC Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek and Patrick Eugster. "Efficient Session Type Guided Distributed Interaction". In proceedings of *International Conference on Coordination Models and Languages*. June 2010.