ABSTRACT

YU, XIAO. Understanding and Debugging Complex Software Systems: A Data-Driven Perspective. (Under the direction of Guoliang Jin.)

As the complexity of software systems has been growing rapidly over recent years, the assurance of reliability and performance is facing more and more challenges. One notable challenge lies in the understanding and debugging of complex systems that consist of many interconnected and interdependent subsystems and components. Such internal dependencies with the growing system scale create difficulties for both humans and tools. This dissertation focuses on addressing this challenge by new techniques that build various models from runtime data generated by live systems. The resulting models provide enhanced understanding for system development and management.

This dissertation presents three techniques that target at three different but common types of systems: operating systems, cloud infrastructures, and request-based applications. These systems are the major parts of a typical modern software stack. All the three techniques are driven by runtime data that contain useful information for system understanding and debugging, but they differ in analysis purposes and targeted problems.

One of these techniques shows how modeling and mining millions of stack traces can help developers spot and diagnose performance problems in real-world Windows installations. This technique can greatly complement the existing testing techniques, which may fail to capture problems that do not appear in the in-house environment. With real-world data, we show that many performance problems are created and amplified by dependencies and interactions between kernel components.

In addition to mining offline data, it is equally important to detect failures and performance problems in real time, especially for systems requiring high availability. So we present CLOUDSEER, a technique that performs online monitoring and problem detection over unstructured, distributed, and interleaved logs generated by cloud infrastructures. CLOUDSEER builds workflow models from existing logs, and uses such models to check live logs for potential execution problems in the form of structured workflow. Such workflow information can facilitate the understanding and resolving of the reported problems.

Finally, we present a technique called DATAFLOW TUNNELING for request-based applications that rely on modern frameworks. These applications face analysis challenges caused by modern frameworks hiding data dependencies between request-handler meth-
ods. **Dataflow Tunneling** infers the hidden data-dependency specifications by a combination of program analysis, tracing, and mining. By our customized tracing, we are able to get useful runtime data that are not directly available to overcome analysis challenges. The inferred specifications enable future work to implement inter-request analysis, which works in the scope across request-handler methods. We expect inter-request analysis to help developers gain application insights from a holistic view.
Understanding and Debugging Complex Software Systems: A Data-Driven Perspective

by
Xiao Yu

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science
Raleigh, North Carolina
2018

APPROVED BY:

Rainer Frank Mueller
Xiaohui Gu

Xipeng Shen
Guoliang Jin
Chair of Advisory Committee
BIOGRAPHY

Xiao Yu was born in Shanghai, China. He grew up with a strong interest in technologies, leading him to become interested in computer programming. While studying for a Bachelor’s degree in Software Engineering at East China Normal University, he briefly worked with Dr. Geguang Pu on a research project. This experience inspired and motivated Xiao to pursue a career in research. In 2008, he started his graduate study under the direction of Dr. Geguang Pu, and finally earned a Master’s degree in 2011.

During his graduate study at East China Normal University, Xiao was mainly working on automated test generation. He participated in the development of CAUT, a tool using dynamic symbolic execution to automatically generate test data for C programs, and published multiple publications in improving effectiveness and efficiency of test generation.

In 2011, Xiao entered the Ph.D. program in Computer Science at North Carolina State University. During his Ph.D. study, he interned at multiple industrial research laboratories, including Microsoft Research Asia and NEC Laboratories America, to broaden his research topics. These experiences stimulated his interest in data-driven techniques for understanding and debugging software reliability and performance problems, and formed the basis of this dissertation. Along this research direction, he developed novel ideas and techniques to leverage logs and traces to enhance performance debugging, system monitoring, and program analysis. He also published multiple publications in flagship conferences, and was granted a U.S. patent.

To open his eyes to and get more involved in the research community, Xiao visited the U.S. Food and Drug Administration and University of Illinois at Urbana-Champaign during his Ph.D. study. In addition, he once served as a reviewer for the journal of Empirical Software Engineering and a member of the ASPLOS 2018 Shadow Program Committee.
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Guoliang Jin for his continuous and generous support during my Ph.D. study. This dissertation would not be a success without his guidance and the research freedom that he provided. I would also like to express my appreciation to my previous advisor Dr. Tao Xie, who helped me build a solid foundation during early years of my Ph.D. study. In addition to my two advisors, I would like to express my special thanks to the two directors of graduate program: Dr. Douglas Reeves and Dr. George Rouskas for their strong support during my advisor transition.

I am very grateful to my internship mentors: Shi Han from Microsoft Research Asia, and Dr. Pallavi Joshi, Dr. Jianwu Xu, and Dr. Hui Zhang from NEC Laboratories America. They provided me valuable guidance and great experiences in industrial research. The real-world research problems unveiled by them led to milestones in my Ph.D. study.

To my advisory committee members Dr. Rainer Frank Mueller, Dr. Xiaohui Gu, and Dr. Xipeng Shen, I would like to thank them for their valuable feedback on this dissertation as well as suggestions for my future research. And I would like to express my special thank to Dr. Xiaohui Gu, who provided me a strong recommendation letter for my job hunting.

Finally, I would like to acknowledge all my lab mates and co-workers for their support. I cannot forget the memorable experience with Sihan Li and Wei Yang in the first three years of our Ph.D. study. We took the same flight from Shanghai to Raleigh, and joined Dr. Tao Xie’s research group together. Although our journey parted later, we still supported each other in many ways. I would also like to thank Xusheng Xiao, who set a great example for me to follow. It was a great experience with all of you. Thank you all!
# TABLE OF CONTENTS

LIST OF TABLES ................................................................................................. vii

LIST OF FIGURES ............................................................................................... viii

PUBLICATIONS ................................................................................................... ix

## Chapter 1 Introduction .................................................................................. 1
  1.1 Overview of Proposed Techniques .............................................................. 4
    1.1.1 Understanding Performance Problems Related to Device Drivers .... 4
    1.1.2 CLOUDSEER: Monitoring Cloud Infrastructures via Logs ............ 5
    1.1.3 DATAFLOW TUNNELING: Mining Inter-request Data Dependencies for
           Request-based Applications ......................................................... 6
  1.2 Problem Dimensions ............................................................................... 7
  1.3 Related Work .......................................................................................... 9
  1.4 Contributions ....................................................................................... 9

## Chapter 2 Comprehending Performance from Real-World Execution Traces: A
Device-Driver Case ......................................................................................... 11
  2.1 Introduction .......................................................................................... 11
  2.2 Background and Motivating Examples .................................................... 14
    2.2.1 Execution Traces ........................................................................... 14
    2.2.2 A Real-World Case ...................................................................... 16
    2.2.3 How Our Approach Facilitates Performance Analysis .................. 18
  2.3 Impact Analysis .................................................................................... 19
    2.3.1 Data Abstraction ........................................................................... 19
    2.3.2 Impact Measurement .................................................................... 20
  2.4 Causality Analysis ................................................................................ 21
    2.4.1 Data Abstraction and Pattern Representation ............................... 22
    2.4.2 Pattern Discovery ......................................................................... 24
  2.5 Empirical Results and Evaluation on Device Drivers ........................... 28
    2.5.1 Results of Impact Analysis on Device Drivers ............................... 28
    2.5.2 Evaluation of Causality Analysis .................................................. 29
  2.6 Related Work ........................................................................................ 35
    2.6.1 Performance Analysis .................................................................. 35
    2.6.2 Quality Assurance for Device Drivers ......................................... 37

## Chapter 3 CLOUDSEER: Monitoring Cloud Infrastructures via Interleaved Logs . 38
  3.1 Introduction .......................................................................................... 38
  3.2 Overview ............................................................................................. 42
    3.2.1 Cloud-Infrastructure Example: OpenStack .................................. 42
4.5.1 Analysis of Web Applications ........................................... 91
4.5.2 Program Specification Mining and Its Applications ............... 91
4.5.3 Dynamic Taint Tracking .................................................. 92

Chapter 5 Conclusion and Future Work ........................................ 93
5.1 Future Work ...................................................................... 94
  5.1.1 End-to-end Modeling of Cross-component Flows .................. 94
  5.1.2 Reproducing Problematic System Behaviors ...................... 95
  5.1.3 Applications of Inter-request Analysis on Web Applications .... 95

BIBLIOGRAPHY .................................................................... 97
LIST OF TABLES

Table 1.1 Problem dimensions of the proposed techniques. ............... 7
Table 2.1 Selected scenarios for the evaluation of causality analysis. .......... 30
Table 2.2 Impactful-time and total-time coverages of discovered patterns on the total driver time. .............................................. 31
Table 2.3 Coverages by top 10%, 20%, and 30% of the ranked patterns. ........ 33
Table 2.4 Drivers involved in top-10 patterns of each scenario categorized by driver types. .................................................. 33
Table 3.1 Demonstration of CLOUDSEER’s checking process. .................. 46
Table 3.2 VM tasks for experiments. .............................................. 56
Table 3.3 Experiments for accuracy and efficiency. ................................. 58
Table 3.4 Execution points for failure injection. ................................. 58
Table 3.5 Experiment results for accuracy. ........................................ 59
Table 3.6 Experiment results for efficiency. ........................................ 61
Table 3.7 Experiment results for problem detection. ................................ 61
Table 4.1 Instrumentation and tracing rules. Trac Point: whether a trace event is fired on an instrumentation point (Inst) or on a proxy object (Proxy); Exec Point: the current scope of execution, in a request-handler method or the framework; Target Statement: the instrumented code for an instrumentation point, the intercepted method invocation for a proxy point(obj always references a proxy object); Tracing Action and Conditions: the tracing step performs the action under certain conditions. 77
Table 4.2 Specification accuracy of ITracker and OpenMRS. .................... 86
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An illustration of cost propagation due to lock contentions and hierarchical dependencies among device drivers.</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>An illustration of an Aggregated Wait Graph for device drivers.</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Inter-service interactions when booting a VM.</td>
<td>42</td>
</tr>
<tr>
<td>3.2</td>
<td>Simplified log sequences for booting VMs in OpenStack.</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>The automaton for booting a VM.</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>Removing message dependencies from task automata.</td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Message ordering: normal and reordering cases.</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>The abstract request-based execution model.</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>An excerpt of three related request-handler methods and their view templates from OpenMRS. Only method parameters related to our discussion are shown.</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>Tracing overhead by requests.</td>
<td>88</td>
</tr>
</tbody>
</table>
PUBLICATIONS

This dissertation is based on the following publications from my Ph.D. research:

• Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie.  
  Comprehending Performance from Real-World Execution Traces: A Device-Driver Case.  
  ACM ASPLOS 2014 [136].  
  The major part of this work was completed during my internship at Software Analytics Group of Microsoft Research.

• Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang.  
  ACM ASPLOS 2016 [138].  
  The major part of this work was completed during my internship at Autonomic Management Department of NEC Laboratories America.

• Xiao Yu and Guoliang Jin.  
  Dataflow Tunneling: Mining Inter-request Data Dependencies for Request-based Applications.  
  ACM/IEEE ICSE 2018 [137].
INTRODUCTION

Software systems have never been as complex as they are today. Such complexity is driven by the increasing user requirements, and it is in part reflected by numerous integrated and stacked subsystems and components, which range from those in low-level operating systems to high-level applications. A large data center can run numerous servers, in which operating systems host the whole software infrastructure; some infrastructure software supports individual applications; and specialized applications serve end-user requests. Within each of these elements, there are also many individual components designed for more specialized functions. Despite the physical size of hardware, a small smart phone can also have a similar structure including multilevel subsystems and components, and they work collaboratively to serve functionality impacting our daily life.

While dividing a system into subsystems and components provides great flexibility and scalability for the increasing user requirements, it also brings challenges to the assurance of reliability and performance throughout system development, operation, and maintenance. Generally speaking, a growing system has to integrate more and more components, which are diverse, and interdependent or interconnected. A poorly designed component or inter-
action between components would not only introduce failures and performance problems, but also escalate the problems through interactions to other components. Such escalation is prevalent and can be disastrous in real-world systems. For example, multiple notorious incidents in Amazon Web Services (AWS) [110, 111] simply started from some ordinary network problems, but these problems later triggered latent software bugs that caused severe consequences affecting millions of users. Such incidents clearly exhibit how a small and seemingly unharmed problem in one single component can trigger a disastrous total system failure in real-world complex systems.

Developers have long been seeking and applying various techniques to ensure system reliability and performance. In the development stage, in-house testing is the front line of catching problematic components, but it does not always succeed. In the real-world scenarios, test cases may be insufficient to cover all possible component behaviors and interactions under diverse operation environments. And since complex systems are usually extensible, some third-party components may not even be available in the in-house testing environment. Program analysis is another in-house means, but it does not scale well to large systems with sophisticated software architectures. In such systems, component dependencies are usually determined and established at runtime, which would cause program analysis to generate imprecise results.

As problems would inevitably happen in production runs, developers hope to capture them on their occurrences, and further debug and fix their underlying causes in the operation stage. That is the last line of defense against unreliability and poor performance. To effectively capture such problems, developers rely on certain runtime data collection mechanisms, which are commonly tracing and logging. The collected data provide a limited view for developers to observe system behaviors in the real-world scenarios. Then the question becomes how developers can effectively analyze such data. In complex systems, logs and traces can be in a very large size, and they usually contain very noisy information, both relevant and irrelevant to problems to be analyzed. As a result, developers still need proper tools and techniques to navigate through pieces of collected information to understand the occurred problems and their root causes. And it still remains an open question that how such data and findings would further complement and benefit the development stage in building more reliable and performant systems other than just being the last defense.

Under the aforementioned status quo, this dissertation attempts to address a fundamental question:
**How to leverage runtime data to understand and debug complex software systems?**

Our attempt leads a direction that spans multiple research areas including software engineering, software systems, and data science with two main focuses: problem-driven exploration of runtime data, and usability of discovered insights. As we will present in later chapters of this dissertation, our proposed techniques are all driven by specific reliability and/or performance problems as well as analysis challenges in various systems, including operating systems, cloud infrastructures, and request-based applications. They constitute the modern software ecosystem, and are inevitably facing various problems caused by their specific system characteristics. Driven by these problems and their related characteristics, our techniques can provide useful insights for developers or tools. Such insights may go beyond just catching individual failures or performance problems. For example, **DATAFLOW TUNNELING**, our most recent technique targeting at request-based applications, infers data dependencies across hard-to-analysis application components. The inferred dependencies aim at enhancing program analysis tools to perform more analyses and find different kinds of problems. **DATAFLOW TUNNELING** has gained very positive feedback from its reviewing process. One reviewer even called it “a very fundamental advance in the topic of analysis of web applications.”

All techniques in this dissertation attempt to address a common research challenge, which is the modeling of data in applying data-driven methods. Runtime data, specifically logs and traces, could be very noisy, containing both relevant and irrelevant information of interest. A representative example is system-wide execution traces collected by modern tracing frameworks, such as Event Tracing for Windows (ETW) [42]. It produces tracing events in multiple system layers, from user-mode applications to the system kernel. As a result, the potential investigation scope for reliability or performance problems could be hard to determine, calling for effective modeling techniques to assist.

Sometimes too much information does not mean useful information. As the log and trace generation process incurs performance overhead, it always involves balanced decisions between acceptable overhead and collected information that preserves enough details of system behaviors. Logging is a typical example of such balanced decisions. Although logs with great details can be very useful in debugging live systems, a verbose logging level is rarely used as it would degrade system performance. Consequently, the information in logs may be incomplete or not explicit enough for debugging purposes. Some existing techniques, such as LogEnhancer [141] and Errlog [140], attempt to augment logs with
additional information for debugging.

This dissertation presents various models that provide a proper scope and extract useful information over noisy runtime data. To analyze performance problems related to calling and waiting dependencies between operating system components, we propose a graph-based data abstraction over millions of callstack traces. For cloud systems involving distributed components being executed collaboratively, we propose a workflow model over logs to facilitate the understanding of system executions. For components with implicit dependencies in request-based applications, we propose a tracing technique to model object accesses across components for further analysis of data dependencies.

1.1 Overview of Proposed Techniques

We present a high-level overview of all the proposed techniques, which will be discussed in details in later chapters. This overview discusses specific problems, challenges, and highlights in each technique.

1.1.1 Understanding Performance Problems Related to Device Drivers

An operating system can be regarded as a software ecosystem, in which multiple components interact with each other. The operating system performance can therefore be affected by problematic components and their interactions with others. And it is difficult to diagnose performance problems involving component interactions. Such difficulties are mainly reflected by subtle triggering conditions and root causes hidden deeply from places where problems manifest.

Addressing performance problems related to component interactions depends on two key questions: (1) what problematic components and their runtime behaviors are in the real world; and (2) to what extent and how they would affect the overall system performance. In practice, execution traces from real-world systems provide rich information to answer these questions, because they capture usage scenarios with diverse components and interactions that are not seen during in-house testing. However, these traces contain millions of discrete tracing events, rendering inspection without automated techniques ineffective.

With execution traces collected from 339 hours of real-world usage of Windows, our findings indicate that Windows device drivers constitute a non-trivial part in the overall system performance through component interactions including function invocations and
synchronizations: (1) a driver can invoke other drivers in the same driver stack through the kernel, and (2) a driver can block other drivers by synchronizations. The combination of these interactions can propagate and amplify performance problems across multiple components and processes. Based on these findings, we propose two general automated analyses to answer the key questions: (1) impact analysis measuring what drivers and to what extent they impact the overall system performance, and (2) causality analysis identifying interaction patterns of high-impact drivers that are likely to cause perceived performance problems. On the collected traces, the impact analysis reveals that device drivers constitute 36.4% on waiting time and 1.6% on running time, and 26.0% of the driver waiting time is due to the amplification of interactions. The causality analysis further discovers some real-world cases of high-impact drivers with their long interaction paths that lead to significant performance problems.

1.1.2 CLOUDSEER: Monitoring Cloud Infrastructures via Logs

Cloud infrastructures provide a rich set of management tasks, whose executions involve interactions and collaborations among multiple distributed components. Monitoring such task executions is crucial for cloud providers to promptly identify and fix problems that compromise cloud availability. When a reliability problem occurs, such as performance degradation, it is particularly useful to know what task execution is affected and what component may be the source of the problem.

Monitoring logs centralized from different components is a common practice. This process is lightweight and non-intrusive, and does not introduce extra complexity or overhead to the system under monitoring. However, even when centralized, logs are still unstructured and highly interleaved due to the distributed and multi-tenant nature of such systems, limiting log monitoring to specific types of log messages, but not associating them with task executions. So this question naturally raises: is it possible to monitor how individual tasks are being executed by attributing individual log messages to their corresponding task executions?

By investigating logs in OpenStack, a popular open-source cloud-infrastructure system, we identify both opportunities and challenges for log monitoring: (1) log timestamps can determine the order of distributed components executing the same task, but asynchronous executions also introduce non-deterministic message orders; (2) some resource identifiers shared by log messages can be used to attribute messages to executions, but no single
identifier is shared by all messages in the same execution. Based on these observations, we propose CLOUDSeer, a log-based monitoring approach to monitor workflows of task executions. CLOUDSeer features: (1) a modeling technique, which abstracts log messages and their possible orders into an automaton model for each task, and (2) an efficient checking algorithm, which uses the modeled automata and partially shared identifiers to accurately attribute interleaved log messages to their corresponding task executions, and reports state divergences as potential failures. The experiments on OpenStack show that the approach can correctly and efficiently associate at least 92.08% of the task executions with their log messages, and can achieve a precision of 83.08% and a recall of 90.00% on the injected problems.

1.1.3 DATAFLOW TUNNELING: Mining Inter-request Data Dependencies for Request-based Applications

Request-based applications serve as the front end of server applications, such as web sites and RESTful APIs. These applications adopt the request-based execution model, which typically consists of a set of request-handler methods invoked upon user requests by an application framework. In this model, highly modular and stateless methods make the analysis and understanding of the application as whole difficult. Such difficulties often lead to repetitive and uncoordinated application behaviors across requests, further affecting application's reliability, performance, and security. For example, methods handling two related requests may repeatedly create identical data objects without the knowledge of request-input dependencies. To avoid such problems from happening, it is crucial for developers to understand what data dependencies may exist between related request-handler methods.

Conventional interprocedural analysis is ineffective to infer and analyze such data dependencies, because they along with method calling relationships are implicitly established by the complex framework and request sequences. In addition, components involved in propagating data from request input to output are usually written in different languages, e.g., request-handler methods in ordinary object-oriented languages, while view templates rendering request output in markup languages.

To solve these analysis challenges, we propose DATAFLOW TUNNELING, a dynamic approach to capture indirect data dependencies between request-handler methods. This approach uses a lightweight tracing technique to capture how object data flow along complex
framework behaviors without instrumenting the framework code, and a mining technique to reconstruct data dependencies. Our evaluation shows that the reconstructed dependencies are over 80% accurate and useful in understanding the behaviors of request-handler methods in the inter-request scope, such as tuning object caching policies. We envision this approach as a foundation for future full-fledged analysis tools.

1.2 Problem Dimensions

Along with the overview of proposed techniques, we summarize and present in Table 1.1 critical dimensions that we need to consider in applying data-driven techniques for software systems. These dimensions are related to the purposes of applying data-driven analysis, characteristics of systems, how data are handled, and how results are represented. For instance, this dissertation covers three different purposes that can benefit from data-driven analysis.

In other dimensions, component interactions are the one directly related to system behaviors, and it is correlated to the dimension of data modeling. Although our techniques target at different types of systems, it is their underlying component interactions that make

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Performance analysis <em>(Chapter 2)</em></td>
</tr>
<tr>
<td></td>
<td>Monitoring <em>(Chapter 3)</em></td>
</tr>
<tr>
<td></td>
<td>Program analysis <em>(Chapter 4)</em></td>
</tr>
<tr>
<td>Component interactions</td>
<td>Calling dependencies <em>(Chapter 2, Chapter 3, Chapter 4)</em></td>
</tr>
<tr>
<td></td>
<td>Lock contentions <em>(Chapter 2)</em></td>
</tr>
<tr>
<td></td>
<td>Distributed communications <em>(Chapter 3)</em></td>
</tr>
<tr>
<td>Data source</td>
<td>Existing logs or traces <em>(Chapter 2, Chapter 3)</em></td>
</tr>
<tr>
<td></td>
<td>Customized tracing <em>(Chapter 4)</em></td>
</tr>
<tr>
<td>Data processing</td>
<td>Offline <em>(Chapter 2)</em></td>
</tr>
<tr>
<td></td>
<td>Online <em>(Chapter 3, Chapter 4)</em></td>
</tr>
<tr>
<td>Data modeling</td>
<td>Graph-based <em>(Chapter 2)</em></td>
</tr>
<tr>
<td></td>
<td>Automaton-based <em>(Chapter 3)</em></td>
</tr>
<tr>
<td></td>
<td>Sequence-based <em>(Chapter 4)</em></td>
</tr>
<tr>
<td>Result representation</td>
<td>Human-oriented <em>(Chapter 2, Chapter 3)</em></td>
</tr>
<tr>
<td></td>
<td>Tool-oriented <em>(Chapter 4)</em></td>
</tr>
</tbody>
</table>
true differences in the technique design. Types of interactions can affect what runtime data could be and how they should be modeled and analyzed, i.e., the dimension of data modeling. For example, calling dependencies can usually be reflected by stack traces, which can be modeled as simple sequence patterns. However, the co-existence of lock contentions or distributed communications in complex systems may result in nondeterministic logging or tracing entries, which calls for sophisticated modeling, such as the graph-based or the automaton-based model.

On the data source, complex software systems usually produce various kinds of logs and traces, but we need to consider whether such data are sufficiently useful for our specific analysis purposes. This dissertation describes techniques for both cases. In our first technique for device drivers, we use execution traces that have been practically used in performance analysis. In CLOUDSeer, we repurpose existing end-to-end logs from distributed components for online monitoring. And in DATAFLOW TUNNELING, we design a customized tracing mechanism for request-based applications in which no available runtime data can fit our purpose.

With proper data, we must determine how our analysis algorithms would work to process data: in an offline manner or online. This choice is closely related to the purpose dimension. While offline analysis is widely applicable to gain insights for development and debugging activities such as performance analysis, we must design techniques that can process incoming data online for scenarios like monitoring. It is notable that we also design DATAFLOW TUNNELING to be able to infer data dependencies online to avoid storing concrete user data.

The ultimate goal of data-driven analysis is to get useful and actionable results from data. Then the way of representing the results is another consideration to make. We aim at two types of representations: human-oriented and tool-oriented. A human-oriented representation helps developers understand system behaviors and related reliability or performance problems. We design a function-signature based representation to explain how calling dependencies and lock contentions would cause performance problems. We also design an automaton-based representation to pinpoint the component involved in a failed execution workflow for developers to investigate. On the other hand, a tool-oriented representation aims at enhancing tool capability in analyzing complex system components. In DATAFLOW TUNNELING, we feature a data-dependency specification for program analysis tools to perform analysis in the inter-request scope.
1.3 Related Work

While the discussion of related work for each technique is in their respective chapter, we highlight in this section some related techniques that are subsequently developed by others after the publications of ours. We consider that they are partly inspired by our techniques and take one step forward from ours.

SyncProf [134, 135] is an end-to-end technique that detects, localizes, and optimizes synchronization bottlenecks. Similar to our technique in Chapter 2, SyncProf uses a graph-based model to abstract wait relations between threads, measure performance impact of synchronization bottlenecks, and identify potential root causes. As our technique only focuses on detecting such bottlenecks, SyncProf takes one step forward to automatically optimize them by a set of lock-related rules. Such subsequent optimization clearly shows the usefulness of data-driven analysis in analyzing complex systems.

Some newly published log-based techniques, such as structural event detection by Wu et al. [130], anomaly diagnosis by Jia et al. [55], and DeepLog [40], all adapt the idea of workflow-based monitoring used in CloudSeer to detect anomalies in complex systems. DeepLog [40] adapts the cutting-edge deep learning techniques to build workflow models and monitor logs with a higher accuracy. Wu et al. [130] presents an advanced learning technique to detect structured workflows in noisy logs from open systems. Such open-system logs would render temporal relations adopted by CloudSeer inaccurate in describing workflows. As machine learning techniques keep flourishing in recent years, we will see more improvement opportunities for our proposed techniques.

1.4 Contributions

We describe our main contributions and the organization of this dissertation as follows.

In Chapter 2, we describe a trace-based performance analysis technique that works on millions of executions traces from real-world Windows installations. That chapter presents not only the technique, but also empirical evidence on how device drivers in the Windows kernel would cause significant performance problems, as well as findings and insights from real-world scenarios.

In Chapter 3, we describe CloudSeer, a technique that performs online monitoring and problem detection over unstructured, distributed, and interleaved logs generated by
cloud infrastructures. **CLOUDEER** builds workflow models from logs, and uses such models to check live logs for potential execution problems in the form of structured workflow. Such workflow information can facilitate the understanding and resolving of the reported problems.

In Chapter 4, we describe **DATAFLOW TUNNELING**, a technique for request-based applications that rely on modern frameworks. **DATAFLOW TUNNELING** infers the hidden data-dependency specifications by a combination of program analysis, tracing, and mining. The inferred specifications enable future work to implement inter-request analysis, which works in the scope across request-handler methods, and may help developers gain application insights from a holistic view.

Finally, in Chapter 5, we conclude our experience of applying data-driven analysis in various software systems, and discuss some future directions that make a step further towards reliable and performant systems.
2.1 Introduction

Real-world execution traces collected by mature infrastructures, such as the Event Tracing for Windows (ETW) [42] and DTrace [39] for Unix-like systems, offer tracing events with timestamps and callstacks for both user-mode applications and system layers. Such traces can record performance problems that are likely perceived at deployment sites and usually triggered by complex runtime environments and real-world usage scenarios. By analyzing large-scale and diverse execution traces collected from deployment-site computers, performance analysts and developers can gain useful knowledge for system design and optimization.

However, finding performance problems from real-world traces is challenging. The
problems can be rooted subtly and deeply into system layers or other components far from the place where delays are initially observed. Performance analysts have to spend great efforts to confirm the existence of subtle problems, and to figure out how they were caused and spread. Given many real-world execution traces where bad performance may be amortized over various underlying problems, such performance-analysis activities become harder or even infeasible.

Based on our observation in real-world traces, we identify and generalize the concept of cost propagation in systems consisting of interacting components as an important underlying mechanism that introduces the subtleness of performance problems. In particular, the term “cost propagation” refers to a phenomenon that the cost of executing a component is propagated and accumulated to the execution of another component through cross-component interactions.

There are different kinds of interactions causing cost propagation. Call dependency is a well-known kind, which accumulates costs from callees to callers. Lock contention is a subtle kind, which allows a delay to be propagated to all other components waiting for the same lock. In complex software ecosystems (e.g., Windows systems), different kinds of interactions can be combined together to create performance bottlenecks involving multiple victims. For example, there can be a very long propagation path or multiple propagation paths consisting of both call dependencies and contentions on different locks.

The cost propagation brings challenges to performance analysis of complex systems. Existing techniques, such as call-graph profiling [47] and lock-contention analysis [115], face two major limitations. First, these techniques cover a single aspect of underlying interactions only, which is either call dependency or lock contention. Their combinatorial effects are not generally considered or addressed. Second, these techniques usually work on a single program with a limited investigation scope, but cannot explain performance problems with complex cause-and-effect chains that are transparent to the program, but are grounded in the underlying system. To better reveal and explain these problems, a large number of system-level execution traces can be leveraged from real-world deployment sites. Handling these traces requires effective abstractions that can capture essential information, and classify similar and distinct runtime behaviors towards providing concise and actionable results.

To address these challenges, we propose a new approach consisting of two steps: impact analysis and causality analysis. The impact analysis extracts Wait Graphs [49] (Section 2.3.1)
from execution traces, and measures performance impacts with respect to cost propagation on Wait Graphs. Performance analysts can narrow down the investigation scope by choosing highly suspicious components to measure, and decide whether the causality analysis is needed based on the significance of performance impacts.

The causality analysis discovers behavioral patterns of highly suspicious and high-impact components that are likely to cause perceived performance problems. To capture the essence of problematic behaviors with respect to cost propagation, we design the Signature Set Tuple (Section 2.4.1) as a pattern representation that presents abstracted and generalized behaviors to performance analysts. A Signature Set Tuple generalizes interacting behaviors among components in the form of signature sets (a set of function signatures extracted from callstacks). Such abstraction and generalization facilitate the comprehension of performance problems by narrowing down the investigation scope as well as preserving highly suspicious behaviors that may appear similarly in different traces.

The causality analysis is grounded in the technique of contrast data mining \cite{16} to address challenges in discovering problematic behaviors. Such challenges are two-fold: bad performance can be amortized over multiple problematic behaviors, and there is no easy oracle for identifying problematic behaviors, especially for the expensive but necessary ones. Our technique exploits the fact that a large number of execution traces may contain various runtime behaviors with both good and bad performance. In particular, our technique defines two contrast classes over execution traces as a fast class and a slow class based on thresholds of execution time (usually specified by developers as the expectations on performance). The fast class contains expected runtime behaviors, whereas the slow class contains problems to be identified. Our technique applies two criteria to identify contrast patterns: (1) a pattern appears in the slow class but not in the fast class; (2) a pattern appears in both classes, but the pattern in the slow class shows a significant amount of execution time compared with the pattern in the fast class. The identified contrast patterns can reflect underlying key factors that discriminate the performance contrasts.

In this work, we choose device drivers in Windows operating systems as a representative to demonstrate our approach and study performance problems. Device drivers constitute a majority part (about 70\%) in the Linux code base \cite{96}, as well as in the Windows kernel \cite{64}. Performance analysts in Microsoft product teams have noticed that a non-trivial number of response delays in user-mode applications were likely caused by device drivers. Therefore, it is interesting, as well as necessary, to study how device drivers affect overall system
performance.

We notice that two key characteristics in device drivers can cause cost propagation. First, locks are widely used in device drivers to synchronize shared resources over the system kernel. Second, device drivers are organized in a hierarchical architecture (a.k.a driver stack) in which drivers can interact with each other via system services, e.g., the IoCallDriver routine in Windows. The form of hierarchical dependencies connecting multiple contention points of different locks can propagate a delay to all affected drivers and corresponding user-mode applications to create remarkable performance problems.

**Main Contributions.** In this work, we make main contributions from two aspects. On performance analysis, we propose a practical two-step approach with effective data and pattern abstractions to measure performance impacts manifested through cost propagation, and discover behavioral patterns closely related to performance problems via contrast data mining. The effectiveness of the pattern discovery is examined and confirmed by our automated evaluation, driver developers, and performance analysts.

By applying the impact analysis on 19,500 real-world execution traces (339 hours of duration in total, collected from Windows machines), we show the empirical evidence of performance impacts posed by device drivers. Our findings show that device drivers constitute 36.4% on waiting time and 1.6% on running time in overall Windows performance, within which cost propagation causes 26.0% on waiting time. Moreover, by applying the causality analysis on some typical application scenarios, we show concrete examples of performance problems related to device drivers. These findings suggest that identifying and minimizing potential cost propagation in device drivers should be an important direction for designing a high-performance system, especially for the cases involving driver/driver interactions, which complement the discussion by Kadav et al. [64].

### 2.2 Background and Motivating Examples

#### 2.2.1 Execution Traces

Execution traces are the data source of our trace-based performance analysis. To simplify the data representation while preserving essential information, we use an abstracted schema called *trace stream*. The schema is compatible with existing popular tracing infrastructures, e.g., ETW [42] and DTrace [39].
Trace Stream. A trace stream $TS$ is a sequence of tracing events $e_0e_1e_2\ldots e_{L-1}$. Each event $e$ falls into one of the following four types: (1) a running event represents CPU usage sampled in a constant interval, e.g., 1 millisecond in ETW and DTrace; (2) a wait event occurs when a thread enters the waiting state due to blocking operations, e.g., a thread tries to acquire a lock being held by others; (3) an unwait event occurs when a running thread signals another thread in the waiting state to continue execution, e.g., a thread releases a lock or sends a message; (4) a hardware service event is recorded with a start timestamp and duration in the period of a hardware operation.

Each event $e$ is also related to a set of fields, which are callstack (denoted as $e.S$), timestamp ($e.T$), cost as time duration ($e.C$), the related thread ID ($e.TID$), and the ID of the other thread to be unwaited ($e.WTID$). These fields are sufficient for capturing and characterizing essential runtime behaviors under performance analysis from a cost-propagation perspective.

Scenario. Performance analysts usually start performance analysis with a scenario, e.g., BrowserTabCreate for creating a browser tab, to explore what run slow in that scenario. During a period of time, a system could have multiple ongoing scenarios simultaneously, since a user can browse websites while the system is performing other operations. Performance analysts have a set of predefined scenarios that are used to capture scenario-related execution traces.

Scenario Instance. A trace stream can contain events for multiple scenarios that were being performed by the system during the tracing period. Among these events, an event sequence representing the execution of a single scenario is called a scenario instance. A scenario instance typically starts and ends with the events from a single initiating thread, which initiates the execution of a particular scenario. For example, a browser UI thread that reacts to a user’s request of creating a new browser tab is an initiating thread for BrowserTabCreate. Formally, a scenario instance $I$ of a scenario $S$ is a tuple $(TS, S, TID, t_0, t_1)$, indicating that the execution of the scenario $S$ starting from a thread (with the identifier $TID$) within the time period between $t_0$ and $t_1$ is recorded in a trace stream $TS$.

Some events may overlap across multiple instances in the same trace stream. Such overlap indicates that other threads are suspended by the thread that triggers those overlapped events, so it is a typical manifestation of cost propagation. Such observation is the key to our approach to be effective in analyzing performance problems related to cost propagation.
2.2.2 A Real-World Case

We use a real-world case to show how cost propagation in device drivers is involved in producing a performance problem. Due to confidentiality, we anonymize the names of device drivers and relevant resources in this example.

An execution instance of the scenario BrowserTabCreate cost over 800 milliseconds to complete. From the user’s perspective, the web browser fully displayed a new tab in over 800 milliseconds after the user had clicked “create a new tab.” Such delay is perceivable on the user interface.

This problem involves three device drivers having lock contentions and hierarchical dependencies. First, File Virtualization filter driver fv.sys uses locks to synchronize queries on the entries of an internal File Table that maps some “virtual” files to their physical locations on the file system. Second, File System driver fs.sys acquires locks on Meta Data Units (MDUs) that contain file metadata when there are requests on reading or writing a file. Third, Storage Encryption driver se.sys performs computation-intensive encryption and decryption when data are being read and written on storage media. The three device drivers form a hierarchy in which the top-most driver fv.sys invokes fs.sys, whereas fs.sys invokes se.sys. To ease the illustration, we omit those less relevant drivers that may exist in the hierarchy on a real machine.

Figure 2.1 is a thread-level snapshot restructured from the trace stream to show the period of delay. The figure presents each thread by its corresponding callstack that contains

![Figure 2.1 An illustration of cost propagation due to lock contentions and hierarchical dependencies among device drivers.](image-url)
ongoing operations. There were six threads executing the three device drivers during this period. We denote these threads by the notation $T_{X,Y}$, which means the thread $Y$ of the process $X$. The dotted arrows in the figure represent the directions of cost propagation among threads.

**Lock Contentions on the File Table.** When the user clicked “create a new tab” on the browser, the browser UI thread $T_{B,UI}$ started to access some “virtual” files. So $T_{B,UI}$ executed $fv.sys$ to query the File Table. However, two other worker threads $T_{B,W0}$ and $T_{B,W1}$ were performing file operations in the background, and happened to execute $fv.sys$ at the same time. As a consequence, the three threads were contending a lock on some entries in the File Table, thereby forming a region of lock contentions in $fv.sys$ (shown as the upper dashed box in Figure 2.1). Among these three threads, $T_{B,W1}$ first got the lock, so the other two threads had to wait.

**Hierarchical Dependency Between $fv.sys$ and $fs.sys$.** $T_{B,W1}$ proceeded to execute $fs.sys$ by a function call initiated from $fv.sys$, thereby creating a hierarchical dependency between the two device drivers (shown as the arrow (4) in Figure 2.1). Thus, the continuation of executing $fv.sys$ depended on the return of $fs.sys$.

**Lock Contentions on Meta Data Units (MDUs).** During the period that $T_{B,W1}$ was running, two threads $T_{A,W0}$ and $T_{C,W0}$ from two other applications (i.e., the AntiVirus and the Configuration Manager applications in Figure 2.1) were contending a lock on MDUs in $fs.sys$. Coincidentally, $T_{B,W1}$ joined to contend the same lock. Therefore, $T_{B,W1}$, $T_{A,W0}$, and $T_{C,W0}$ created another region of lock contentions in $fs.sys$, shown as the lower dashed box in Figure 2.1. $T_{C,W0}$ first got the lock, and proceeded.

**Hierarchical Dependency Between $fs.sys$ and $se.sys$.** $T_{C,W0}$ established a hierarchical dependency between $fs.sys$ and $se.sys$ by a system-service call that reads data from disk (shown as the arrow (1) in Figure 2.1). The system thread $T_{S,W0}$ was scheduled to serve the read request by fetching data from disk and executing $se.sys$ to decrypt. To accomplish the task, $T_{S,W0}$ cost hundreds of milliseconds.

**Summary: Manifestation of Cost Propagation.** The three device drivers together created a bottleneck that consisted of two regions of lock contentions, and two hierarchical dependencies over the six threads (i.e., three browser threads, one AntiVirus thread, one Configuration-Manager thread, and one system thread). A delay occurring in $T_{S,W0}$ was propagated and accumulated through the path shown as the arrows from (1) to (6) in Figure 2.1 to the UI thread $T_{B,UI}$. Consequently, the user perceived an obvious delay when
clicking a button on the browser to create a new tab. In addition, the other two applications in this case were also affected by such performance impact. Reducing the granularity of locks is a general principle to alleviate such problem.

2.2.3 How Our Approach Facilitates Performance Analysis

The preceding example shows the subtleness caused by cost propagation in real-world performance problems. In practice, a performance analyst has to start with the browser UI thread to examine many executed functions in many traces, since the costs may vary in different executions. Such work is usually tedious. Even if the analyst finds out that \texttt{fv.sys!QueryFileTable} could incur high execution cost in lock contentions among different browser threads under some circumstances, the analyst still needs to realize the relations across three drivers and many threads that constituted the whole performance. Realizing such relations is the key to future performance tuning.

By employing impact analysis, the analyst can first realize to which extent the performance of scenario \textit{BrowserTabCreate} was affected by cost propagation. The analyst may conduct impact analysis on different scopes to realize performance impacts of different components. The results serve as the preliminary evidence pointing to some potentially hidden problems. Then the analyst can apply causality analysis on high-impact components, for instance, device drivers in this case. The causality analysis can suggest a list of contrast patterns sorted by their performance impacts. We pick the following one to explain in detail. The pattern is in the form of Signature Set Tuple, which is formally defined in Section 2.4.1.

\[
\begin{align*}
\text{wait signatures} & : \{\texttt{fv.sys!QueryFileTable}, \texttt{fs.sys!AcquireMDU}\} \\
\text{unwait signatures} & : \{\texttt{fv.sys!QueryFileTable}, \texttt{fs.sys!AcquireMDU}\} \\
\text{running signatures} & : \{\texttt{se.sys!ReadDecrypt}, \texttt{DiskService}\}
\end{align*}
\]

This pattern describes that the cost of the running signatures \texttt{se.sys!ReadDecrypt} and \textit{Disk Service} can be propagated through the two unwait signatures \texttt{fv.sys!QueryFileTable} and \texttt{fs.sys!AcquireMDU} to the wait signatures, while functions represented by the wait signatures are invoked by the browser threads. The causality analysis can discover this pattern because in normal cases either such pattern does not appear, or it has much lower cost.
The discovered pattern can help an analyst from two aspects. First, it guides the analyst to realize the concrete performance incident by investigating a specific trace stream. The preceding example is actually restructured with the help of the discovered pattern. Second, the pattern as a generalized representation is a clue for similar cases. The analyst may prioritize the search of the three driver signatures in other cases to facilitate future analysis.

2.3 Impact Analysis

The goal of impact analysis is to scope and measure performance impacts for some chosen components, such as all device drivers. The impact analysis takes two inputs: (1) the instances of various scenarios over trace streams; (2) the component name(s) that are used to filter tracing events for the chosen components to be measured.

The impact analysis outputs three metric values: (1) the running percentage \( IA_{run} \); (2) the wait percentage \( IA_{wait} \); (3) \( IA_{opt} \), the percentage of waiting time introduced by cost propagation.

The three metrics suggest three performance aspects of the chosen components. A large \( IA_{wait} \) indicates that the execution of the components is frequently blocked by others, whereas a large \( IA_{run} \) reflects a computation-intensive characteristic. \( IA_{opt} \) suggests how much extra cost is introduced by waiting on others. It can also serve as an upper bound for the optimization potential.

2.3.1 Data Abstraction

The impact analysis measures the running time and the waiting time for the chosen components, respectively. To serve this purpose, we use the Wait Graph structure from Stack-Mine [49] to model scenario instances. A Wait Graph encodes \textit{wait} and \textit{unwait} events into wait chains, with \textit{running} events as well. So it enables an easy way to measure both running and waiting time for impact analysis; otherwise, we would need to simultaneously keep track of each \textit{wait} event to its corresponding \textit{unwait} event.

\textbf{Definition 1} A Wait Graph is a graph \( WG = (V, E) \). \( V \) is a set of nodes \( V = \{e\} \), where \( e \) is a tracing event. \( E \) is a set of directed edges \( E = \{e_i \rightarrow e_j\} \). For each edge \( e_i \rightarrow e_j \), \( e_i \) must be a wait event, indicating that the thread triggering \( e_i \) keeps suspended before \( e_j \) occurs and completes, and \( e_j \) is triggered by another thread during the wait interval of \( e_i \).
The Wait Graph of a scenario instance is basically constructed by (1) pairing each wait event with its corresponding unwait event to restore wait chains among threads, (2) edging events as graph nodes based on the restored wait chains, and (3) restoring the duration of wait events from the timestamps on paired wait/unwait events. StackMine [49] contains an algorithm description for constructing a Wait Graph. We construct Wait Graphs based on that algorithm for our impact analysis and causality analysis with characteristics of Windows systems.

2.3.2 Impact Measurement

2.3.2.1 Basic Metrics

For every scenario instance, the impact analysis constructs a corresponding Wait Graph. To evaluate the values for $IA_{\text{run}}$, $IA_{\text{wait}}$, and $IA_{\text{opt}}$, the impact analysis introduces the following metrics that are calculated from all constructed Wait Graphs.

The total duration $D_{\text{scn}}$ defines the aggregated execution time of all scenario instances in the input trace streams. Given the Wait Graph for each scenario instance, the impact analysis evaluates $D_{\text{scn}}$ by adding up the time periods of top-level tracing events in the Wait Graph instance by instance.

The total wait duration $D_{\text{wait}}$ defines the aggregated wait time cost by the chosen components in waiting for others. To determine which components should be counted, the impact analysis uses the names of the chosen components to filter the topmost signatures on the callstacks of tracing events. The impact analysis uses a breadth-first search on the Wait Graphs, and adds up the time periods of only top-level wait events of the chosen components to avoid counting child events that constitute the time cost already counted from their parent events.

The total running duration $D_{\text{run}}$ defines the aggregated running time cost by the chosen components. Similar to $D_{\text{wait}}$, the impact analysis evaluates $D_{\text{run}}$ by counting duration of the running events that contain signatures of the chosen components. Note that (1) some portions of periods in $D_{\text{run}}$ are overlapped with $D_{\text{wait}}$ because the running events are mostly the leaf nodes of some wait events; (2) $D_{\text{run}}$ is an approximate value since ETW samples running events every millisecond and thus those running costs are at the millisecond granularity.

The total distinct-wait duration $D_{\text{wait dist}}$ defines the aggregated duration of distinct
wait events of the chosen components over all scenario instances. A wait event may be involved in multiple Wait Graphs for multiple scenario instances, indicating that (1) the enclosing instances are captured in a common or close period of time from the same deployment site, and (2) the event has performance impact on multiple instances. The impact analysis measures $D_{\text{waitdist}}$ by excluding the duration of duplicate events across different Wait Graphs from $D_{\text{wait}}$.

### 2.3.2.2 Output-Metric Derivation

The impact analysis derives the values of output metrics $IA_{\text{run}}$, $IA_{\text{wait}}$, and $IA_{\text{opt}}$ from the preceding metrics. In particular, $IA_{\text{run}} = D_{\text{run}}/D_{\text{scn}}$ represents the performance impact of running time, and $IA_{\text{wait}} = D_{\text{wait}}/D_{\text{scn}}$ represents the performance impact of wait time.

For deriving the severity of cost propagation across different scenario instances, there is $IA_{\text{opt}} = (D_{\text{wait}} - D_{\text{waitdist}})/D_{\text{scn}}$. $IA_{\text{opt}}$ comes from the observation that any performance impact involving the chosen components can affect multiple different scenario instances. Namely, $D_{\text{wait}}/D_{\text{waitdist}} > 1$, which describes that the $D_{\text{waitdist}}$ of duration actually causes $D_{\text{wait}}$ long wait among multiple scenario instances. The cost propagation is the underlying reason causing such performance impact. For instance, in the motivating example (Section 2.2.2), the delay initiated by $T_{S,W0}$ actually affected not only BrowserTabCreate, but also other scenario instances corresponding to two other applications along the propagation path. If we suppose the optimal case $D_{\text{scn}}^{\text{opt}}$ as removing cost propagation between irrelevant scenarios, the extra percentage of cost introduced by cost propagation to the entire system would be $1 - D_{\text{scn}}^{\text{opt}}/D_{\text{scn}} = (D_{\text{wait}} - D_{\text{waitdist}})/D_{\text{scn}}$. This value is also an upper bound for the optimization of cost propagation. However, the actual optimization depends on the root causes of performance problems and other factors.

### 2.4 Causality Analysis

If the results from impact analysis indicate that the chosen components are of high impact, performance analysts can use causality analysis to discover runtime behaviors that may cause the observed performance impacts.

We adapt contrast data mining [16] to achieve causality analysis. Contrast data mining is the mining of patterns contrasting two or more classes for knowledge discovery [16]. In the setting of performance analysis, there are few specifications that can be used to distinguish
Figure 2.2 An illustration of an Aggregated Wait Graph for device drivers.

between normal and problematic behaviors affecting performance, but we do know the difference on performance measurements. Contrast data mining can thus serve as a link from effect to cause. Namely, it allows us to discover behaviors that may cause the observed difference on measurements.

2.4.1 Data Abstraction and Pattern Representation

To adapt contrast data mining in performance analysis, we introduce the Signature Set Tuple as the pattern representation, and derive the Aggregated Wait Graph as the data abstraction from Wait Graphs. So the contrast data mining can discover contrast patterns in the form of Signature Set Tuple among different Aggregated Wait Graphs that are of distinguishable performance measurements.

2.4.1.1 Aggregated Wait Graph

An Aggregated Wait Graph is essentially the abstraction and aggregation of runtime behaviors in a set of Wait Graphs belonging to the same scenario, based on the fact that runtime behaviors in the same scenario represent the similar tasks under investigation. The definition of an Aggregated Wait Graph is as follows, in which the term signature specially denotes the topmost signature related to the chosen components on the callstack e.S of a tracing event e, if there exists such signature on the callstack.

Definition 2 An Aggregated Wait Graph is a graph $AWG = (V, E)$. $V$ is a set of nodes $V = \{v\}$, in which each node represents the aggregated execution of a function signature in one of
the following statuses: running, waiting, or hardware service. \( E \) is a set of directed edges \( E = \{ v_i \rightarrow v_j \} \), in which each edge must start from a waiting node, indicating that the pointed node performs its operation within the waiting cost of the starting node.

In addition, we attach some extra properties to the nodes in an Aggregated Wait Graph to ease our analysis.

**Definition 3** In an Aggregated Wait Graph, each running node has a signature \( v.r \); each hardware-service node has a dummy signature \( v.h \); and each waiting node has two signatures: a wait signature \( v.w \) and its paired unwait signature \( v.u \). Each node has a performance metric \( v.C \) as its duration of execution time, and has an occurrence counter \( v.N \) indicating the number of the same nodes from the source Wait Graphs.

Figure 2.2 shows a partial Aggregated Wait Graph for the motivating example in Section 2.2.2. The region highlighted with a dashed box shows an aggregated path that represents a group of paths of similar cost propagation from multiple scenario instances. The aggregated path captures cost propagation from a hardware service via the se.sys driver and then the fs.sys driver, finally to fi.sys.

### 2.4.1.2 Signature Set Tuple

Causality analysis presents mined patterns in the form of Signature Set Tuple. A Signature Set Tuple generalizes runtime interactions related to cost propagation into three signature sets: a wait-signature set, an unwait-signature set, and a running-signature set. The wait-signature set contains signatures that reside in wait events; namely, such signatures can cause the caller thread to be suspended. The unwait-signature set contains signatures from unwait events, which signal suspended threads to continue. The running-signature set contains signatures recorded in running events, or a dummy signature representing hardware service events. We extract Signature Set Tuples from path segments on Aggregated Wait Graphs. The definitions of the path segment and Signature Set Tuple are as follows.

**Definition 4** A path segment \( S \) of an Aggregated Wait Graph \( AWG \) is a sequence of nodes \( S = \langle v_i \rangle: v_0 v_1 \ldots v_{L-1}, \) where \( v_i \in AWG.V \) and for \( 0 \leq i < L-1, v_i \rightarrow v_{i+1} \in AWG.E \). The performance metric of \( S \) is defined as the metric of its end node, i.e., \( S.C := v_{L-1}.C \), as well as the occurrence counter \( S.N := v_{L-1}.N \).
**Definition 5** A Signature Set Tuple $P$ from a path segment $S$ is $P(S) := (\bigcup v.w, \bigcup v.u, \bigcup v.r)$, where $v \in S$. The performance metric of $P(S)$ is defined as the metric of the path segment, i.e., $P(S).C := S.C$, as well as the occurrence counter $P(S).N := S.N$.

We design such Signature Set Tuple based on two key rationales. First, the three signature sets appropriately generalize the phenomenon that the cost of some time-consuming operations (represented by the running-signature set) propagates to other parts in the system via interactions in the direction from functions represented by the unwait-signature set to those of the wait-signature set, regardless of the particular kinds of interactions. Second, signature sets can accommodate variations of the cost-propagation sequences. Consider a case of device drivers, in which two drivers contend a resource held by the third driver. Such case creates two possible execution sequences in reality, depending on which driver acquires the resource first. A pattern in the Signature Set Tuple can represent both possibilities.

### 2.4.2 Pattern Discovery

Given a set of instances of a particular scenario, and two performance thresholds $T_{fast}$ and $T_{slow}$ as inputs, the causality analysis adapts contrast data mining, and works in three steps: (1) classifying contrast classes, (2) constructing data abstractions, and (3) mining contrast patterns.

#### 2.4.2.1 Classifying Contrast Classes

The causality analysis first classifies these instances into two contrast classes $\{I\}_{fast}$ and $\{I\}_{slow}$ by comparing their recorded execution time against performance thresholds $T_{fast}$ and $T_{slow}$. In practice, developers need to explicitly specify the two thresholds for each application scenario as a part of performance specification. Specifically, $T_{fast}$ is the upper bound reflecting normal performance, and $T_{slow}$ is the lower bound of performance degradation. For example, the *BrowserTabCreate* scenario of a web browser should be completed within 300ms, and should not exceed 500ms, which typically makes users feel slow. In this case, $T_{fast} = 300\ m\ s$ and $T_{slow} = 500\ m\ s$, so the two classes would be *less-than-300ms* as the fast class and *more-than-500ms* as the slow class. The contrasts between the two classes are highly suspicious to be the constitution of performance problems. Typically we have $T_{slow} - T_{fast} \gg 0$, enabling little confusion between the two contrast classes.
Algorithm 1: The algorithm to aggregate Wait Graphs.

Input: a set of Wait Graphs \{WG\}, a set of component names \{C\}
Output: an Aggregated Wait Graph \(AWG\)

// Algorithm 1 works in four steps. For each Wait Graph, it first eliminates nodes that are irrelevant to the components to be analyzed from Wait Graphs (Lines 3 to 8). Then it merges paired wait/unwait nodes (Lines 9 to 12). After that it aggregates the processed Wait Graph to the Aggregated Wait Graph (Lines 13 to 14). Finally, we introduce a heuristic to reduce the size of the graph in Line 15. We describe the key points for each step as follows.

Eliminating Component-Irrelevant Nodes. A node is component-irrelevant if none of the signatures on the callstack belongs to the specified components \{C\}. From roots of the graph, the algorithm removes each component-irrelevant node, and promotes its child nodes as new roots. The algorithm repeats until all root nodes are component-relevant. The processed Wait Graph contains only behaviors initiated by the components in \{C\}.

Merging Wait/Unwait Nodes. The algorithm merges \textit{wait} events with corresponding
unwait events as well as their edges to create waiting nodes (MergeWaitUnwaitPair in Line 12). The result structure represents that the child nodes perform operations within the cost of the parent node. In fact, the Aggregated Wait Graph is essentially a forest in which all inner nodes must represent wait/unwait event pairs, whereas all leaf nodes must be of running events or hardware service events.

**Aggregating Wait Graphs.** The algorithm aggregates the processed Wait Graphs based on common prefixes of paths (MergeWithCommonSigPrefix in Line 14). Paths having common prefix indicate that they have the identical sequence of signatures on nodes along their prefixes. Common prefixes indicate that there are common behaviors starting at the beginning of executions, and subsequent behaviors may be diverse due to external or internal reasons, such as various inputs from higher-level applications, or lock contentions and different dependencies.

**Non-Optimizable Portions.** To help subsequent mining and eliminate data noises, the algorithm performs a reduction on Aggregated Wait Graphs (ReduceAWG in Line 15). Specifically, it prunes all structures in the pattern of a waiting node as a root pointing to a single hardware-service leaf. This pattern implies that the performance impact of hardware services is not propagated to other components. Developers usually do not have any chances to optimize such cases.

### 2.4.2.3 Mining Contrast Patterns

The algorithm to discover contrast patterns between the two Aggregated Wait Graphs works in the following three steps. To address challenges from both data complexity and computational complexity, we particularly introduce a step of meta-pattern enumeration.

**Enumerating Meta-Patterns.** Given the two Aggregated Wait Graphs for the two contrast classes, the causality analysis first enumerates meta-patterns in the form of Signature Set Tuples from path segments in the two graphs. Specifically, we introduce a constant $k$ to bound the maximum length of path segments in order to improve efficiency. Given a certain value of $k$, the causality analysis iteratively enumerates all path segments that have the lengths from 1 to $k$, and collects meta-patterns from these segments. Given two path segments having a common meta-pattern $P$, the causality analysis aggregates their $P.C$ and $P.N$ values. After this step, we have two groups of meta-patterns for the fast and slow contrast classes.

We have three considerations of enumerating meta-patterns from a bounded number
of path segments rather than from all possible path segments or full paths in the two Aggregated Wait Graphs. First, contrast mining on simple data abstractions such as sets is of high complexity [126]. For a complex data structure such as an Aggregated Wait Graph with a potentially large amount of data in it, the computation is challenging. Second, in fact meta-patterns collected from a bounded number of path segments are adequate without loss of patterns, compared with expensive all-segment enumeration. The causality analysis enumerates path segments from the smallest length; thereby, all other patterns could be the combinations of the smaller ones. Third, using patterns collected from complete paths is too specific for causality analysis. If a pattern of a complete path from one contrast class does not appear in another contrast class, we cannot claim that this path is a contrast, because its meta-patterns may appear frequently in both classes. So by conducting the segment enumeration, we can have these meta-patterns to exclude non-contrast paths.

**Discovering Meta-Pattern Contrasts.** The contrasts among meta-patterns in the two groups infer the behaviors having potential impacts to performance. There are two criteria to discover such contrasts. First, a meta-pattern appearing only in the slow class indicates that runtime behaviors represented by it may harm the whole performance to some extent, since such behaviors never happen in the cases of normal performance. The causality analysis selects such meta-pattern as a contrast. Second, a meta-pattern that is common in the two classes, but whose attached performance metrics are significantly higher in the slow class, also implies highly suspicious and high-impact runtime behaviors to the performance. In particular, for the common meta-pattern \( P_s \) in the slow class, and \( P_f \) in the fast class, whenever \( \frac{P_s.C}{P_f.C} / \frac{P_s.N}{P_f.N} > \frac{T_{slow}}{T_{fast}} \), the causality analysis selects the meta-pattern as a contrast.

**Discovering Contrast Patterns.** Finally, the causality analysis discovers contrast patterns based on the contrast meta-patterns in two steps. First, the causality analysis uses contrast meta-patterns as clues to select contrast paths. It computes a pattern in the Signature Set Tuple for each full path in the Aggregated Wait Graph of the slow class. If the extracted pattern contains any contrast meta-patterns, the causality analysis selects it as a contrast pattern. Second, the causality analysis merges identical contrast patterns with their \( P.C \) and \( P.N \) counters. Recall that multiple paths representing the same kind of problem could share the same pattern, even when their cost-propagation sequences might be different.

These contrast patterns are the output of causality analysis. To further facilitate manual
efforts on inspection, we rank these patterns by their performance impacts, with the highest impacts ranked the first. We define the impact of a contrast pattern by its average execution cost in performing its task, i.e., $P.C/P.N$.

2.5 Empirical Results and Evaluation on Device Drivers

We apply our approach to study the performance of device drivers in Windows. We first perform impact analysis on device drivers, and then present an evaluation of using causality analysis to reveal patterns of performance problems.

Our data set contains about 19,500 trace streams, consisting of about 505,500 scenario instances that fall into 1,364 usage scenarios of Windows and various applications. The total recorded execution time is approximately 339 hours. These trace streams are collected by the ETW infrastructure.

Our study and evaluation offer developers and performance analysts two main benefits. First, the study and evaluation uncover the impact and problems of device drivers in the real world. Second, the study and evaluation show that our approach can facilitate performance analysis by reducing manual efforts on the inspection of performance problems.

2.5.1 Results of Impact Analysis on Device Drivers

We set the impact analysis to analyze device drivers exclusively over all scenario instances in the data set, namely, by setting the names of components to include all device drivers (matching the wildcard pattern “*.sys” in all function signatures).

The impact analysis shows that the wait percentage $IA_{wait} \approx 36.4\%$, and running percentage $IA_{run} \approx 1.6\%$, which are compelling metrics to inform the performance impact. $IA_{wait}$ directly explains what percentage device drivers block application executions. We can consider it as the average percentage of performance impact introduced by device drivers in each instance. $IA_{run}$ indicates what percentage device drivers consume CPU time. The small percentage of $IA_{run}$ indicates that the running time of device drivers is not a dominating part in overall system performance.

The impact analysis also shows that the percentage of extra waiting time $IA_{opt} \approx 26\%$, indicating the severity of cost propagation in device drivers, and the fact that the total execution time $D_{scn}$ could be reducible by optimizing device drivers and/or relevant components. In addition, the ratio of the total wait duration to the total distinct-wait duration
\( D_{\text{wait}} / D_{\text{wait dist}} \approx 3.5 \) shows that any cost involving device drivers in one scenario instance is propagated to the other 2.5 instances on average, indicating the severity of cost propagation from a high-level of view.

In summary, the results indicate that device drivers are worth being further investigated. In particular, the results disclose three aspects. First, device drivers constitute a non-trivial part (\( \approx 36.4\% \)) in OS and application performance. Second, the percentage of time likely wasted due to cost propagation would be 26\%, which is also an upper bound for the optimization of cost propagation. Third, CPU consumption of device drivers has limited impact (\( \approx 1.6\% \)). It is consistent with the expectation that drivers do little computation [97], and thus their computation cost is not our focus of analysis.

### 2.5.2 Evaluation of Causality Analysis

Performance analysts can manually inspect output patterns to determine whether these patterns represent high-impact performance problems and need further actions towards optimization. To show the effectiveness of causality analysis, and how causality analysis can improve the efficiency of manual inspection, we address the following three research questions.

**RQ1:** To what extent would the discovered patterns explain the bad performance in device drivers?

**RQ2:** How does the ranking strategy applied on discovered patterns help improve efficiency of inspecting performance problems?

**RQ3:** What insights can the discovered patterns offer to performance analysts in identifying real performance problems?

#### 2.5.2.1 Evaluation Setup

We apply the causality analysis on 8 selected out of all 1,364 scenarios. There are 17,612 instances in the selected scenarios. Table 2.1 shows these scenarios. We choose these scenarios for two reasons: (1) the selected scenarios represent a set of typical applications (e.g., web browsers) that need to interact with multiple device drivers, and (2) the number of instances in the selected scenarios are relatively large to avoid data noises and biases affecting contrast data mining. There are 2,201 instances for each selected scenario on average, compared with the overall average of 370 instances per scenario in our data set.
Table 2.1 Selected scenarios for the evaluation of causality analysis.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>#Instances</th>
<th>in ({I}_{fast})</th>
<th>in ({I}_{slow})</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppAccessControl</td>
<td>1547</td>
<td>598</td>
<td>772</td>
</tr>
<tr>
<td>AppNonResponsive</td>
<td>631</td>
<td>164</td>
<td>392</td>
</tr>
<tr>
<td>BrowserFrameCreate</td>
<td>1304</td>
<td>437</td>
<td>707</td>
</tr>
<tr>
<td>BrowserTabClose</td>
<td>989</td>
<td>134</td>
<td>678</td>
</tr>
<tr>
<td>BrowserTabCreate</td>
<td>2491</td>
<td>597</td>
<td>1601</td>
</tr>
<tr>
<td>BrowserTabSwitch</td>
<td>2182</td>
<td>1122</td>
<td>914</td>
</tr>
<tr>
<td>MenuDisplay</td>
<td>743</td>
<td>171</td>
<td>499</td>
</tr>
<tr>
<td>WebPageNavigation</td>
<td>7725</td>
<td>4203</td>
<td>1175</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>17612</strong></td>
<td><strong>7426</strong></td>
<td><strong>6738</strong></td>
</tr>
</tbody>
</table>

Table 2.1 shows the selected scenarios, the number of scenario instances, and the numbers of instances in the two contrast classes. For confidentiality, we anonymize the real names of the selected scenarios, and use \(T_{fast}\) and \(T_{slow}\) to denote the two thresholds. These two thresholds are determined by application vendors, and may vary in different scenarios. We set the component names \(\{C\}\) to be all device drivers, and the maximum length of segment enumeration to be 5 in all experiments to focus on the propagation of performance impact up to that length.

To address RQ1, we first use an automated rule to determine whether the discovered patterns represent high-impact driver behaviors. In particular, we determine that a contrast pattern is of high impact if at least one of its executions in trace streams exceeds \(T_{slow}\). Such high-impact pattern indicates that the involved drivers must be able to constitute the bad performance as a significant factor. Otherwise, the pattern’s impact to the bad performance is relatively low. After classifying the discovered patterns by the automated rule, we show two execution-time coverages for the classified patterns over the total time cost of device drivers in the slow class. In particular, the impactful-time coverage (ITC) is the sum of the time cost \(P.C\) for the high-impact patterns over the total time cost of device drivers, and the total-time coverage (TTC) is the sum of the time cost \(P.C\) for all patterns over the total time cost of device drivers. The ITC suggests the scope where there must be high-impact driver behaviors causing the bad performance, and the TTC suggests the scope where there may be such behaviors. The ITC and TTC can show the effectiveness of causality analysis, because high coverages indicate that the discovered patterns can interpret the bad performance to a great extent, which offers a higher chance for performance analysts to find high-impact performance problems.
Table 2.2 Impactful-time and total-time coverages of discovered patterns on the total driver time.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Driver Cost</th>
<th>ITC</th>
<th>TTC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppAccessControl</td>
<td>66.4%</td>
<td>18.9%</td>
<td>35.5%</td>
</tr>
<tr>
<td>AppNonResponsive</td>
<td>64.6%</td>
<td>41.0%</td>
<td>48.7%</td>
</tr>
<tr>
<td>BrowserFrameCreate</td>
<td>76.5%</td>
<td>24.1%</td>
<td>35.4%</td>
</tr>
<tr>
<td>BrowserTabClose</td>
<td>21.9%</td>
<td>27.1%</td>
<td>38.0%</td>
</tr>
<tr>
<td>BrowserTabCreate</td>
<td>51.3%</td>
<td>23.1%</td>
<td>35.3%</td>
</tr>
<tr>
<td>BrowserTabSwitch</td>
<td>41.0%</td>
<td>7.8%</td>
<td>17.5%</td>
</tr>
<tr>
<td>MenuDisplay</td>
<td>77.0%</td>
<td>39.2%</td>
<td>49.2%</td>
</tr>
<tr>
<td>WebPageNavigation</td>
<td>34.7%</td>
<td>18.4%</td>
<td>28.5%</td>
</tr>
<tr>
<td>Average</td>
<td>54.2%</td>
<td>24.9%</td>
<td>36.0%</td>
</tr>
</tbody>
</table>

For RQ2, we show an execution-time coverage of top $n\%$ patterns based on the ranking strategy. In practice, a performance analyst needs to manually investigate the discovered patterns in trace streams to confirm the root causes, and prioritize the investigation efforts for high-impact patterns. So a relatively small $n\%$ with high coverage indicates that the ranking strategy prioritizes high-impact patterns properly, and can substantially improve the inspection efficiency.

For RQ3, we first present what kinds of drivers are involved in the top-10 patterns from each selected scenario (80 patterns in total). Then we describe three major observations that we can make from the relations between scenarios and drivers. We also use some representative cases to illustrate how causality analysis can help performance analysts identify real-world problems. Note that the analysis in this section was conducted independently by the authors, but all cases described in detail have been reviewed and confirmed by performance analysts and developers at Microsoft. We are continuing the confirmation process for other cases.

2.5.2.2 RQ1: Effectiveness

We show the effectiveness of causality analysis by presenting the impactful-time and total-time coverages for the discovered patterns. Table 2.2 shows the results of the two coverages. The column “Driver Cost” presents total execution time of device drivers in each scenario. The columns “ITC” and “TTC” present the impactful-time and the total-time coverages on the total driver time.

Overall, the total driver cost in the slow class can be characterized by three portions.
The first portion is represented by the driver behaviors that may account for the bad performance. The second one is represented by the driver behaviors that are likely of normal performance. The two portions do not have an easily distinguishable boundary. The causality analysis can help performance analysts identify the boundary. In particular, the contrast patterns that are counted in the ITC must fall into the first portion, whereas the rest of the contrast patterns may cross the boundary of the two portions, but still likely cause the bad performance. Let us take the AppNonResponsive scenario as an example. The total driver cost covers 64.6% of the total execution time. The ITC is 41.0%, and the difference between the ITC and the TTC is 7.7%. The contrast patterns counted in the ITC should be the first priority for performance analysts to inspect, and the rest of the patterns falling between the ITC and the TTC reflect a non-negligible amount of execution time that performance analysts should inspect to confirm potential problems.

On the other hand, the third portion is represented by the driver behaviors that directly interact with hardware without cost propagation. These behaviors are usually of high impact, but non-optimizable, since the real cost depends on external hardware. During the construction of an Aggregated Wait Graph, the reduction of non-optimizable portions would remove such behaviors, even though some of them could account for long-time executions. Since device drivers frequently interact with slow hardware, the non-optimizable portions are very common. For example, in the BrowserTabSwitch scenario, 66.6% of the total driver cost is manifested in direct hardware services without cost propagation, thus being removed from the Aggregated Driver Graph. The resulting graph represents the remaining 33.4%, and more than half of the remaining portions (17.5%) are represented by contrast patterns, and classified into the total-time coverage.

2.5.2.3 RQ2: Efficiency

We calculate another execution-time coverage to show the efficiency improvement brought by the causality analysis on manually confirming potential performance problems. Based on the ranking strategy used in the pattern discovery, we select the top 10%, 20%, and 30% contrast patterns, and measure their execution-time coverages over all discovered patterns. Table 2.3 shows the results, in which the column “#Patterns” presents the number of contrast patterns in each scenario, and the last three columns show the coverage for the top 10%, 20%, and 30% contrast patterns.

The numbers of the output patterns range from 1,075 to 5,122 across different scenarios.
Table 2.3 Coverages by top 10%, 20%, and 30% of the ranked patterns.

<table>
<thead>
<tr>
<th>Scenario (T_{slow})</th>
<th>#Patterns</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppAccessControl</td>
<td>4875</td>
<td>55.3%</td>
<td>91.1%</td>
<td>98.3%</td>
</tr>
<tr>
<td>AppNonResponsive</td>
<td>1158</td>
<td>29.6%</td>
<td>39.2%</td>
<td>95.1%</td>
</tr>
<tr>
<td>BrowserFrameCreate</td>
<td>1933</td>
<td>51.6%</td>
<td>92.0%</td>
<td>96.8%</td>
</tr>
<tr>
<td>BrowserTabClose</td>
<td>1075</td>
<td>55.1%</td>
<td>90.0%</td>
<td>93.5%</td>
</tr>
<tr>
<td>BrowserTabCreate</td>
<td>5045</td>
<td>49.0%</td>
<td>87.5%</td>
<td>97.0%</td>
</tr>
<tr>
<td>BrowserTabSwitch</td>
<td>1514</td>
<td>42.3%</td>
<td>64.9%</td>
<td>98.0%</td>
</tr>
<tr>
<td>MenuDisplay</td>
<td>1855</td>
<td>64.5%</td>
<td>86.5%</td>
<td>91.9%</td>
</tr>
<tr>
<td>WebPageNavigation</td>
<td>5122</td>
<td>35.6%</td>
<td>89.3%</td>
<td>96.5%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>2822</strong></td>
<td><strong>47.9%</strong></td>
<td><strong>80.1%</strong></td>
<td><strong>95.9%</strong></td>
</tr>
</tbody>
</table>

Table 2.4 Drivers involved in top-10 patterns of each scenario categorized by driver types.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AppAccessControl</td>
<td>9</td>
<td>9</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>AppNonResponsive</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>BrowserFrameCreate</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>2</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>BrowserTabClose</td>
<td>5</td>
<td>6</td>
<td>–</td>
<td>2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>BrowserTabCreate</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>BrowserTabSwitch</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>MenuDisplay</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>–</td>
<td>2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>WebPageNavigation</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

The thousand magnitude of patterns came from the scale of input traces and the high complexity of the OS ecosystem, especially the complexity of interactions and dependencies in device drivers. As shown in Table 2.3, the inspection of the top 10% patterns would involve 107 to 512 patterns and achieve 30% to 65% execution-time coverage. According to the evaluations provided for StackMine [49], a performance analyst could inspect the top 400 stack-trace patterns of an analysis within 8 hours to achieve 60% execution-time coverage in all analyzed regions of scenario instances, with over 90% inspection effort saved as an estimation. Such comparable experiences indicate the affordability and efficiency regarding inspection efforts required by the causality analysis as well.

2.5.2.4 RQ3: Real Cases

Table 2.4 shows that in each scenario what kinds of drivers are involved in top-10 patterns. Each cell shows the number of patterns containing the corresponding type of drivers. From Table 2.4, we have three major observations regarding the relations between scenarios and drivers. These observations can suggest proper starting points for the performance analysis.
on similar cases.

First, most patterns contain both file-system drivers and filter drivers, especially in the \textit{AppAccessControl} scenario. In fact, most of those filter drivers belong to security software. Security software uses system-wide filter drivers to intercept various application requests, but usually uses a single process and database for security inspection. Such architecture makes the performance of security inspection very critical. Once the workload of the system increases, it is easy to form bottlenecks starting from those filter drivers. If the system also enables storage encryption, the situation could become worse, as the encryption drivers could induce extra overheads. The example described in Section 2.2.2 is a typical case of how filter drivers combined with encryption drivers slow down the system. Developers should estimate the granularity and potential contentions of locks in filter drivers to alleviate overheads.

Second, some scenarios are especially vulnerable to particular types of drivers. In addition to the \textit{AppAccessControl} scenario, which suffers from file-system drivers and filter drivers, we can observe that network drivers take a big part in the scenario of \textit{MenuDisplay} (7 out of 10). It is understandable that network drivers can be delayed by unstable bandwidth. So if a menu needs to display items from remote servers, developers should take into account such instability, and use an asynchronous mechanism or prefetched cache to prevent network delays from being propagated to user interfaces.

Third, hard faults can establish more subtler interactions between drivers, and cause severe performance degradation. In the \textit{AppNonResponsive} scenario, an interesting pattern shows that a graphics driver \textit{graphics.sys} appears with a file system driver \textit{fs.sys} and a storage encryption driver \textit{se.sys}. From driver signatures we can infer that \textit{graphics.sys} should not have interactions with the other two in normal cases, because it does not need to access files on disk. Based on our knowledge, it is highly suspicious that this pattern hints a hard fault occurring in \textit{graphics.sys}.

From trace streams, we find a corresponding instance that costs about 4.73 seconds to finish the execution. To simply describe this case, we next describe only three threads that are most relevant. The initiating one was a UI thread \(T_{U,U1}\). \(T_{U,U1}\) was executing \textit{graphics.sys}, and the driver tried to acquire resources of a Graphics Processing Unit (GPU). At the same time, there was a system worker thread \(T_{S,W0}\) in which \textit{graphics.sys} was executing a routine in response to a system event. Since \(T_{S,W0}\) had acquired GPU resources, \(T_{U,U1}\) had to wait. At this point, a hard fault occurred in \(T_{S,W0}\) when \textit{graphics.sys} initialized an internal structure.
To solve the hard fault, the system scheduled another worker thread $T_{S,W1}$ to perform a page read. $T_{S,W1}$ was actually executing se.sys because the system was storage-encrypted. Finally, the system spent about 4.7 seconds to complete the page read. This significant performance degradation was then spread to $T_{U,U1}$, and made the user interface unresponsive.

The preceding case indicates that solving hard faults could be very time-consuming, so device drivers should be developed to minimize the usage of paged memory in order to avoid expensive disk I/O and the consequence of potential cost propagation.

### 2.5.2.5 Validity and False Positives in Analysis Results

To validate the analysis supported by our approach, we sent the case described in Section 2.2.2, a representative case of security software, and the hard fault in graphics.sys as high-impact problems to relevant performance analysts and developers at Microsoft, and received their confirmations. The high impact was partially confirmed after their frequent observation of such problems, reflecting the limitations of the conventional performance analysis. One developer from a Windows product team for driver quality provided his enthusiastically positive feedback: “...It is very impressive to see how performance issues of device drivers can be root caused in such a detail... yes, this is the technique we are interested in w.r.t. driver qualities...”

Except those high-impact real cases, we can observe some false positives. In some special circumstances, our approach fails to distinguish between by-design behaviors and problematic behaviors. For instance, in Table 2.4 there is a type of drivers called “Disk Protection,” which can halt hard drives to prevent damages when a computer is in motion. Drivers of this type are designed to block all disk reads and writes when necessary, and meanwhile performance can be compromised. The appearance of such patterns suggests that we need to incorporate such knowledge to filter out known and exceptional cases.

### 2.6 Related Work

#### 2.6.1 Performance Analysis

The high-level goal of our work is to help with the performance analysis of software ecosystems, particularly the evaluation of impacts and identification of root causes for performance degradation, by acquiring the knowledge from a large number of execution traces.
We organize the discussion of related work by three major aspects, specifically, (1) handling cross-component interactions, (2) leveraging multiple executions, and (3) oracles for finding root causes.

Cross-component interactions may work concurrently and cause the effect of cost propagation. The potential transparency of these interactions to individual components makes the diagnosis of performance to be even harder. Some existing work [4, 114] identified the idleness as a performance characteristic of multithreaded and multi-component programs. Tallent et al. [114, 115] analyzed lock-based interactions as the source of such idleness. However, their work is limited to isolate the effect of a single lock only. In contrast, our work reveals the combinatorial effect of multiple locks connected by hierarchical dependencies, which commonly exist in large and complex systems.

Multiple executions of a system can provide diverse performance information. StackMine [49] and Magpie [19] also work on multiple ETW traces. Our previous StackMine work discovers callstack patterns via costly-pattern mining, resulting in patterns capturing within-thread behaviors. Our current work complements it with the mining of contrast patterns characterizing cross-thread behaviors. Magpie correlates workloads with events synthesized from traces to build performance models that support performance prediction and debugging. Our work can further help Magpie diagnose root causes by using workload-based scenarios. Jovic et al. [60] targeted at finding and prioritizing performance bugs in GUI applications from multiple profiles in the wild. However, this work relies on the manual selection of landmark methods to measure latency, and has the limitation of not being able to handle concurrency.

Using proper oracles is helpful to find performance problems and root causes effectively. Rule-based problem identification [4, 7, 56, 128] is limited due to finding only problems defined by existing rules. Some work performs thresholding and filtering on performance measurements [109] combined with frequencies of observed system behaviors [2]. However, in complex systems, no clear-cut criteria can be used to attribute bad performance to behaviors that may be expensive in nature and vary in different usage scenarios. Our work leverages information from the labeled normal executions as the criteria addressing such variations. Some work [15, 48, 104] proposed similar ideas exploiting contrasts between different executions to identify performance problems. Compared to such previous work, our work does not require concrete inputs of systems. These inputs are not always available especially when performance data are collected from client sites. Furthermore, our work
provides useful data abstractions that group similar behaviors in a large volume of data, and provides a pattern representation to represent actionable patterns to further narrow down the scope of the root-cause localization.

### 2.6.2 Quality Assurance for Device Drivers

Quality of device drivers includes mainly reliability, security, and performance. Most existing research work is to improve driver reliability [17, 18, 45, 46, 63, 69, 72, 96–99, 106, 112, 113]. While certain kinds of reliability issues are handled, e.g., memory corruption, the security is also improved. In addition, isolation-based approaches [29, 106] were also introduced to improve driver security. In contrast, little work has been proposed on driver performance. In fact, there is a trade-off between reliability (together with security) and performance. Leslie et al. [70] and Ganapathy et al. [46] demonstrated that it is possible to build user-level drivers to improve reliability without significant performance overheads. Menon et al. [85] explored the opportunity to balance reliability, security, and performance with a framework allowing to create safe and efficient hypervisor drivers. Huang and Chen [51] developed an approach to find performance bottlenecks of TCP/IP protocol and the corresponding drivers. Such work usually addresses driver performance by a specific analysis or from a specific perspective, but not from a general perspective of real-world production ecosystems.
3.1 Introduction

With the fast growth of the global cloud-computing market, many dedicated software infrastructures have emerged to provide convenient access to cloud-based computing, storage, and networking resources. For example, Amazon Elastic Compute Cloud (EC2) \cite{6} and Microsoft Azure \cite{86} are two widely used public cloud infrastructures that enable users to easily create computing platforms in the cloud with multiple servers and configurable resources. In the open-source community, OpenStack \cite{91} has been gaining popularity steadily in recent years \cite{1}.

Cloud infrastructures provide access to cloud resources via a series of management tasks, e.g., tasks enabling users to spawn virtual machines (VMs), stop VMs, and delete
VMs. To execute a task, cloud infrastructures coordinate multiple internal service processes distributed on different server nodes, each of which takes part in the whole execution (e.g., a scheduler to assign VMs to different nodes and hypervisors on assigned nodes to boot up VMs). The distributed and multi-process nature of task executions introduces extra complexity and non-determinism, which can sometimes result in subtle problems.

From the perspective of cloud providers, monitoring is an important way to help the identification and diagnosis of execution problems that compromise cloud availability. The provider-side administrators expect to know through monitoring if each task execution completes successfully and in time. When execution problems occur, administrators also expect enough information from monitoring that can guide the diagnosis. However, cloud infrastructures usually do not expose enough execution details. De facto practices include checking the outcomes of test requests [78] and monitoring running statuses of service processes and resource usages to detect abnormalities. Unfortunately, test requests may fail to exercise and expose problems in specific problematic execution paths, and resource usages may not reveal problems that do not cause usage irregularities (e.g., expected inter-service messages not sent or received). As a result, revealing internal execution steps across service processes for direct monitoring in the workflow level is challenging.

Monitoring logs is another common practice to find and diagnose problems. For example, Amazon CloudWatch [5] provides a log-streaming interface for real-time log monitoring on VM instances. There are also open-source projects, such as Logstash [79], to support a similar feature for centralizing distributed logs from multiple server nodes to a single location. However, it is tedious and labor-intensive to manually monitor and understand numerous distributed and interleaved log messages by human effort. In a real-world banking system, that some of the authors had experiences with, there were 200 full-time operators dedicated to 24x7 log monitoring with 67 screens for 190 subsystems.

To facilitate monitoring through logs, we present a non-intrusive lightweight approach for workflow monitoring. Our prototype for OpenStack, CloudSEER, purely works on logs that are readily available in existing cloud infrastructures. CloudSEER is backed by the observations that logging is a general practice for complex software systems and logs implicitly carry workflow information. In these logs, not only error messages, but also sequence-based information, can help monitor and discover execution problems. To enable log-based workflow monitoring, CloudSEER addresses the following key challenges.

**Working with interleaved log sequences.** When a cloud infrastructure is executing mul-
multiple tasks in parallel, log messages from different executions are usually interleaved in log files. Even in a single execution, asynchronous operations among different services can interleave log messages in different ways. To recover workflow information, it is necessary, but not straightforward, to identify the correspondence between log messages and task executions. Some existing work [43] assumes that messages have unique request and thread identifiers, which can distinguish messages from different task executions. However, such an assumption does not always hold. While working with OpenStack, we found that there could be multiple non-unique identifiers representing different entities (e.g., VMs, physical machines, and networks) in different log messages. Only the combinations of some identifiers may identify sequences of log messages that belong to different task executions.

**Working in an online scenario.** The sooner a problem is noticed, the quicker can it be fixed and the more can its effects be mitigated. Existing work [65, 82, 131] on detecting problems using data mining and clustering on logs can only operate in an offline manner, since they require task or program executions to finish so that logs are available in their entirety beforehand. Such a design delays problem detection and is more problematic if the executions may not finish at all. In the monitoring scenario, the detection of execution problems should be quick without the need to wait for the tasks to finish.

**Detecting problems without explicit error messages.** Execution problems can manifest in different forms. Some of them may not be easily noticeable, e.g., performance degradation and silent failures that do not come with explicit error messages. From the logging practices in OpenStack [91], we found that failures were not always accompanied with indicative error messages. A recent study [140] on logging practice also reports that over half of the studied failures were not properly logged. Thus, only looking for error messages might not reveal subtle execution problems.

**Providing workflow information.** As a monitoring approach, just reporting a failure is not enough for administrators to diagnose the failure and take further actions. The context of a failure, particularly, workflow [23, 81], is useful for guiding administrators to narrow down and isolate possible causes. A monitoring approach should leverage log messages to include such information in failure reports.

**CLOUDSEER** addresses these challenges by combining an offline modeling stage (Section 3.3) and an online checking stage (Section 3.4). In the offline modeling stage, CLOUDSEER builds an automaton for each task via the logs generated by multiple correct executions. The result automaton depicts the task workflow, in which state transitions capture tempo-
ral dependencies between log messages. We introduce this automaton to serve two roles. **CLOUDSEER** uses it as a specification to identify log sequences in the online checking stage. The automaton is also a bookkeeper to keep track of the execution progress or context on-the-fly by maintaining automaton states. The bookkeeping enables the automaton to be used as an informative output that provides context information when **CLOUDSEER** detects a potential problem.

In the online checking stage, **CLOUDSEER** works on a log stream that consists of interleaved log messages from distributed services. It uses the pre-built automata to identify individual log sequences out of interleaved ones, and check them for divergences that may indicate execution problems.

We define two divergence criteria for two symptoms of execution problems: (1) error messages and (2) expected log messages not appearing within a time interval. While the first case is straightforward as it represents failures that are properly caught and handled by a system, the second case indicates unhandled silent failures that are not accompanied by error messages (e.g., when a key service stops responding) or performance degradation. In this work, we do not distinguish between silent failures and performance degradation. Our goal is to show that **CLOUDSEER** is capable of detecting different kinds of execution problems, rather than directly identifying the root causes of the detected problems.

To ensure efficiency for monitoring via interleaved logs, we design effective heuristics that leverage non-unique message identifiers to group related log messages into growing sequences on-the-fly. Based on the heuristics, **CLOUDSEER** associates a limited number of automaton instances with each sequence and checks if the sequence followed by a new message still conforms with its associated automaton instances. As a result, **CLOUDSEER** is able to avoid the brute-force checking on each message against all automaton instances, thus significantly reducing performance overhead.

During the checking process, **CLOUDSEER** marks and reports automaton instances that satisfy the divergence criteria. The reported instances provide workflow information about the task and the step within the task where the problem is occurring. Administrators can then use the provided information to understand and mitigate ongoing problems.

We evaluate **CLOUDSEER** by a series of experiments on primitive VM tasks of OpenStack (Section 3.5). The experiments evaluate how accurate and efficient **CLOUDSEER** checks interleaved log sequences and its capability of detecting some injected common problems. The results show that the checking accuracy of **CLOUDSEER** is at least 92.08% on interleaved
logs, with a satisfactory efficiency (an average of 2.36s/1k-messages in our test bed), making it suitable for the monitoring scenario. The results further show CLOUDSEER’s problem-detection capability with a precision of 83.08% and a recall of 90.00% on the injected problems.

3.2 Overview

We start with a general description about a representative cloud infrastructure, OpenStack, and a common set of logging characteristics that exist in similar systems. Then, we use examples based on OpenStack to briefly describe how CLOUDSEER monitors task executions through logs.

3.2.1 Cloud-Infrastructure Example: OpenStack

OpenStack is an open-source cloud-infrastructure platform. It consists of multiple service components that manage computing, storage, and networking resources in the cloud. For example, the component nova provides a set of tasks to manage VMs. Cloud users can issue task commands, e.g., “nova boot” and “nova stop” from the command-line interface (CLI) to boot up and stop VMs.

The execution of a task involves coordination between multiple service components. Figure 3.1 depicts how service components interact during the booting of a VM. When a user executes a task with the “nova boot” command, a request is sent to nova-api, which is a web service built using Web Server Gateway Interface (WSGI). The authentication service, keystone, authenticates the incoming request. If authenticated, the scheduling service, nova-scheduler, selects a compute server on which to boot up a new VM. The

![Figure 3.1 Inter-service interactions when booting a VM.](image-url)
compute service, nova-compute, running on the compute server, obtains the appropriate VM image from glance, which provides VM images. The VM hypervisor then takes over the request from nova-compute to launch the VM. The proxy service, nova-conductor, records states of VMs in the database and manages remote procedure calls (RPCs) across service boundaries using Advanced Message Queuing Protocol (AMQP).

### 3.2.2 Logging Characteristics

We are aware of two logging characteristics that make the log-based workflow monitoring not straightforward. The two characteristics are the manifestation of a set of underlying system behaviors, which are common in similar systems and not specific to OpenStack examples.

First, besides the message interleaving caused by simultaneous task executions, log messages from the same execution might also be interleaved. This is caused by asynchronous operations (e.g., AMQP communications in OpenStack), which create multiple execution paths through different service processes. As a result, log messages from different paths could be interleaved without a fixed ordering, making task workflows more complicated to monitor.

Second, there is often no single and unique identifier associated with log messages related to a task execution. In cloud infrastructures, each service manages its own resources (e.g., users, VMs, and images are managed by different services) and logs them with identifiers based on resource types. Those identifiers may not be propagated across different services during a task execution, because different services do not share the knowledge on different resources. For instance, a hypervisor knows which VM is running, but it has no knowledge about the user whom the VM belongs to. Consequently, different log messages may not share common identifiers.

We use an example in OpenStack to further illustrate the two characteristics. Figure 3.2 shows simplified log messages for two executions of the “nova boot” task. To ease the explanation, we simplify messages by substituting timestamps with numbers at the beginning of the messages, removing logging levels, replacing identifiers (IPs, URLs, and UUIDs) with simple text (e.g., IP1 for an actual IP address), and shortening text in the messages (e.g., api for nova-api). These identifiers indicate different resources, e.g., the identifiers in square brackets indicate users who initiate the tasks, the identifiers preceded by the text “Instance” indicate VMs, and others in the message bodies may indicate tenants or VMs.
In Figure 3.2, there are two interleaved log sequences, $1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 11$ and $2 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 10 \rightarrow 12$. Messages $8 \rightarrow 9$ in the first sequence and $7 \rightarrow 10$ in the second one show two different orderings for one pair of log messages. This is because the service nova-scheduler, which generates messages 5 and 6, uses asynchronous AMQP communications to schedule VM instances $UUID5$ and $UUID6$ to compute nodes. Such asynchronicity causes nova-api (generating messages 7 and 9) and nova-compute (generating messages 8 and 10) to execute their parts of the task in parallel, resulting in two possible orderings.

It is also noticeable how diversified the log identifiers are. For instance, messages 1 and 5 from the same sequence do not have any common identifiers. On the other hand, many a time some messages can connect other seemingly disconnected ones. For instance, message 3 shares $IP1$ and $UUID1$ with messages 1 and 5, respectively. It can effectively establish the relation between messages 1 and 5. This observation suggests that shared identifiers, regardless of how far they are propagated across services, can create a chain that transitively connects messages together. This observation thus enables a heuristic that separates interleaved log sequences.

### 3.2.3 Demonstration of CloudSeer

We show the basic idea of CloudSeer using the example in Figure 3.2. We assume that we have already built a task automaton using logs collected from multiple normal executions of booting VMs. Figure 3.3 shows the automaton for the “nova boot” task, and it is simplified
accordingly as what we have done for the log sequence in Figure 3.2. Section 3.3 describes the modeling step in detail.

**Task automaton.** The automaton in Figure 3.3 represents the booting-VM workflow by encoding the dependencies among log messages. The automaton takes message templates as inputs. The message templates are log messages without variable parts. Determining the variable parts requires certain domain knowledge. We consider IP addresses, UUIDs, and numbers as the variable parts in log messages.

To model interleaved messages caused by asynchronous operations, we introduce two special types of states in addition to the conventional automaton states: the *fork* states (q3 in Figure 3.3) and the *join* states (q6). A *fork* state has multiple outbound transitions for log messages that do not have dependencies with each other but share the same precedent, and each outbound transition transits to a new state from the fork state (e.g., q4 and q5 are forked from q3 after consuming the messages “GET” and “Starting”). A *join* state has multiple inbound transitions for a log message that must always be followed by some other messages, and all inbound transitions transit to the same join state (e.g., q4 and q5 join to q6 after consuming the message “Spawned”).

**Checking interleaved sequences.** CLOUDSEER takes one log message at a time. It checks if the message belongs to a growing log sequence and if the growing sequence with the new message still conforms with its corresponding automaton. This checking process continues until a message completes a sequence or satisfies divergence criteria.

To leverage multiple identifiers to determine the association of a message with a developing log sequence, CLOUDSEER uses a data structure, *identifier set*, as the signature of each
Table 3.1 Demonstration of CLOUDSEER’s checking process.

<table>
<thead>
<tr>
<th>Message</th>
<th>Identifier Set</th>
<th>Instance Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Accepted</td>
<td>1: {IP1}</td>
<td>1: {q0} → {q1}</td>
</tr>
<tr>
<td>(2) Accepted</td>
<td>2: {IP2}</td>
<td>2: {q0} → {q1}</td>
</tr>
<tr>
<td>(3) POST</td>
<td>1: {IP1, UUID1/2}</td>
<td>1: {q1} → {q2}</td>
</tr>
<tr>
<td>(4) POST</td>
<td>2: {IP2, UUID3/4}</td>
<td>2: {q1} → {q2}</td>
</tr>
<tr>
<td>(5) Scheduling</td>
<td>1: {IP1, UUID1/2/5}</td>
<td>1: {q2} → {q3}</td>
</tr>
<tr>
<td>(6) Scheduling</td>
<td>2: {IP2, UUID3/4/6}</td>
<td>2: {q2} → {q3}</td>
</tr>
<tr>
<td>(7) GET</td>
<td>2: {IP2, UUID3/4/6}</td>
<td>2: {q3} → {q3, q4}</td>
</tr>
<tr>
<td>(8) Starting</td>
<td>1: {IP1, UUID1/2/5}</td>
<td>1: {q3} → {q3, q5}</td>
</tr>
<tr>
<td>(9) GET</td>
<td>1: {IP1, UUID1/2/5}</td>
<td>1: {q3, q5} → {q4, q5}</td>
</tr>
<tr>
<td>(10) Starting</td>
<td>2: {IP2, UUID3/4/6}</td>
<td>2: {q4, q4} → {q4, q5}</td>
</tr>
<tr>
<td>(11) Spawned</td>
<td>1: {IP1, UUID1/2/5}</td>
<td>1: {q4, q5} → {q6}</td>
</tr>
<tr>
<td>(12) Spawned</td>
<td>2: {IP2, UUID3/4/6}</td>
<td>2: {q4, q5} → {q6}</td>
</tr>
</tbody>
</table>

log sequence. An identifier set stores identifiers appearing in all messages of a sequence.

Table 3.1 shows the process of CLOUDSEER checking log messages. The column “Identifier Set” presents the identifier set after processing the corresponding message shown in the column “Message.” Due to space limit, we use the symbol UUID1/2 to represent UUID1 and UUID2 in the column “Identifier Set.” The column “Instance Transition” shows the state transitions of automaton instances. The label before each identifier set and state transition indicates the log sequence under checking.

CLOUDSEER considers messages 1 and 2 to be the beginning of two new sequences because their identifier sets do not have any common element. Therefore, CLOUDSEER creates a new identifier set and a new automaton instance for each of these two messages.

The identifiers in message 3 have a common element with the identifier set of the first sequence ({IP1}) only. Therefore, CLOUDSEER associates the message with the sequence “1.” It then tests if the message causes the corresponding automaton instance to make a state transition. As the transition happens, CLOUDSEER extends the identifier set of the sequence with the UUIDs in the message. When message 4 arrives, CLOUDSEER associates the message with sequence “2” since the IP address IP2 appears in the current identifier set for that sequence. For the rest of the messages, CLOUDSEER repeats this checking process until the two automaton instances accept their log sequences.

Interpreting results. When an incoming message, or lack thereof, satisfies divergence criteria, CLOUDSEER is able to report the diverged automaton instance indicating a prob-
lematic execution. For investigation, the diverged instance provides useful information that includes the ongoing task with its normal workflow, and the execution point where the problem occurs in the form of the current automaton state(s) and all log messages consumed so far.

For example, in Figure 3.2, if a problem occurs in the compute node assigned to run the VM UUID 5, the message 8 or 11 may not appear. Such a case is detectable by a “timeout” criterion that specifies a time period in which an automaton instance must consume a message. CLOUDSEER would then report an instance with the states \(\{q_3, q_4\}\) or \(\{q_4, q_5\}\), depending on if the message 8 has appeared. Then, a knowledgeable administrator may quickly narrow down the problem to immediately before booting the VM (if \(\{q_3, q_4\}\), network could be the main cause) or during the booting process (if \(\{q_4, q_5\}\), I/O or hypervisor could be the main cause).

### 3.3 Modeling Task-Based Log Sequences

The offline modeling process creates automata from logs generated by correct task executions. CLOUDSEER uses the output automata to check interleaved logs in the online scenario. For each task to model, the modeling process takes log sequences from multiple executions of the task. The term log sequence denotes all log messages ordered by their timestamps (or in the order they arrive) from a single execution. Since asynchronous operations among services may cause some messages to be interleaved in different ways, using multiple log sequences of a task helps the modeling process reconstruct temporal dependencies, which describe the order relation of log messages to represent the task workflow.

The modeling process consists of three steps: (1) preprocessing log sequences to extract key log messages for the task workflow; (2) mining temporal dependencies in the preprocessed log sequences; and (3) constructing automata from the mined temporal dependencies.

#### 3.3.1 Preprocessing

The preprocessing step takes log sequences as input. A log sequence \(LS\) is a vector of log messages \(\langle m_1, m_2, ..., m_n \rangle\). Each log message \(m_i\) in \(LS\) has two attributes: a template \(t\) representing the constant text of \(m_i\), and a value set \(S_v\) containing values in the variable parts of \(m_i\). There are different ways to determine the template \(t\) and the value set \(S_v\) of
a log message \( m \), and we consider numbers, IP addresses, and UUIDs as variable parts, where UUID is in the form of a well-formatted string. We use regular expressions to match and extract them to create \( m.S_v \), and \( m.t \) is naturally the part of \( m \) after the extraction.

For a set of log sequences \( S_{LS} \) collected from multiple executions of a task, the preprocessing step keeps key log messages in each log sequence of \( S_{LS} \) and removes all other messages which are considered less relevant to the task workflow. To determine if a log message \( m \) is a key message, this step counts the appearance of \( m.t \) in each log sequence of \( S_{LS} \). The message \( m \) is a key message if \( m.t \) appears the same number of times in every sequence.

The preprocessing provides two benefits. First, it removes irregular log messages from loops, conditional choices, and periodical background tasks. Such messages are irrelevant to the task workflow and would introduce excessive possibilities when CLOUDSEER checks the interleaved log sequences. We consider that keeping only key messages is sufficient and efficient for the monitoring purpose. Second, the result sequences bring out in-sequence message interleaving caused by asynchronous operations. Capturing and modeling such interleaving is important for recovering the task workflow.

### 3.3.2 Mining Temporal Dependencies

Given a set of preprocessed log sequences \( S_{LS} \), *temporal dependencies* describe the order relation in which collaborative services generate log messages. We define the following two types of temporal dependencies over log templates \( \{t_1, t_2, \ldots, t_n\} \) shared by sequences in \( S_{LS} \): (1) a strong dependency \( t_i \xrightarrow{strong} t_j \) iff \( t_j \) is always next to \( t_i \) for each of their occurrences in \( S_{LS} \); (2) a weak dependency \( t_i \xrightarrow{weak} t_j \) iff \( t_i \) always happens before \( t_j \) in \( S_{LS} \), but it is not necessarily followed by \( t_j \) immediately. Any pair of \( t_i \) and \( t_j \) not having the two dependencies is considered to not have any specific order. For example, in Figure 3.2, the messages “accepted” and “POST” have a strong dependency, since both the occurrences 1 → 3 and 2 → 4 follow one specific order. On the other hand, the message “Scheduling” (5 and 6) always leads “Starting” and “GET” (5 → 8 → 9 and 6 → 7 → 10), but the latter two messages do not follow a specific order. Therefore, the messages “Starting” and “GET” have weak dependencies with “Scheduling.” The weak dependencies imply that the execution becomes asynchronous after the message “Scheduling” and before “Starting” and “GET.”

To mine temporal dependencies, this step consists of three parts. It first enumerates and
collects all pairs of log templates for each sequence in \( S_{LS} \). For example, given a sequence \( \langle a, b, c \rangle \), this step enumerates pairs of \( (a, b), (a, c), (b, c) \). Then it classifies all collected pairs into strong and weak temporal dependencies based on their definition. Finally, it performs a reduction of transitive relations on the classified temporal dependencies. The transitive relations are the result of enumerating all message pairs. In particular, for any dependencies \( a \rightarrow b \) and \( b \rightarrow c \) before the reduction, there must be a dependency \( a \rightarrow c \). The reduction removes such dependencies as \( a \rightarrow c \) to keep the result dependencies minimal.

### 3.3.3 Constructing Automata

This step constructs a task automaton that encodes the mined temporal dependencies. A task automaton is a septuple \((Q, \Sigma, \Delta, q_0, F, Q_f, Q_j)\) where:

1. \( Q \) is a set of states of a task execution;
2. \( \Sigma \) is a set of log message templates;
3. \( \Delta \) is a transition function \( Q \times \Sigma \rightarrow \mathcal{P}(Q) \) based on the temporal dependencies;
4. \( q_0 \) is an initial state;
5. \( F \) is a set of final states;
6. \( Q_f \subset Q \) contains fork states, which lead transitions to multiple other states by messages not dependent on each other but having weak temporal dependencies with the previous message;
7. \( Q_j \subset Q \) contains join states, to which multiple states converge on a message having weak temporal dependencies with multiple previous messages.

The construction of a task automaton is as follows. All log templates in the temporal dependencies constitute the input set \( \Sigma \). Each input in \( \Sigma \) is assigned a state, indicating the state after taking the input. Then, the assigned states with the mined dependency pairs constitute transitions in \( \Delta \). To determine the first transition from the initial state \( q_0 \), this step looks up an input that does not depend on any others. To determine the final states \( F \), the fork states \( Q_f \), or the join states \( Q_j \), this step looks for the states without any outbound transitions, have multiple outbound transitions, or have multiple inbound transitions in \( \Delta \). For each fork state, this step adds a self-loop to the transition function to allow state forking, as shown in the rows 7 and 8 of Table 3.1. In addition, we limit a fork state to take self-transitions at most the number of its outbound transitions, so that the automaton only processes a finite length of log sequence.

### 3.4 Checking Interleaved Log Sequences

In the online checking stage, CLOUDSEER uses task automata from the previous offline modeling stage to check interleaved log sequences. The input and output of this checking
Algorithm 2: Checking individual sequences.

Input: a log message \( m \); an automaton group \( G \), empty by default
Output: the updated automaton group \( G \) after its automaton instances consume the message \( m \)

Global: a set of task automata \( M \) for different tasks

1. \( G' \leftarrow \) CopyInstances\((G)\);
2. if \( G' \) is empty then
   3. \( G' \leftarrow \) InitializeInstances\((M)\);
4. \( G \leftarrow \{\} \);
5. foreach \( a \) in \( G' \) do
   6. if TryInputMessage\((a, m, t)\) is true then
      7. \( G \leftarrow G \cup \{a\} \);
8. return \( G \);

stage are as follows. CLOUDSeer takes one log message at a time from a log stream. The input message belongs to one of some interleaved log sequences being generated. Based on the task automata, CLOUDSeer outputs (1) an accepting automaton instance, if the last message completes a correct log sequence; (2) an erroneous automaton instance, if the last message meets problem-detection criteria; or (3) no output, indicating more log messages are expected.

We start from the basic case for individual log sequences assuming tasks are executed one by one. Then we relax this assumption and present the full checking algorithm. Finally, we discuss our problem detection criteria.

3.4.1 Basic Case: Individual Sequences

CLOUDSeer needs to choose a correct automaton from multiple task automata to check incoming log messages belonging to the current log sequence. Since CLOUDSeer takes one log message at a time, it does not know the right choice of automaton for sure until an automaton accepts the last message, completing the current log sequence.

Algorithm 2 shows the checking algorithm for individual sequences. Its main idea is using an automaton group \( G \) to keep track of all possible automata for the sequence being checked. \( G \) initially contains instances created from a global set of task automata \( M \) (Lines 2 to 3). Each automaton instance \( a \) in \( G \) represents one possible task, and it maintains its own state transitions based on input messages.

For each log message \( m \) with the current automaton group \( G \), the algorithm finds the
Algorithm 3: Checking interleaved sequences.

**Input:** a log message \( m \)

**Output:** an accepting or erroneous automaton instance, if the message \( m \) completes a correct log sequence, or meets an error criterion; no output otherwise

**Global:** a set of identifier sets \( J \); a set of automaton groups \( G \); a relation set \( R : J \times G \)

1. \( S_{id} \leftarrow \text{GetIdentifierValues}(m.S_v) \);
2. \( J_{\text{max}} \leftarrow \text{ComputeMaxIdentifierSets}(S_{id}, J) \);
3. \( G_{\text{max}} \leftarrow \text{GetAutomatonGroups}(R, J_{\text{max}}) \);
4. \( G_{\text{after}} \leftarrow \{\} \);
5. foreach \( G \) in \( G_{\text{max}} \) do
   6. \( G' \leftarrow \text{Apply Algorithm 2 with } m \text{ and } G \);
   7. if \( G' \) is not empty then
      8. \( G_{\text{after}} \leftarrow G_{\text{after}} \cup \{(G, G')\} \);
   9. if \( |G_{\text{after}}| = 1 \) then
      10. \( G, G' \leftarrow \text{Single}(G_{\text{after}}) \);
      11. \( ID \leftarrow \text{GetAssociatedIdentifierSet}(R, G) \);
      12. \( R \leftarrow R \setminus \{(ID, G)\} \);
      13. \( ID' \leftarrow \text{ExpandOrCreateIdentifierSet}(ID, m.S_v) \);
      14. \( R \leftarrow R \cup \{(ID', G')\} \);
   15. else if \( |G_{\text{after}}| > 1 \) then
      16. \( NID \leftarrow \text{CreateIdentifierSet}(m.S_v) \);
      17. \( G' \leftarrow \{\} \);
      18. foreach \( (G, G') \) in \( G_{\text{after}} \) do
         19. \( ID \leftarrow \text{GetAssociatedIdentifierSet}(R, G) \);
         20. \( NID \leftarrow \text{ExpandOrCreateIdentifierSet}(NID, ID) \);
         21. \( G' \leftarrow G' \cup \{G'\} \);
         22. \( R \leftarrow R \cup \{(NID) \times G'\} \);
   23. else
      24. **Divergence Recovery and Problem Detection**
      25. \( a, J, G, R \leftarrow \text{PruneIfAcceptingOrErroneous}(J, G, R, G_{\text{after}}) \);
      26. return \( a \) or \( \text{None} \);

automaton instances that make state transitions with \( m \) (Lines 5 to 7, determined by the function \( \text{TryInputMessage} \) that lets an instance decide whether it can consume \( m \)), and it passes them to the next invocation by returning the updated \( G \). A final state of any instance in the result \( G \) indicates the completion of checking a log sequence. On the other hand, an empty \( G \) indicates that none of the automaton instances can consume \( m \).
3.4.2 Checking Interleaved Sequences

We extend the checking algorithm shown in Algorithm 2 in two ways. First, the new algorithm keeps multiple automaton groups simultaneously to track interleaved sequences, where each automaton group tracks the log messages from one task execution. Second, to avoid exhaustively trying every incoming message on all automaton groups, the algorithm associates an identifier set with each automaton group, which helps determine the proper automaton group to use on-the-fly.

An identifier set is a signature for an automaton group, and it is generated based on the log sequence that has been checked by the automaton group. During the growth of a sequence, its corresponding identifier set incrementally stores identifiers appearing in log messages of the sequence. Identifier sets serve as the key to determine whether an incoming message belongs to a sequence. This is based on an observation that a log message is likely to belong to a sequence with whose identifier set the message shares the greatest number of common identifiers. Since each automaton group is associated with an identifier set, the checking algorithm can find the most likely automaton group(s) to consume an input message without trying all the groups.

Algorithm 3 shows the detail of checking interleaved log sequences. The algorithm first gets the automaton group(s) associated with the identifier set(s) having the greatest number of common identifiers with the incoming message \( m \) (Lines 1 to 3). Then the algorithm uses Algorithm 2 with the selected automaton groups to check \( m \) (Lines 4 to 8). The checking may result in three cases: (1) only one automaton group left (the branch from Lines 9 to 14); (2) multiple automaton groups taking \( m \) (the branch from Lines 15 to 22); or (3) none of the selected automaton groups to take \( m \) (the branch from Lines 23 to 24). After handling the three cases, the algorithm looks for and returns any automaton instance that is in the accepting state, or identifies a potential problem. The algorithm also cleans up related identifier sets, automaton groups, and their relations (Lines 25 to 26).

We describe the handling of the three cases below:

**Case (1): Decisive Checking.** When there is only one decisive automaton group, which shares the greatest number of common identifiers with the incoming message and can consume the message, the algorithm applies the steps from Lines 10 to 14 to update the associated identifier set and the relation between the set and the group. In particular, Line 13 expands the set \( ID \) by including new identifiers in the message \( m \), i.e., \( ID \cup m.S_v \). If the set is associated with multiple automaton groups besides the group taking the message \( m \),
the algorithm creates a new identifier set from the original one, and it then expands the
new set with new identifiers in \( m \), i.e., two sets \( ID \) and \( ID \cup m.\Sigma \). Upon the acceptance of
an automaton instance, Line 12 removes the old relation between the original automaton
group and its associated identifier set, and Line 14 adds the new relation.

**Case (2): Brute Force and Heuristics.** There can be multiple automaton groups from
Lines 4 to 8. These automaton groups all share the same greatest number of identifiers
with the incoming message and can consume the message, but they may have different
identifier sets. The algorithm cannot decisively determine an automaton group among
all the chosen ones, so it has to keep all of them before and after taking a message to
explore different possibilities. For example, suppose there are two automaton groups \( G_1 \)
and \( G_2 \) with identifier sets \( ID_1 \) and \( ID_2 \), both of which share the most identifiers with an
incoming message \( m \). \( G_1 \) and \( G_2 \) make their own state transitions to \( G_1' \) and \( G_2' \) by taking the
message \( m \). The algorithm keeps \((ID_1, G_1)\) and \((ID_2, G_2)\), and it adds \((ID_1 \cup ID_2 \cup m.\Sigma, G_1')\)
and \((ID_1 \cup ID_2 \cup m.\Sigma, G_2')\) to the relation set \( R \). When the message after \( m \) comes in, the
extended identifier set \((ID_1 \cup ID_2 \cup m.\Sigma)\) gives \( G_1' \) and \( G_2' \) an equal chance to check the
message. As a result, the algorithm tracks both possibilities: \( G_1' \) and \( G_2' \), or \( G_1 \) and \( G_2' \). Upon
the acceptance of an automaton instance, the algorithm removes all automaton groups,
identifier sets, and relations that are related to the other possibilities.

To reduce the number of possible automaton groups to track, we introduce two heuris-
tics to deal with the two causes of having multiple automaton groups. First, if two identifier
sets share the same number of identifiers with an incoming message, the function \( ComputeMaxIdentifierSets \) also computes the numbers of identifiers that are different between
these sets and the message, and it selects the set with the least difference. Second, if there
are multiple automaton groups associated with the same identifier set, the function \( GetAu-
tomatonGroups \) randomly selects one group when it finds multiple equivalent groups. Two
automaton groups are equivalent if they contain automaton instances of the same kind,
and the instances of the same kind are in the same state.

**Case (3): Divergence Recovery.** Divergences during the checking on the message \( m \)
result in an empty set returned from Algorithm 2. Such divergences may not indicate
execution problems under the following four causes. We provide a recovering heuristic for
each of these causes, and these causes are prioritized in the ascending order based on our
knowledge of their recovering cost and side effect: (a) \( m \) is not in the \( \Sigma \) of any automaton;
(b) \( m \) starts a new log sequence, so there is no corresponding automaton group; (c) the
chosen identifier set is likely not correct, so the automaton group is unable to take \( m \); and (d) \( m \) does not arrive in the order defined by the modeled temporal dependencies.

CloudSeer applies the recovering heuristics one by one until it recovers. For the cause (a), the algorithm allows unrecognizable messages to pass through. Such messages are usually not of interest to the checking process. If an error message appears, the error-message criterion described below can capture it. For the cause (b), Algorithm 2 is applied with an empty automaton group to create new automaton instances for tracking a new sequence.

If the first two heuristics cannot resolve the divergence, it is likely that the algorithm has chosen a wrong automaton group (the cause (c)). Such situation can sometimes happen if two interleaved log sequences share some identifiers, e.g., the same user boots two VMs at the same time. The algorithm tries another automaton group that has less common identifiers than the best match to take the message.

The unexpected message reordering may fail all other heuristics (the cause (d)). In this situation, a message \( B \) arrives earlier than another message \( A \), which should not happen according to the dependency \( A \rightarrow B \) from the modeling stage. There are two possible reasons for such reordering: \( A \rightarrow B \) is a false dependency or a message-delivery delay causes \( A \) to arrive late. We consider that it is unnecessary to determine the particular reason in CloudSeer’s usage scenario, so we treat all such reordering as the result of false dependencies encoded in task automata and remove such dependencies from the automaton instances.

Figure 3.4 shows an example of the removal step. The graph on the left side shows an automaton with a false temporal dependency \( B \rightarrow C \). To remove \( B \rightarrow C \) from the automaton and make it accept a new sequence \( ACBD \), the algorithm first removes the state transition from \( q2 \) to \( q3 \), and it adds two new transitions \( q1 \rightarrow q3 \) and \( q2 \rightarrow q4 \) (shown by dashed arrows in the right-side graph of Figure 3.4). The state \( q1 \) thus becomes a fork state, and \( q4 \) becomes a join state. These changes reflect two weakened dependencies \( A \rightarrow C \) and \( B \rightarrow D \) as the results of removing \( B \rightarrow C \).

### 3.4.3 Problem Detection

We implement two simple criteria in CloudSeer to detect common execution problems. The two criteria are based on two common manifestations of execution problems: (1) the presence of error messages, indicating task failures; (2) the absence or delay of messages,
indicating task failures or performance degradation.

**Error-Message Criterion** is in effect when an error message (determined by its logging level) arrives, and the checking algorithm in Algorithm 3 results in a state of divergence, as shown in Line 24. This criterion then applies the functions *ComputeMaxIdentifierSets* and *GetAutomatonGroups* to identify and output the most likely automaton group that may be associated with the error message.

**Timeout Criterion** marks any automaton groups that do not take any messages within a specified time period. The marked automaton groups become the outputs of Algorithm 3. The timeout value may vary in different systems and configurations. Determining such values is not in the scope of this work, and we leave it for future work.

### 3.5 Evaluation

We present a series of systematic experiments conducted on OpenStack to evaluate the following aspects of CLOUDSEER.

**Checking Accuracy.** The accuracy that CLOUDSEER correctly checks interleaved log sequences is fundamental to CLOUDSEER’s problem detection. We measure the accuracy based on logs collected from correct task executions. CLOUDSEER is supposed to recognize and accept all log sequences in the correct logs. Therefore, any nonacceptance would indicate inaccuracies.
Table 3.2 VM tasks for experiments.

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>Messages</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot</td>
<td>Create a new VM.</td>
<td>23</td>
<td>34</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete a VM.</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Start</td>
<td>Start a stopped VM.</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Stop</td>
<td>Stop a running VM.</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Pause</td>
<td>Pause a running VM in memory.</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Unpause</td>
<td>Bring back a paused VM.</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Suspend</td>
<td>Suspend a running VM to disk.</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Resume</td>
<td>Bring back a suspended VM.</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Efficiency. To show CLOUDSEER's capability of working in the monitoring scenario, we measure the throughput of CLOUDSEER consuming log messages and present how the identifier-based heuristic affects the checking efficiency.

Capability of Problem Detection. With typical execution problems that are injected to certain weak points in the OpenStack system, we show how well CLOUDSEER detects them and provides automaton instances as useful information related to the detected problems.

3.5.1 Test Bed

We deploy an OpenStack instance in a five-node cluster as the test bed for our experiments. The OpenStack version is Havana. In our deployment, there are one controller node, one network node running network-related services, and three compute nodes. We follow the installation guide of OpenStack [12] to deploy this instance.

We set the logging levels for all nova service components to INFO. Therefore, nova-api, nova-scheduler, and nova-compute generate INFO-level messages for VM-related tasks. Such a setting is typical for deployment, and our evaluation suggests that INFO level is sufficient to provide workflow information without imposing excessive burdens on servers.

We deploy our prototype of CLOUDSEER alongside Elasticsearch [41], which is a central log database, on the controller node. To centralize logs from OpenStack services, we deploy Logstash [79] on each server node. Logstash parses log messages generated by OpenStack services on each server node and sends the parsed messages to both CLOUDSEER and Elasticsearch on the controller node. While Logstash instances on multiple server nodes create a log stream, Elasticsearch stores all log messages in the stream. We use the log stream and stored log messages in different experiments.
3.5.2 Workload Generator

To evaluate CLOUDSER with interleaved log sequences, we implement a workload generator that simulates multiple users submitting OpenStack tasks concurrently through the command-line interface. This generator can create various workloads varying in the number of concurrent users and the number of tasks submitted by each user.

The generator uses eight primitive and typical VM-related tasks to create workloads, and they are listed in Table 3.2. To obtain the automaton for each task, we keep running the task and applying the modeling algorithm on the output logs, until logs from any subsequent task executions do not change the result automaton. In this way, the result automaton is likely to be concise regarding key log messages and complete regarding relations between messages. The number of runs for modeling each task ranges from 200 to 800, depending on the extent of nondeterminism within task executions. In Table 3.2, the column “Messages” shows the number of messages in the log sequence of each execution after preprocessing, and the column “Transitions” shows the number of transitions in the corresponding automaton.

For each simulated user, the generator randomly populates tasks by the following regular expression:

$$(\text{Boot} \ (\text{StopStart} \mid \text{PauseUnpause} \mid \text{SuspendResume})^* \text{Delete})^+$$

This expression produces multiple task groups, starting with a “boot” task and ending with a “delete” task. Within each task group, there can be multiple pairs of “stop/start,” “pause/unpause,” and “suspend/resume” tasks on the same VM. When booting a VM, we use CirrOS [34] as the guest system. To ensure the completion of each submitted task, we set each user to wait 15 seconds between two tasks.

3.5.3 Experiment Design

We design six groups of experiments (shown in Table 3.3) to evaluate checking accuracy and efficiency of CLOUDSER. These experiments test two potential factors that may affect the results of accuracy and efficiency: the number of tasks being executed concurrently, and the diversity of identifiers appearing in logs. We control the first factor by setting various concurrent users in the workload generator (shown in the column “Users”). To control the second factor, we set the generator to simulate users with identical or different user identifier (shown in the column “Single UID?”) when submitting tasks. We repeat the experiment
Table 3.3 Experiments for accuracy and efficiency.

<table>
<thead>
<tr>
<th>Group</th>
<th>Data Sets</th>
<th>Users</th>
<th>Single UID?</th>
<th>Total Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1−10</td>
<td>2</td>
<td>N</td>
<td>1600</td>
</tr>
<tr>
<td>2</td>
<td>11−20</td>
<td>3</td>
<td>N</td>
<td>2400</td>
</tr>
<tr>
<td>3</td>
<td>21−30</td>
<td>4</td>
<td>N</td>
<td>3200</td>
</tr>
<tr>
<td>4</td>
<td>31−40</td>
<td>2</td>
<td>Y</td>
<td>1600</td>
</tr>
<tr>
<td>5</td>
<td>41−50</td>
<td>3</td>
<td>Y</td>
<td>2400</td>
</tr>
<tr>
<td>6</td>
<td>51−60</td>
<td>4</td>
<td>Y</td>
<td>3200</td>
</tr>
</tbody>
</table>

Table 3.4 Execution points for failure injection.

<table>
<thead>
<tr>
<th>Injection Point</th>
<th>Type</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMQP-Sender</td>
<td>Net</td>
<td>Request RPCs to services that control VMs, e.g., nova-scheduler and nova-compute.</td>
</tr>
<tr>
<td>AMQP-Receiver</td>
<td>Net</td>
<td>Receive and process RPC requests.</td>
</tr>
<tr>
<td>Image-Create</td>
<td>I/O</td>
<td>On each nova-compute server node, create a new VM image from an image template.</td>
</tr>
<tr>
<td>Image-Delete</td>
<td>I/O</td>
<td>When destroying a VM, delete all related files.</td>
</tr>
<tr>
<td>WSGI-Client</td>
<td>Net</td>
<td>Send HTTP-based requests to query image information stored by glance services.</td>
</tr>
<tr>
<td>WSGI-Server</td>
<td>Net</td>
<td>Process queries of image information.</td>
</tr>
</tbody>
</table>

of each combination of the two factors for 10 times to produce different combinations of interleaved log sequences (shown in the column “Data Sets”). Each simulated user in an experiment is set to submit 80 tasks, resulting in the total numbers shown in the column “Total Tasks.”

The evaluation of checking accuracy and efficiency requires measuring multiple aspects of CLOUDSEER’s checking algorithm in a repeatable way. Therefore, we use the same set of stored log messages to feed CLOUDSEER multiple times to ensure consistent measurements.

To show the capability of problem detection, we randomly inject execution problems to the test bed to mimic the real-world scenario, and let CLOUDSEER monitor the test bed for the injected problems. We configure the workload generator to use four users to submit tasks concurrently. In addition, we configure the timeout value of task automata to be 10 seconds based on the performance of our test bed.

We choose six execution points to inject execution problems and apply CLOUDSEER to detect them. Table 3.4 shows these injection points and their major responsibilities for tasks in the scope of our experiments. These injection points are potentially error-prone because of network and I/O, and they reflect general and typical failure patterns that have been well
Table 3.5 Experiment results for accuracy.

<table>
<thead>
<tr>
<th>Group</th>
<th>Accuracy Range</th>
<th>Median</th>
<th>% Interleaved (≥ 2, 3, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.24%−100.0%</td>
<td>96.83%</td>
<td>48.50%</td>
</tr>
<tr>
<td>2</td>
<td>96.82%−100.0%</td>
<td>98.09%</td>
<td>65.00%, 31.17%</td>
</tr>
<tr>
<td>3</td>
<td>95.78%−98.72%</td>
<td>97.22%</td>
<td>74.34%, 48.63%, 22.84%</td>
</tr>
<tr>
<td>4</td>
<td>96.15%−97.47%</td>
<td>97.47%</td>
<td>49.00%</td>
</tr>
<tr>
<td>5</td>
<td>94.16%−99.37%</td>
<td>98.07%</td>
<td>64.67%, 32.71%</td>
</tr>
<tr>
<td>6</td>
<td>92.08%−97.87%</td>
<td>96.51%</td>
<td>80.12%, 55.59%, 30.06%</td>
</tr>
</tbody>
</table>

studied [38, 59, 62]. During the experiment, we enable these injection points one at a time and set every enabled injection point with a 25% chance to trigger an execution problem. The problem is chosen randomly by the injection point from: (a) delaying execution by a significant amount of time to simulate a performance problem, (b) aborting the current thread to simulate an unexpected exception, and (c) ignoring a network-based request or returning an incorrect file-system status, to simulate network or I/O specific problems. We use the workload generator to keep running tasks until each injection points triggers 10 execution problems. Then we examine whether the problems reported by CLOUDSeer match with the injected ones.

3.5.4 Checking Accuracy

Since there are nearly 15,000 log sequences from the experiment, it is impractical to manually determine whether CLOUDSeer accurately checks each of them with the correct automaton instance. Instead, we approximate the checking accuracy using the following formula:

\[
\text{Accuracy} = 1 - \frac{\text{# of Not-Accepted Sequences}}{\text{# of Interleaved Sequences}}
\]

We only use the number of interleaved log sequences from parallel task executions in the denominator. False temporal dependencies are the only cause of checking inaccuracies in sequential executions, because other checking heuristics are not in effect. As our task modeling is sufficient for sequential executions, we exclude them from the accuracy calculation for a close approximation. Since we cannot precisely control whether and how OpenStack executes parallel task requests, we estimate the number of interleaved sequences caused by parallel executions using the number of automaton instances simultaneously tracked.
by CloudSeer.

We determine the number of not-accepted sequences by counting automaton instances that are not in accepting states. We also compare the number of instances of each task with the number of tasks populated by the workload generator. If they do not match, we manually check related automaton instances to identify accepted instances that may happen to accept messages from multiple sequences, and we count them as not-accepted ones. However, we cannot identify the case where an accepted instance may happen to take messages from multiple sequences of the same kind of task.

Table 3.5 presents the experiment results. The column “Accuracy Range” gives the accuracy ranges showing the worst and the best results in each experiment group, and the column “Median” gives the median values. The column “% Interleaved” shows the average percentages of interleaved sequences in each experiment group. For the groups 1 and 4, the percentages are of interleaved sequences from at least two executions being in parallel (shown by the first percentage in each row). For the groups having more than two users, we also show the percentages of interleaved sequences from at least three and four executions being in parallel (shown by the second and the third percentages in each row).

The results suggest that CloudSeer is accurate enough to check most interleaved log sequences. Among six groups of experiments, the worst accuracy is 92.08% in one experiment of the group 6, while two experiments in the groups 1 and 2 result in a perfect accuracy. We also find no substantial links between the accuracy and the number of paralleled task executions or the diversity of identifiers. On the other hand, there are about 271 sequences that are not accurately checked by CloudSeer. They end up not being accepted or being accepted by wrong automaton instances. Such inaccuracies are caused by message-interleaving patterns that invalidate the heuristics on identifier sets and divergence recovery.

### 3.5.5 Efficiency

Table 3.6 presents the results of efficiency by measuring the checking time for each data set listed in Table 3.3. The column “Average Messages” shows the average number of messages produced by each experiment group. The columns “Average Time” and “Average 1k” present the average checking time and the average time per 1000 messages in each experiment. The column “% Decisive” presents the percentage of decisive checking led by the identifier-based heuristic, i.e., case (1) in Algorithm 3, among all cases. These results reflect the throughput of the checking algorithm of CloudSeer (Algorithm 3), but not the overhead
Table 3.6 Experiment results for efficiency.

<table>
<thead>
<tr>
<th>Group</th>
<th>Average Messages</th>
<th>Average Time</th>
<th>Average 1k</th>
<th>% Decisive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2164</td>
<td>3.91s</td>
<td>1.81s</td>
<td>83.13%</td>
</tr>
<tr>
<td>2</td>
<td>2990</td>
<td>6.25s</td>
<td>2.09s</td>
<td>80.76%</td>
</tr>
<tr>
<td>3</td>
<td>3849</td>
<td>8.98s</td>
<td>2.33s</td>
<td>78.18%</td>
</tr>
<tr>
<td>4</td>
<td>2158</td>
<td>4.32s</td>
<td>2.00s</td>
<td>80.12%</td>
</tr>
<tr>
<td>5</td>
<td>2989</td>
<td>7.37s</td>
<td>2.47s</td>
<td>75.48%</td>
</tr>
<tr>
<td>6</td>
<td>3820</td>
<td>11.57s</td>
<td>3.03s</td>
<td>71.43%</td>
</tr>
</tbody>
</table>

Table 3.7 Experiment results for problem detection.

<table>
<thead>
<tr>
<th>Injection Point</th>
<th>Tasks</th>
<th>D</th>
<th>A</th>
<th>S</th>
<th>Detected</th>
<th>F/P</th>
<th>F/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMQP-Sender</td>
<td>45</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>AMQP-Receiver</td>
<td>85</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td></td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Image-Create</td>
<td>256</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>*10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Image-Delete</td>
<td>240</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td></td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>WSGI-Client</td>
<td>208</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>WSGI-Server</td>
<td>87</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td></td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

*. CLOUDSEER caught an unplanned and unexpected performance problem.
**. One of the injected problems does not reflect on logs.

Overall, the results suggest a satisfactory checking efficiency regarding the throughput and reveal two key observations that are related to the achieved efficiency. First, the identifier-based heuristic is effective in improving the overall checking efficiency. The higher the percentage of decisive checking is, the less the checking time is needed. In particular, comparing groups with similar numbers of log messages, e.g., groups 1 and 4, we conclude that the throughput is proportional to the percentage of decisive checking. Second, both the number of parallel executions and the diversity of identifiers can affect the efficiency. The diversity of identifiers directly affects the chance of applying the expensive brute-force and divergence-recovery algorithms, as the throughputs of groups 4, 5, 6 are lower than those of groups 1, 2, 3. Subtly, the number of parallel executions affects the number of candidate identifier sets, which further increases the chance of not having decisive checking in the first place.

of Logstash delivering logs from each server node to the centralized location.
3.5.6 Capability of Problem Detection

Table 3.7 presents the detection results. The column “Tasks” shows the number of tasks running for each injection point, as we keep running OpenStack tasks until each point triggers 10 problems. The execution problems come randomly from three types: column “D” for delaying execution, column “A” for aborting execution, and column “S” for network or I/O specific problems. The column “Detected” shows the number of true problems detected by CLOUDSEER. The columns “F/P” and “F/N” are false positives and false negatives.

Among all 60 injected problems, 17 of them have related error messages. The other 43 problems do not lead to any error messages. Within these 43 problems, 25 of them are execution failures, and the other 18 are performance problems that are not supposed to generate any error messages.

CLOUDSEER detects 16 problems by the error-message criterion and 38 problems by the timeout criterion. These results show a precision of 83.08% and a recall of 90.00%. To verify true positives and investigate causes of false positives and false negatives, we manually check relevant automaton instances produced by CLOUDSEER and the task-submission logs by the workload generator. We also check the injection history that records injection time and VM information.

We identify two major factors affecting the precision and recall of CLOUDSEER reporting execution failures. The checking accuracy is by nature a major reason, which causes both false positives and false negatives. In Table 3.7, eight false positives and two false negatives are the results of inaccurate checking. We find that the major cause of such inaccuracies is message ordering missed by task automata due to insufficient modeling.

Figure 3.5 presents a simplified but representative case of message reordering. $A_1$ and $A_2$ represent automaton excerpts for the tasks “Stop” and “Start.” Both automata can...
consume the messages $m_1$ and $m_2$, but in different orders. The concrete messages for
$m_1$ and $m_2$ are “Lifecycle event” from a callback registered with the VM hypervisor and “Instance destroyed” from the nova-compute service. For a normal log sequence
$m_0, m_1, m_2, m_3$ (bottom left of Figure 3.5), CLOUDSeer would choose $A1$ as the automaton for the sequence after checking messages followed by $m_3$. However, $m_1 \rightarrow m_2$ turns out to be a false dependency, where the two messages could be sometimes reordered when the server workload increases (bottom right of Figure 3.5). Coincidentally, it happens that $A2$ is able to recognize the reordered messages. Therefore, CLOUDSeer would keep $A2$ instead of $A1$ for the sequence without triggering divergence-recovery strategies. Consequently, $A2$ could trigger a timeout because the expected message $m_4$ never comes, leading to a false positive. Meanwhile, CLOUDSeer may find another automaton instance to consume the unhandled message $m_3$, leading to a potential false negative if the instance should have triggered a timeout. A straightforward mitigation of such reordering is to involve manual efforts in refining the task automata once false dependencies are identified during the checking process.

In addition, multiple error messages for a single execution problem lead to three false positives and three false negatives. Our error-message criterion assumes that each execution problem would be associated with only one error message. However, we find that occasionally different services may generate error messages at the same time for the same injected problem. In such cases, after associating an error message with an automaton instance, CLOUDSeer stops choosing that instance to check any other messages. Consequently, if identifier values in the extra error messages match with existing identifier sets, CLOUDSeer may find other automaton instances to associate with, which leads to false positives. Any falsely associated automaton instances would lose the capability of detecting problems of their own log sequences, so potentially lead to false negatives.

3.6 Related Work

In essence, we contribute an approach that monitors interleaved logs and reports potential execution problems in an automated lightweight manner. The approach uses the task automaton as the workflow abstraction to provide context information out of unstructured logs for further problem diagnosis. Based on our main contribution, we differentiate our approach from existing work as follows.
3.6.1 Mining Workflow Models from Logs

Recent work [23, 24, 77, 81, 125] uses logs or traces to create workflow models for software testing, debugging, and the understanding of system behaviors. Notably, CSight [23] creates finite state machines by mining temporal invariants in logs of concurrent systems. Lou et al. [81] propose an intuitive automaton model and corresponding mining algorithms for reconstructing concurrent workflows from event traces.

Compared to such work, CLOUDSEER specifically addresses the log-monitoring problem using workflow models. To make workflow models suitable for both monitoring and system-behavior understanding that helps problem diagnosis, we adapt the idea of mining workflow models by tailoring such models to be lightweight specifications (i.e., task automata). With the tailored models, CLOUDSEER can effectively and efficiently check interleaved log sequences for the purpose of monitoring; meanwhile, it can also output monitoring results represented in workflow models that are understandable by administrators. Furthermore, we add on-the-fly refinement to task automata in order to mitigate false dependencies from mining and message-delivery delays in the production environment.

3.6.2 Log Analysis for Distributed Systems

Some log-analysis approaches apply data-mining and machine-learning techniques on logs of distributed systems [33, 43, 65, 82, 87, 131]. Some of these approaches also focus on workflow-based analysis. Fu et al. [43] proposes an approach that learns workflow models from unstructured logs and uses the models to detect performance anomalies in new input logs. The Mystery Machine [33] relies on a tremendous amount of log data to infer dependency models for critical-path analysis.

Compared to these approaches, CLOUDSEER addresses the following two key points. First, in the monitoring scenario that CLOUDSEER targets, the log-message set for analysis keeps growing and changing. Such dynamics make these techniques tend to be unstable and inefficient to produce results, because they rely on existing offline logs in which there are sufficient data points representing both normal and abnormal executions. On the other hand, CLOUDSEER separates the analysis into offline and online phases. The offline phase produces task automata as stable models for the ground truth of correct-execution logs. Then the online phase leverages the models to identify partial log sequences and look for potential problems along the growing of logs.
Second, some of the workflow-based approaches require a unique and global identifier for each request. As we discussed in Section 3.2, although the use of identifiers in logs is ubiquitous, there is usually no single and unique identifier that is propagated across all distributed components. In CLOUDSEER, we lift the requirement of such unique and global identifiers by introducing the identifier-based heuristics. The heuristics work on-the-fly to associate multiple identifiers that should belong to the same log sequence, but are used by different distributed components.

In addition, lprof [144] is a request-flow profiler for distributed systems. It reconstructs request flows from logs and assists the diagnosis of performance anomalies. lprof relies on static analysis on Java bytecode to identify logging information, such as request identifiers for attributing log messages to flows. To the contrary, CLOUDSEER is language-agnostic and complements lprof on systems where static analysis is tedious or difficult to implement.

3.6.3 Log-based Failure Diagnosis

Insight [88] and SherLog [139] use error logs and other artifacts, such as source code, to infer or reproduce failure paths. These failure-diagnosis techniques work on a single-thread or single-request basis. CLOUDSEER may be beneficial to these techniques when being applied to complex systems, because CLOUDSEER can provide abstracted log sequences including erroneous ones from parallel task executions.

3.6.4 Distributed Tracing Frameworks

There are frameworks targeting at tracing large-scale distributed systems [10, 20, 105, 145]. These frameworks require instrumentation in specific system components, e.g., network library, to generate traces for further analysis. Compared to these frameworks, CLOUDSEER purely works on logs that are readily available in existing running systems, so it does not require any special instrumentation. Since logging is a common practice in developing large-scale systems, CLOUDSEER is highly usable on existing production systems with general logs.
4.1 Introduction

Server-side applications commonly expose services to users in a request-based paradigm, in which user requests are served by application-defined request-handler methods. We refer to such applications as request-based applications. Request-based applications are playing an increasingly important role in the current software ecosystem, providing classic web pages and emerging cloud-based application services. The increasing prevalence of these applications brings new demands and challenges regarding software quality, thereby calling for advanced techniques to ensure application correctness, performance, and security.

Modern request-based applications are usually built on top of some supporting frameworks, e.g., Spring [107] and Struts [11] for web applications. In the request-based execution
model, upon receiving a request, the framework invokes one or more request-handler methods, and the invoked method(s) return data objects containing the necessary information to serve the request. These data objects are further processed by the framework to generate a concrete response to be sent to the user side (e.g., in the form of HTML). While each request is served modularly, users can issue a series of related requests, which may be based on the received responses (e.g., through a hyperlink generated dynamically from some data objects). Therefore, the output of one serving request-handler method could become the input of another method, creating data dependencies between these methods. We say two request-handler methods are related if there are data dependencies between them.

With the execution model, it is important for program analysis techniques targeting request-based applications to track data dependencies across related methods and perform inter-request analysis. While there is a natural analogy between inter-request analysis and interprocedural analysis [93, 100], one cannot directly apply techniques developed for interprocedural analysis for inter-request analysis. The challenges are mainly rooted in the way how data dependencies between request-handler methods are established: (1) because of the separation of server and user sides, data dependencies are not directly established by propagating data objects across methods through return values or parameters, and (2) because of the framework, there is no explicit caller-callee relationship between methods serving different requests. These complications make inter-request data dependencies not perceivable by conventional interprocedural or summary-based analysis techniques [94, 95, 132, 133].

Existing techniques in different fields are affected by the limited capability of conducting inter-request analysis. In the performance field, a previous study [57] shows that many performance bugs are caused by skippable function calls or inefficient function-call combinations. Request-based applications are more prone to such inefficiencies, as the request abstraction is highly modular, and developers can easily write inefficient code spread across individual request-handler methods. Although various techniques [31, 32, 83, 92] have been proposed to alleviate such inefficiencies, they may not reach their full potential without inter-request capability. In the security field, investigating malicious-data propagation is crucial. Request-to-response propagation within each request is a common target for existing work [108, 122], but propagation that spans multiple requests can also create vulnerabilities. Some work [3, 84] has discussed and tried identifying such vulnerabilities by exploring navigation sequences of web pages.
To overcome challenges and build a solid foundation for inter-request analysis on modern request-based applications, we propose an approach called dataflow tunneling. Our approach is based on the key observation that framework and user behaviors are of less interest in application-code analysis. For example, knowing how a framework accesses some data objects is less important than knowing what data these objects carry. Therefore, our approach effectively abstracts away the involvement of frameworks and users in propagating data across requests, and directly exposes data dependencies across request-handler methods. We define such dependencies in the form of data-dependency specifications, which describe how the output of a method tunnels (i.e., being propagated and transformed) through frameworks and users to become the input of another method. The resulting specifications provide necessary information to allow conventional analysis techniques to go across the boundaries of request-handler methods without analyzing supporting frameworks or user behaviors.

Our approach analyzes concrete application executions to capture data dependencies. We design our approach with three key techniques: (1) a static analysis on request-handler methods to identify input and output sites where these methods and their supporting framework pass data objects to each other; (2) on data objects passing through the identified sites, an object-centric tracing technique to capture object accesses and their accessed values throughout the whole processing of every single request; and (3) a mining technique to identify potential data dependencies between request-handler methods in traces from different requests. With this design, our approach adheres to the general request-based execution model instead of specific framework implementations, thereby avoiding ad hoc analyses on framework code.

We have implemented a prototype for Java-based web applications, and evaluated the specification accuracy and tracing overhead on two real-world open-source applications developed with different supporting frameworks. We have also manually inspected the generated specifications to investigate how useful they are for program understanding and future inter-request analysis techniques. The results show good accuracies (above 80%) in typical configurations, and suitable overheads for an in-house testing environment. Our manual inspection helps us gain a better understanding of the two applications. Based on the understanding, we present a use case of the specifications in tuning object caching, which shows promising results in database-query reduction and execution-time reduction. We believe these specifications can help enable future sophisticated inter-request analysis.
Overall, we make four main contributions. First, we elaborate the importance and challenges of inter-request analysis, especially on challenges posed by modern frameworks. Second, we propose an approach to infer inter-request data dependencies. One novelty of our approach is to abstract away the involvement of frameworks without human-assisted modeling. Third, we prototype and evaluate our approach on two open-source applications regarding accuracy and overhead. Fourth, we study the data-dependency specifications generated by our prototype and present a usage scenario on how they can help program understanding and future analysis techniques.

4.2 Background and Motivation

We present an extended discussion on the background of request-based applications and the challenges of inter-request data-dependency analysis. The discussion is based on a real-world example, which involves a popular Java request-based framework.

4.2.1 Background

Figure 4.1 illustrates the general request-based application execution model, which captures the commonalities among applications developed with different frameworks. This abstract model shows a single request-to-response flow. Along the flow, three major components of
a request-based application are involved: (1) request-handler methods, (2) data objects, and (3) view templates.

*Request-handler methods* are the entry points of a request-based application to process incoming requests. For each request, the framework determines and invokes the appropriate handler method(s) through a predefined *request-handler mapping*. The request-handler mapping defines requests that each handler method can process. It is specified by developers in different forms depending on the underlying frameworks, e.g., Spring [107] uses Java annotations and Struts [11] uses configuration files. Depending on specific framework implementations, one or more request-handler methods may get invoked to process one request.

During the execution of a request-handler method, it creates *data objects* and passes them to the framework. These data objects encapsulate data that are usually retrieved from a backend database. The framework then references and uses these objects to generate concrete responses. Based on our investigation of popular frameworks for Java listed in an online source [117], the majority of them allow request-handler methods to pass data objects through either method returns or collection-based parameters, while some require request-handler methods to use framework-specific APIs. In this work, we follow the object-passing model of the majority, but our approach can be extended to include framework-specific APIs.

*View templates* define how data objects should be further processed to display to the client, and such templates are usually written in different markup languages, such as FreeMarker [9], JSTL [61], and Velocity [118]. With the data objects from request-handler methods, the framework chooses a view template, either programmatically or based on a predefined *view-template mapping*, to populate a *concrete view*. Finally, the concretized view will be sent to the user as the *response*, which can be as complex as a web page or as simple as plain text that is serialized from data objects.

### 4.2.2 Motivating Example

Figure 4.2 shows an example from OpenMRS [90], which is an open-source web-based medical-record system. It uses the Spring Framework [107] as the application framework. The listed code includes three request-handler methods implemented in Java, with excerpts of JSP [54] files that are their corresponding view templates. The request-handler method and view template for handling requests to the path /patientDashboard.form
Figure 4.2 An excerpt of three related request-handler methods and their view templates from OpenMRS. Only method parameters related to our discussion are shown.

are listed from lines 1 to 19. They produce an overview page for the patient specified in the request. The method `renderDashboard()` takes two parameters: (1) an integer `patientId` passed in by the Spring Framework as a request parameter, and (2) a `ModelMap` object `map` carrying data objects to the corresponding view template. The method uses
patientId to create a Patient data object (line 7), and then puts it into map (line 8). With the method returning the string “patientDashboardForm” (line 10), the framework chooses the view template patientDashboardForm.jsp (lines 14 to 19) to produce a concrete patient-overview page. The page contains a button (line 15) that can fire another request to show some visit information of the specified patient. This new request requires the patient identifier to be embedded as a request parameter. At runtime, the Spring Framework evaluates model.patient.patientId (line 18) to fill in the request path (lines 17 - 18) with a concrete value. The value evaluation relies on the Java reflection mechanism to find the data object bound to model.patient, and invoke the accessor method bound to the field name patientId. In this example, the object is the Patient object created by renderDashboard(), and the accessor method is Patient.getPatientId().

When a user clicks the button on the overview page, a request for the patient’s visit is fired to the request path /admin/visits/visit. This request is handled by the code from lines 21 to 45. Lines 21 to 27 comprise the request-handler method showForm(), and this method requires a Visit object populated by another method getVisit() listed from lines 28 to 38. At runtime, the Spring Framework invokes getVisit() before showForm() to create the Visit object with its corresponding Patient object from request parameters visitId and patientId. The view template visitForm.jsp accesses the Visit and Patient objects bound to visit.patient multiple times (lines 43 and 45) for different pieces of information.

**An inter-request caching opportunity.** We show that inter-request analysis is useful to identify an optimization opportunity. For the two requests in the example, their corresponding methods invoke PatientService.getPatient() (lines 7 and 35), which queries a database for patient data to create a Patient object. With inter-request data dependencies, one can determine that the flow 4→7→8→18→29→35 propagates the value of patientId from the first method invocation to the second. This propagation hints that one may add a cache mechanism to cache the Patient object from the first invocation, so the cached object is available on the second invocation without additional queries to the database. In Section 4.4.6, we show a use case using Hibernate [50], a data-access framework, to achieve such caching.

**The challenges of inter-request data-dependency analysis.** Following the discussion in Section 4.1 that framework behaviors and user interactions complicate the analysis of inter-request data dependencies, we concretely discuss the challenges based on our
example with the assumption that we were to extend conventional interprocedural analyses to compute inter-request data dependencies.

First, handling application components in different languages would be necessary but ad hoc. Request-handler methods and view templates are commonly developed in different languages, and concrete responses can also contain scripts that perform client-side operations to propagate data across requests. In our example, three languages are involved to propagate the patientId value: Java for handler methods, Expression Language [124] for evaluating concrete values on data objects in view templates, and a piece of JavaScript code (lines 16 to 18) to propagate the patientId value by firing the second request from the user’s browser. As data propagation goes through all the three components, ad hoc supports for different languages would be necessary to capture data dependencies between the involved request-handler methods.

Second, analyzing framework code would be necessary but challenging to capture data propagation outside request-handler methods. The framework is the caller of request-handler methods and view templates, and it provides the supports of parsing requests, binding and accessing data objects, and populating concrete views. As our example shows, the framework heavily relies on the reflection mechanism, which performs dynamic object instantiation and method invocations at runtime. For instance, the framework uses the annotation @ModelAttribute (line 28) to bind the Visit object in Java code to the view template with a textual name visit (line 43). It is not until the time of execution that the framework parses these annotations and textual names, and uses the reflection mechanism to invoke the parsed methods on the bound objects. It is known that this runtime mechanism brings difficulties to static analysis. Researchers have proposed different approaches to address this problem with their own pros and cons [30, 73, 76, 102, 108, 119].

### 4.3 Dataflow Tunneling

Our approach requires two artifacts: the application code of request-handler methods specified by developers for analysis and test inputs for generating traces. With the two artifacts, our approach performs three key steps: (1) static analysis on request-handler methods to identify input and output sites, (2) instrumenting the methods and tracing application executions under test inputs, and (3) mining the generated traces for data-dependency specifications. The results of our approach are data-dependency specifications
across request-handler methods. Such specifications describe how the output of a request-handler method is propagated and transformed into the input of another request-handler method. We use the following notations to represent data-dependency specifications:

1. $a_{\text{method}}^{\text{type}}$ denotes that an entity with a name $a$ is of type and used in method. We use a special notation $obj$ to denote an entity without a name, and we use $VIEW$ as a special method to denote that the entity is used outside a request-handler method.

2. $a \Rightarrow b$ denotes that $a$ is propagated to $b$ without transformation (we omit the method and type on $a$ and $b$ for simplicity). There are two propagation channels: $PARAM$ for propagation through a collection-based parameter, or $RETURN$ for propagation through a method return. We put the propagation channel over the arrow. For channel $PARAM$, we put the parameter name under the arrow.

3. $a \rightarrow b$ denotes that $a$ is transformed and then propagated to $b$. We put the method name related to the transformation over the arrow. Sometimes $a$ may represent a collection of objects (e.g., a set of Patient objects), while $b$ may represent only a single value (e.g., the patientId of one of the objects). Such cases indicate a many-to-one relation between $a$ and $b$. We put the notation $CHOICE$ under the arrow to denote these cases.

With these notations, the data-dependency specifications for the example in Figure 4.2 are:

\[
\text{obj}_{\text{renderDashboard}} \xrightarrow{PARAM_{\text{map}}} \text{obj}_{\text{VIEW}}^\text{Patient} \quad \text{(4.1)}
\]

\[
\text{Patient.getPatientId()} \xrightarrow{\text{patientId}_{\text{Integer}}} \text{getIdVisit()} \quad \text{getVisit()} \quad \text{(4.2)}
\]

Specification (4.1) describes that a Patient object from the method renderDashboard() is passed through a parameter channel map (denoted by $\Rightarrow$) to the view. Then the object is transformed by Patient.getPatientId() into an integer, which is propagated to the input parameter patientId of the method getVisit() (denoted by $\rightarrow$). Specification (4.2) describes that a Visit object is returned from the method getVisit(), and then the object is propagated to the parameter visit of the method showForm().
4.3.1 Identifying Input and Output Sites

To capture data dependencies across request-handler methods, we first need to know what input each method receives and what output each method produces. The goal of this step is identifying the input and output sites in each request-handler method. An input site indicates where a handler method receives input data from the framework, and an output site indicates where the method passes output data to the framework. The identified sites allow the later instrumentation and tracing step to collect necessary information.

We apply a static taint analysis to identify input and output sites, as essentially they are sources and sinks between which data is propagated within the scope of a request-handler method. For each request-handler method, we start with a conservative set of sources and then identify sinks based on a propagation graph computed by the static taint analysis. We further filter sources and sinks using a set of rules that incorporate the request-based execution model. We consider variables in the resulting sources as input sites and variables in the resulting sinks as output sites.

We define the propagation graph as follows, followed by the rules to determine sources and sinks.

**Definition 6** A Propagation Graph is a directed graph \( PG = (V, E) \) where \( V \) is a set of nodes for variables labeled by program locations, and \( E \) contains edges over \( V \). Each edge \( v_1^{l_1} \rightarrow v_2^{l_2} \) indicates a flow of data from variable \( v_1 \) at program location \( l_1 \) to \( v_2 \) at \( l_2 \).

In producing such propagation graphs, we consider two types of variables as sources: (1) parameters of a request-handler method and (2) variables that receive data from static method invocations. These two sets of sources reflect two common ways that a request-handler method receives request inputs from the framework: either from parameters or through certain APIs. Starting from these source variables, we apply conventional propagation rules to compute a propagation graph for each method. In particular, we propagate tainted values starting from sources through normal assignments, and arguments and return values in method invocations. To propagate through virtual method invocations, whose method bodies cannot be determined statically for direct analysis, we introduce two heuristics. First, if \( b \) is tainted in the case of \( a.m(b) \), \( a \) should also be tainted (i.e., introducing an edge \( b \rightarrow a \)). This rule reflects a common pattern that the object \( a \) receives data from the object \( b \). For example, line 35 in Figure 4.2 is a typical case in which the \texttt{Visit} object stores the \texttt{Patient} object in a field. Second, for the case \( a = b.m(c) \), \( a \) should be
tainted if \( b \) or \( c \) is tainted. This is an over-approximation that aggressively creates possible flow edges when \( m \) is not available for direct analysis.

On a propagation graph constructed based on the propagation rules, the nodes without any incoming edges are likely sources, and the nodes without any outgoing edges are likely sinks. We then apply the following three rules to filter sources and sinks for input and output sites. First, the variable in a sink is an output site if it is a parameter (e.g., a collection object), in a return statement (e.g., an object returned by the method), or an argument of a static method invocation. This rule indicates that a sink variable is an output site only if the data carried by the sink variable is visible outside a request-handler method. Second, an object variable marked as both a source variable and a sink variable is likely to be an output site, as the object referenced by the variable is likely to hold new data derived from other inputs. Later such data may be accessed outside the request-handler method. Third, source and sink variables not affected by the first two rules are input sites and output sites, respectively.

For example, we initially consider the parameter \( \text{map} \) of \( \text{renderDashboard} \) in Figure 4.2 as a source, and then we find that it acts like a sink at a later program location. With the second rule, we classify the parameter \( \text{map} \) as an output site.

### 4.3.2 Instrumentation and Tracing

In this step, we instrument the identified input and output sites, and then execute the application with test inputs to trace and collect necessary information for the later mining step. For each method execution, two sets of events are generated: input events and output events. Input events record concrete input values passed into request-handler methods by the framework, and output events record (1) the concrete values in request responses and (2) the method-invocation sequences representing how the framework retrieves these values from output data objects.

We specifically design this step not to ad hoc analyze or trace complex framework executions. Instead, this step focuses on data-object accesses (e.g., getting an object field). This idea is based on the observation that all framework-dependent code, regardless of languages and the use of reflection, finally has to invoke methods on data objects to retrieve concrete data. Therefore, tracing object accesses is a reasonable way to observe data propagation beyond request-handler methods and into framework and view templates.

In particular, we use a dynamic proxy-based technique on data objects. A proxy object is a type-safe delegate of its original object. We can replace a data object with its proxy object,
Table 4.1 Instrumentation and tracing rules. Trac Point: whether a trace event is fired on an instrumentation point (Inst) or on a proxy object (Proxy); Exec Point: the current scope of execution, in a request-handler method or the framework; Target Statement: the instrumented code for an instrumentation point, the intercepted method invocation for a proxy point (obj always references a proxy object); Tracing Action and Conditions: the tracing step performs the action under certain conditions.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Trac Point</th>
<th>Exec Point</th>
<th>Target Statement</th>
<th>Conditions</th>
<th>Tracing Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE1</td>
<td>Inst</td>
<td>Method</td>
<td>enterMethod(mName)</td>
<td></td>
<td>Fire an event of EnterMethodEvent, recording the method name mName.</td>
</tr>
<tr>
<td>RE2</td>
<td>Inst</td>
<td>Method</td>
<td>exitMethod(mName)</td>
<td></td>
<td>Fire an event of ExitMethodEvent, recording the method name mName.</td>
</tr>
<tr>
<td>RI1</td>
<td>Inst</td>
<td>Method</td>
<td>recordInput(var)</td>
<td>var is not of the type HttpServletRequest.</td>
<td>Fire an event of InputEvent, recording the value or object identifier of var.</td>
</tr>
<tr>
<td>RI2</td>
<td>Inst</td>
<td>Method</td>
<td>recordInput(var)</td>
<td>var is of the type HttpServletRequest.</td>
<td>Fire an event of InputEvent that records the object identifier of var, and create a proxy object for the object referenced by var.</td>
</tr>
<tr>
<td>RO1</td>
<td>Inst</td>
<td>Method</td>
<td>recordOutput(var)</td>
<td>var is a simple value.</td>
<td>Fire an event of OutputEvent, recording the value of var.</td>
</tr>
<tr>
<td>RO2</td>
<td>Inst</td>
<td>Method</td>
<td>recordOutput(var)</td>
<td>var references a complex object.</td>
<td>Fire an event of OutputEvent that records the object identifier of var, and create a proxy object for the object referenced by var.</td>
</tr>
<tr>
<td>PM1</td>
<td>Proxy</td>
<td>Method</td>
<td>obj.m(arg1, ...)</td>
<td></td>
<td>For each argi, fire an event of OutputEvent, and create a proxy object for argi if it is of a complex type.</td>
</tr>
<tr>
<td>PM2</td>
<td>Proxy</td>
<td>Method</td>
<td>obj.m(arg)</td>
<td>obj is of the type HttpServletRequest, and m has a return value.</td>
<td>Fire an event of InputAccessEvent to record arg and the returned value.</td>
</tr>
<tr>
<td>PF1</td>
<td>Proxy</td>
<td>Framework</td>
<td>obj.m(...)</td>
<td>m returns a complex object.</td>
<td>Create a proxy object to replace the returned object, and append m to the invocation sequence that leads to the current invocation.</td>
</tr>
<tr>
<td>PF2</td>
<td>Proxy</td>
<td>Framework</td>
<td>obj.m(...)</td>
<td>m returns a simple value.</td>
<td>Fire an event of OutputAccessEvent to record the returned value and the previously recorded invocation sequence plus m.</td>
</tr>
</tbody>
</table>
and the proxy object can intercept method invocations made on the original object without affecting application behaviors. This technique involves two key ideas: (1) dynamically creating proxy objects to intercept and then dispatch method invocations to the original objects, and (2) propagating proxy objects based on the context of method executions to capture framework behaviors in different execution points. By dynamically creating and propagating proxy objects, we can record a sequence of methods that the framework invokes on each data object, together with concrete values returned by this sequence. This technique allows us to capture framework behaviors but effectively avoid the scalability issue of ad hoc analysis on different languages and analysis limitations caused by reflection.

Table 4.1 summarizes the instrumentation and tracing rules. These rules fire tracing events and create proxy objects on demand. We describe these rules in detail as follows.

We instrument `enterMethod(mName)` and `exitMethod(mName)` at the entry and exit points of each request-handler method to record the start and end of each method execution. These points generate events `EnterMethodEvent` and `ExitMethodEvent` (`RE1` and `RE2`). We introduce these events to distinguish different method executions. They only record the name of the method being executed.

For variables identified as input or output sites, we introduce their corresponding instrumentation rules. For each variable `varIn` identified as an input site, we instrument `varIn = recordInput(varIn)` at the beginning of the method. For each variable `varOut` identified as an output site, we instrument `varOut = recordOutput(varOut)` to a point that depends on the output type. If `varOut` appears in a return statement, the instrumentation is placed before the statement. In other cases, the instrumentation is placed where the method first initializes `varOut`: either at the beginning of the method if `varOut` is an out-parameter, or right after an API invocation if `varOut` receives a return object. Such instrumentation serves two purposes: recording necessary information on input and output sites, and creating proxy objects to further trace objects passed through these sites.

The instrumented input and output sites generate events of `InputEvent (RI1, RI2)` and `OutputEvent (RO1, RO2)`. The information recorded in these events depends on the type of `varIn` or `varOut`: a simple value for a simple type, or an object identifier for a complex class. The instrumented sites also create proxy objects to replace the original objects in order to trace method invocations. In particular, the rules `RI2` and `RO2` describe two situations where proxy objects are created. First, if an output-site variable `varOut` references a complex object, the instrumentation creates a proxy object to replace the referenced object (`RO2`).
As a result, the tracing step can trace how the request-handler method and the framework use the referenced object. Second, the instrumentation creates a proxy object on the input site that references an HttpRequest object (RI2). This is for backward compatibility with request-handler methods that directly access the HttpRequest object to retrieve request inputs.

With the initial proxy objects created at instrumentation sites, we discuss rules PM1, PM2, PF1, and PF2, which specify the behaviors of proxy objects when they intercept method invocations.

Rule PM1 or PM2 applies when a proxy object intercepts a method invocation within a request-handler method. If the proxy object is an HttpRequest object, we consider that it presents an input site, therefore an event of InputAccessEvent is fired (PM2). The InputAccessEvent represents the behavior of retrieving request input from the HttpRequest object, and records the return value and argument values of the method invocation.

The cases other than invocations on an HttpRequest object imply that the target proxy object serves as an output site of the method, and the invocation arguments are likely to be visible and used by the framework. Therefore, we apply PM1 to fire an event of OutputEvent on each argument involved in the invocation, and replace each argument that is of a complex type with a proxy object. This rule captures output data that is passed out by the handler method, as well as propagates proxy objects to the framework to further trace runtime behaviors outside the method.

For invocations intercepted outside the scope of a request-handler method, we introduce rules PF1 to trace an intermediate invocation that returns another complex object, and PF2 to trace the final invocation that returns a simple value. These two rules are based on the observation that the framework may invoke a series of methods on a complex object to retrieve a simple value (e.g., visit.patient.patientId). If an intercepted method invocation returns a complex object, we consider it as an intermediate step towards the final output value, and we apply PF1 to create a new proxy object to replace the return object to further trace subsequent invocations and record the current invocation in the invocation sequence. Otherwise, if the invocation returns a simple value, we apply PF2 to fire an event of OutputAccessEvent, which records the returned value, as well as the sequence of invocations recorded along with all the intermediate invocations plus the final one.

Overall, this tracing step produces a trace for each request. Each trace is in the form
of \( (\text{EnterMethodEvent} \mid \text{InputEvent} \mid \text{InputAccessEvent} \mid \text{OutputEvent})^* \text{ExitMethodEvent} \text{OutputAccessEvent}^* \)*, indicating that zero or more request-handler methods are executed for one request. During the execution of each method, there can be zero or more \text{InputEvent}, \text{InputAccessEvent}, and \text{OutputEvent}. After each method execution, there can be zero or more \text{OutputAccessEvent}. For every access event (\text{InputAccessEvent} or \text{OutputAccessEvent}) in a trace, the same trace must have one corresponding source event (\text{InputEvent} or \text{OutputEvent}) before the access event, and the source event records information about the accessed root object.

\textbf{An example with Figure 4.2}. We use \texttt{renderDashboard()} in Figure 4.2 as an example to show how the analysis and tracing steps work. The static-analysis step first identifies the parameter \texttt{patientId} as a source variable and the parameter \texttt{map} as a sink variable. Then we instrument two lines of code \texttt{patientId = recordInput(patientId) and map = recordOutput(map)} at the beginning of the method. We also instrument events \texttt{enterMethod} and \texttt{exitMethod} at the beginning and end of the method, respectively.

In the tracing step, the execution of \texttt{renderDashboard()} yields a trace consisting of \texttt{EnterMethodEvent}, \texttt{InputEvent} on \texttt{patientId}, \texttt{OutputEvent} on \texttt{map}, \texttt{OutputEvent} on the \texttt{Patient} object, \texttt{ExitMethodEvent}, \texttt{OutputAccessEvent} representing a method invocation of \texttt{Patient.getPatientId}. The \texttt{InputEvent} is triggered by \texttt{RI1}, and the first \texttt{OutputEvent} by \texttt{RO2}. A proxy object for \texttt{map} is created to replace the original object during this process. Then this proxy object fires the second \texttt{OutputEvent} by \texttt{PM1} when it intercepts \texttt{map.put()}. It also creates and passes a proxy object for the original \texttt{Patient} object into \texttt{map}. When the framework executes \texttt{patientDashboardForm.jsp}, the proxy object for the \texttt{Patient} object intercepts an invocation \texttt{Patient.getPatientId()}, and fires \texttt{OutputAccessEvent} by \texttt{PF2} as the method returns a simple value.

\subsection*{4.3.3 Specification Mining}

Based on the trace structure defined in Section 4.3.2, each trace represents the processing of a single request. We first leverage the user-session information collected along tracing to distinguish and group traces of requests from different user sessions. We then use a k-length sliding window algorithm to infer data-dependency specifications from traces from the same user session. Our algorithm maintains a k-length sliding window because data dependencies may exist across non-adjacent requests. For example, the two requests mentioned in Section 4.2.2 are actually not adjacent, which is separated by a background
Algorithm 4: Mining traces with a k-length sliding window.

- **Input:** A trace \( t \) from a trace stream of the application
- **Global:** A configurable integer \( k \), a map \( M \) associating every session identifier with a list storing up to \( k - 1 \) previous traces, and a database \( DB \) storing existing data-dependency specifications

```
1 list ← M[GetSessionId(t)];
2 AppendLast(list, t);
3 foreach \( t' \) in list do
4    methodPairs, specs ← MineSpec(t', t);
5       foreach \( p \) in methodPairs do
6          UpdateMethodPairStats(p, DB);
7       foreach \( s \) in specs do
8          UpdateSpecDB(s, DB);
9       if Size(list) = k then
10          RemoveFirst(list);
```

request triggered by the first patient-overview page. In practice, data dependencies are unlikely to exist between methods of a long request distance, and we expect the sliding window size to be small. One can determine a proper window size in different ways, depending on the environment where the application under analysis is running. In an in-house testing setting, one may repeat test runs and gradually increase the window size until no true specification is found. In a production environment where requests and workloads are not repeatable, one may use the windows size determined during in-house testing or heuristically adjust the size online, and we leave the latter for future work.

Algorithm 4 presents an outline of the k-length sliding window algorithm. For each incoming trace \( t \), the algorithm pairs \( t \) with \( k - 1 \) previous traces from the same user session as well as \( t \) itself, and applies the subroutine \( \text{MineSpec} \) on these pairs. Mining each of these trace pairs yields specifications that describe data dependencies between request-handler methods in the paired traces. Note that mining the reflective pair \((t, t)\) yields specifications when multiple request-handler methods are involved in processing the same request. From the subroutine \( \text{MineSpec} \), the algorithm gets a set of method pairs and a set of specifications. These pairs cover request-handler methods between which the algorithm tries to identify data dependencies. The algorithm stores these pairs (via \( \text{UpdateMethodPairStats}() \)) for the statistical purpose of calculating and updating the confidence of each mine specification via \( \text{UpdateSpecDB}() \). For a specification \( s \), we define its confidence as the frequency of having the specification \( s \) identified between two request-
handler methods when these two methods appear together in different trace pairs. We use confidence as an indicator of the likelihood that a specification is true and worth further analysis.

Algorithm 5 shows the subroutine MineSpec, which does the actual work to infer data dependencies across two methods. The basic idea of this algorithm is to check whether some output values from the trace of one request and some input values from the trace of another request are equal. A pair of equal values indicates a possible inter-request data dependency. The adoption of this equivalence relation is backed by the observations that (1) unique values likely exist to approximate the actual dependencies, and (2) these values are usually carried from the output of one request to the input of another without complex computations in between. For example, the value of `patient.patientId` would be unique for every patient, and the value is not involved in any complex computation.

In particular, Algorithm 5 takes two traces $t_1$ and $t_2$ to perform the following three major steps. First, it extracts output and input related events from $t_1$ and $t_2$. The extracted events include `OutputEvent` and `OutputAccessEvent`, indicating the output of request-handler method(s) in $t_1$, and `InputEvent` and `InputAccessEvent`, indicating the input of method(s) in $t_2$. Since `OutputAccessEvent` and `InputAccessEvent` record method-invocation sequences, the same invocation sequence in different events essentially reflects either repetitive or iterative data-access actions, which may indicate a many-to-one relation between the output and input of the request-handler methods (i.e., the CHOICE case in specifications). Therefore, the algorithm clusters events that record an identical invocation sequence (via `ClusterOutputEvents()` and `ClusterInputEvents()`) to form an event set, which is used in later steps as a whole to represent identical object-access actions.

Given the clustered input and output events, the second step constructs a matching table to compute potential data dependencies using concrete values recorded in these events. As shown below, the rows of the table represent clustered output events in trace $t_1$, and the columns represent clustered input events in trace $t_2$.

<table>
<thead>
<tr>
<th>$t_1$ EventOutput1</th>
<th>$t_2$ EventInput1</th>
<th>$t_1$ EventOutput2</th>
<th>$t_2$ EventInput2</th>
<th>$t_1$ EventOutput3</th>
<th>$t_2$ EventInput3</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-match</td>
<td>one-to-one</td>
<td>no-match</td>
<td>no-match</td>
<td>many-to-one</td>
<td>no-match</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

For each cell of the table, the algorithm extracts concrete values from the events, and checks
Algorithm 5: MineSpec on a trace pair.

**Input:** Two traces $t_1$ and $t_2$

**Output:** A set $methodPairs$ storing pairs of request-handler methods, and a set $specs$ storing derived data-dependency specifications

1. $methodPairs \leftarrow []$;
2. $specs \leftarrow []$;
3. $oEventList \leftarrow$ ClusterOutputEvents($t_1$);
4. $iEventList \leftarrow$ ClusterInputEvents($t_2$);
5. $matchTable \leftarrow$ ConstructMatchTable($oEventList, iEventList$);
6. for $i \leftarrow 0; i < \text{Size}(oEventList); i++$ do
   7.     for $j \leftarrow 0; j < \text{Size}(iEventList); j++$ do
   8.         $matchType \leftarrow matchTable[i][j]$;
   9.         if $matchType$ is not no-match then
   10.            $s \leftarrow$ CreateSpec($matchType, oEventList[i], iEventList[j]$);
   11.            Add($specs, s$);
   12.            $method1 \leftarrow$ GetMethodName($oEventList[i]$);
   13.            $method2 \leftarrow$ GetMethodName($iEventList[j]$);
   14.            Add($methodPairs, (method1, method2)$);
15.     return $methodPairs, specs$;

for matched values, resulting in one of three types of outcomes: (1) no value is matched, then the algorithm marks the cell as no-match; (2) a single value is matched, and the output and input events each contains only one concrete value, then the algorithm marks the cell as one-to-one; (3) a single value is matched, and the output event contains multiple concrete values, then the algorithm marks the cell as many-to-one. We currently do not consider the case of many-to-many, as it is not commonly observed, but our approach can be extended to support such cases.

With a constructed matching table, the algorithm goes over each cell in the table and creates a specification for the cell that indicates a potential dependency (lines 6 - 14 in Algorithm 5). In particular, CreateSpec() (line 10) identifies (1) output and input entities by tracking back from OutputAccessEvent or InputAccessEvent to its corresponding OutputEvent or InputEvent, (2) the channel (⇒) that propagates an output object to the framework by determining whether OutputEvent is from a return statement or an out-parameter, and (3) the transformation actions (→) by extracting the invocation sequence from OutputAccessEvent or InputAccessEvent. With the matching type from each cell, the algorithm creates a specification describing how an output object of a method is propagated and transformed into the input of another method.
4.4 Evaluation

We have built a prototype of our approach for Java-based web applications. We use Soot [26] for static analysis and instrumentation, and Byte Buddy [129] for dynamic proxy creation. With the prototype, we evaluated the following aspects of our approach.

**Specification Accuracy.** We measure how accurate the generated specifications are in describing data dependencies between request-handler methods. We consider a specification to be true if it reveals a real data dependency between the involved methods but not due to coincidentally equal values, and we manually examine and label each specification based on our understanding of the subjects’ code and runtime behaviors.

**Tracing Overhead.** We measure the runtime overhead imposed by the tracing step. Measuring the tracing overhead helps us determine the applicability of our approach in various scenarios. A manageable overhead would allow us to run more tests to achieve a better coverage. If the measured overhead is low enough and suitable in a production environment, we can apply our approach in the field to extract specifications based on real-world workload.

**Characterization and Application.** We manually study all the generated specifications to understand their characteristics and how they can facilitate program understanding and future inter-request analyses. We discuss the lessons learned and the experience of using the specifications for tuning object cache policies.

4.4.1 Subjects

We evaluated our approach on two open-source Java-based web applications: ITracker [53] (version 3.3.2), an issue-tracking system, and OpenMRS [90] (version 1.11.5), a medical-record system.

ITracker has about 37,000 lines of Java code and about 8,000 lines of view-template code. It uses Struts [11] as the supporting framework. The request-handler methods defined under Struts follow the convention that method inputs and outputs are passed through special objects, such as HttpRequest objects. For the evaluation, we randomly generated a data set containing 10 developers, 20 projects, and 200 bugs randomly distributed among the 20 projects.

OpenMRS has about 204,000 lines of Java code and about 25,000 lines of view-template code. It uses Spring [107] as the supporting framework, and it contains request-handler
methods defined in three different forms: (1) methods using special objects to pass inputs and outputs, (2) a workflow-based model requiring multiple methods to process a single request (e.g., a method to create a view and another method to create data objects), and (3) a modern model passing method inputs and outputs directly through method parameters by simple values and collection-based objects. We use an anonymized data set with 5,000 patients and 500,000 observations.

These two applications use two widely adopted request-based frameworks in a non-trivial way. The two frameworks have different programming interfaces and paradigms, which lead to differences in the implementation of request-handler methods. We use the two applications to show the generality and usefulness of our approach on applications using different frameworks.

### 4.4.2 Experiment Design

To generate concrete application executions, we have implemented a random workload generator using Selenium [103], a web-browser based testing framework. The generator drives a web browser to simulate user actions on web pages, such as clicking hyperlinks. These actions can trigger requests to the applications. In particular, the generator identifies actionable elements on each page, and then performs an action on a randomly selected element. The identified elements include hyperlinks, web forms, input elements, and so on. For each element, the generator applies a corresponding action: clicking for a hyperlink or a button, and generating random but domain-specific data for a web form. With these capabilities, the generator can keep running to cover a wide range of request sequences on the subject applications. Given the random nature of the generator, the workload it generates may not cover all possible real-world request sequences. Therefore, our results need to be interpreted with the random workload in mind.

We configure the generator with 10 distinct random seeds to exercise different request sequences in the subjects. For each seed configuration, we run the generator 10 times. Each run starts with the same database state and consists of 500 simulated user actions. Our results show that this setup allows our approach to discover most of the possible and valid specifications. Although we aim at producing repeatable results, the generator may encounter nondeterministic and unexpected behaviors from Selenium and the network. So we choose a relatively small number of actions for each run to reduce the potential divergences on the request sequences, which are supposedly fixed by each random seed,
Table 4.2 Specification accuracy of ITracker and OpenMRS.

<table>
<thead>
<tr>
<th>Subject (Window Size)</th>
<th># of Specs</th>
<th># of True Specs</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITracker (reflective)</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>ITracker (2)</td>
<td>64</td>
<td>54</td>
<td>84.38%</td>
</tr>
<tr>
<td>ITracker (3)</td>
<td>16</td>
<td>14</td>
<td>87.50%</td>
</tr>
<tr>
<td>ITracker (4)</td>
<td>2</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>ITracker (5)</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>ITracker (Total)</td>
<td>82</td>
<td>68</td>
<td>82.92%</td>
</tr>
<tr>
<td>OpenMRS (reflective)</td>
<td>15</td>
<td>15</td>
<td>100.00%</td>
</tr>
<tr>
<td>OpenMRS (2)</td>
<td>20</td>
<td>17</td>
<td>85.00%</td>
</tr>
<tr>
<td>OpenMRS (3)</td>
<td>32</td>
<td>20</td>
<td>62.50%</td>
</tr>
<tr>
<td>OpenMRS (4)</td>
<td>11</td>
<td>1</td>
<td>9.09%</td>
</tr>
<tr>
<td>OpenMRS (5)</td>
<td>12</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>OpenMRS (Total)</td>
<td>90</td>
<td>53</td>
<td>58.89%</td>
</tr>
</tbody>
</table>

and repeatedly run each configuration to get stable results. Nevertheless, our reported results may still be affected the likely nondeterminism.

To evaluate specification accuracy, we use a sliding window with an increasing size to process the generated traces. Starting from size two, we keep increasing the window size until our manual inspection determines that the increased window does not yield new true specifications not seen in smaller windows. In the results, we only consider specifications observed at least 100 times with a confidence (defined in Section 4.3.3) of at least 0.2. These thresholds are chosen to ensure that most true specifications would likely be reported, while most false specifications would likely be filtered out. We consider the reported results worth further inspection.

For tracing overhead, we record the processing time for each request on the server side during experiment runs. Then we calculate the overhead by comparing the recorded time of corresponding requests on the instrumented and uninstrumented subjects. We include requests from repeated runs to amortize the likely nondeterminism and performance anomaly.

4.4.3 Specification Accuracy

Table 4.2 presents the accuracy results for ITracker and OpenMRS in different sliding-window sizes. Each row shows the statistics of unique specifications that are not found in
smaller windows. The column “# of Specs” shows the total specifications in the window of size shown by the first column, and “# of True Specs” shows the total true positives that we manually identified. Note that a specification may be found in windows of different sizes, so we attribute it to the smallest window. For example, “ITracker (3)” shows statistics of the specifications that can only be found in window size three or above, and “OpenMRS (reflective)” indicates specifications found between request-handler methods that process the same request. As a result, we observe decreased numbers of specifications in larger windows.

The results show that our approach is effective and can reach a balance between testing efforts and result accuracy with the reporting threshold. We find that increasing the window size does produce more specifications. The additional specifications are either infrequent or not discovered under smaller windows. On the other hand, a larger window introduces more false positives, which lower the overall accuracy. As we can observe from the results, a break point exists from which we may not have more true unique specifications. For instance, the accuracy of ITracker is steady under windows of sizes two and three, but no new true specifications are found in larger windows. This observation suggests that most specifications can be found in small windows, and we can determine the proper window size by gradually increasing the mining window.

We also look into the false positives to investigate the causes. In our mining step, any coincidentally equal output and input values would lead to false positives. Confidence values can mitigate such false positives to some extent. However, a specification with a confidence of 100% does not necessarily mean a true positive, e.g., some of the false positives in OpenMRS have very high confidence values. This is usually due to a combination of overlapped value ranges and insufficient data points of different kinds of data objects. For instance, multiple distinct data objects in OpenMRS, such as hospital locations and question forms, have the same range of identifier values, and sometimes appear in the same output. When only a small number of such data objects exist in the database, the chance of overlapped identifiers increases. Using more diverse data sets (e.g., real-world data sets) can mitigate such problems.

4.4.4 Tracing Overhead

The tracing overhead on the total execution time from all experiment runs is 74.96% or ITracker, and 33.43% for OpenMRS. Figure 4.3 presents the overhead results by requests.
Each bar represents the number of requests whose overhead falls into the range indicated by the x-axis, e.g., 100% means the range from 75% (exclusive) to 100% (inclusive). Due to the space limit, we aggregate all the requests whose overhead is above 200% into the last bar. Also note that the total number of requests for each subject is not exactly 50,000, because some user actions may not trigger requests, and some web pages may trigger background requests automatically.

Overall, the results show that the overhead of nearly 83% of all requests falls within 100%. Given the current overhead measurements, we conclude that our approach is suitable for in-house testing and feasible to be selectively applied in a production environment with less strict performance requirements.

We also observe some requests with high overhead that is related to the large number of object accesses during request processing. In ITracker, a frequent request path of high overhead is /list_issues with a median overhead of 166.67%. To show a list of bugs, the view template accesses every bug object multiple times to retrieve the bug identifier, name, and other information, with an average of 695 method invocations per request on those objects. The situation is similar in OpenMRS, e.g., /concept.htm with a median overhead of 75.57% and 488 invocations per request on average.

### 4.4.5 Characterization

The mined specifications allow us to investigate common characteristics of data dependencies across requests in the level of request-handler methods. We next discuss our findings regarding the propagation and use of inter-request data.
Among 121 true specifications, 86 of them show the propagation of entity identifiers, such as the identifier of a patient in OpenMRS or a project in ITracker. OpenMRS and ITracker store domain-specific entities with identifiers in databases, and create data objects representing these entities when accesses are needed. To achieve certain features, these applications may need the same entity in processing multiple related requests. For instance, ITracker needs the same piece of project information in processing requests /list_issues, which lists all issues in the project, and /view_issue, which shows a particular issue. To share information across requests, these applications choose to propagate entity identifiers instead of data objects. On incoming requests, request-handler methods receive these identifiers and then use them to recreate the corresponding data objects from the database.

For the rest of the specifications, 15 of them represent data dependencies between request-handler methods that process the same request (e.g., getVisit() and showForm() in Figure 4.2). As such methods are executed in the same context without going across the client side, they directly propagate data objects to the supporting framework, which further passes the objects to other methods.

The remaining 20 specifications involve values for pagination and flags, and the propagated objects are not instances of application-defined classes. Examples include an integer value indicating the number of total data entries, which allows a method to calculate how many remaining entries need to be displayed, and a string instructing the next action of a receiving method.

4.4.6 Application: Tuning Cache Policies

As revealed by our characterization study, entity identifiers derived from data objects are common in inter-request data propagation. The request-handler methods tend to use the propagated identifiers to recreate the corresponding data objects. Such behaviors lead to repetitive object creation, which can be expensive, because it usually involves database queries.

The mined specifications can help pinpoint repetitive object creation between related request-handler methods. For the example in Figure 4.2, a hypothetical analyzer could work in the following way: (1) applying dataflow analysis starting from renderDashboard() to identify the creation of the Patient object (line 7) and its output site (line 8), (2) using the specification on renderDashboard() and getVisit() as a summary to confirm that the parameter patientId is an alias of patientId retrieved from the Patient object,
(3) continuing the analysis into getVisit() to identify the creation of another Patient object using the same patientId (line 35). The analyzer could then infer that the two Patient objects represent the same entity, and suggest an optimization strategy, e.g., caching the first Patient object.

We follow the methodology of this hypothetical analyzer to manually optimize repetitive object creation in our evaluation subjects by leveraging the object cache built in Hibernate [50]. Hibernate is used by both subjects as their data-access layer, and its cache is controlled by configuration files specifying classes of objects to be cached and their caching lifetimes.

We first rank the mined specifications by their observed times to identify data objects whose identifiers are frequently propagated across requests. The top identified objects are: Issue and Project for ITracker, and Concept, Obs (Observation), Patient, and Visit for OpenMRS. In ITracker, Issue and Project objects are not cached, and we directly create a new long-term cache policy for them. In OpenMRS, there is a default cache policy applying a long-term cache policy with a 12,000-second lifetime for Concept objects and a short-term policy with a 30-second lifetime for Patient objects. However, the cache of Concept objects is insufficient, as the objects referenced by Concept objects (e.g., ConceptName and ConceptDescription) are not included in the default policy. In addition, the lifetime for Patient objects is too short for multiple requests of related patient features. Therefore, we amend the default cache policy to include Obs and Visit objects, objects referenced by Concept objects, and change the policy for Patient objects to be long-term.

We run the same experiment described in Section 4.4.2 with and without the updated cache policies, and we measure both database queries and execution time. A reduction in database queries indicates that fewer data objects are created by querying data from a database, as the required objects are cached in the application. The results show an average query reduction of 8.50% for ITracker and 5.05% for OpenMRS under our random workload, and the reduction in the total execution time is 3.32% for ITracker and 2.80% for OpenMRS. The presented execution time is measured in an experimental setup where the database server and application server are deployed on the same machine. We expect a greater time reduction in a realistic setting where the network latency in accessing the database introduces more overhead in the overall execution time. On the other hand, the query count will remain stable, as it is less sensitive to instabilities in the execution environment.
4.5 Related Work

4.5.1 Analysis of Web Applications

There is a wide range of work on web applications, such as testing [8, 14, 71, 74, 75, 101, 123, 146], program slicing [89, 120, 122], and security analysis [3, 52, 84, 108, 121, 127]. The proposed approaches perform their analysis based on some forms of data- or control-flows.

Some of the approaches [3, 13, 14, 74, 89, 120] extract page navigation and/or data-flows across web pages. These approaches commonly rely on a language-specific model to analyze program code (e.g., PHP) that directly generates dynamic page content. With a modernized point of view on framework-based web applications, our approach focuses on request-handler methods and their data dependencies, eliminating the involvement of web pages. We make this choice because framework-based web applications are popular nowadays and they expose request-handler methods as application entry points instead of web pages.

Some other approaches, such as TAJ [122], F4F [108], and ANDROMEDA [121], perform analysis on framework-based web applications. These approaches require framework-specific modeling, and treat each request-handler method as a single entry point of analysis. On the contrary, our approach is not tied to specific framework implementations, and can enhance these approaches by providing inter-request data dependencies for end-to-end analysis.

4.5.2 Program Specification Mining and Its Applications

Existing work has shown program specifications inferred from traces to be useful in program analysis, such as data-flow specifications [35, 37] for complex libraries and dynamic language features, and specifications in finite state machines [22, 25, 44, 67, 68, 80] for component interactions, API usages, and temporal invariants.

With the purpose of inter-request analysis, our inferred specifications describe data dependencies between request-handler methods that are modularized by underlying frameworks. Our evaluation has shown a use case of tuning cache policies to reduce database operations. Future work can use our specifications to automate such optimization and perform more fine-grained analysis in combination of many existing approaches [83, 92, 116]. For instance, Tamayo et al. [116] propose an approach to construct dependency graphs for
database operations. Ramachandra et al. [92] propose an approach to identify database operations in the interprocedural scope to enable prefetching. Our specifications can enable these approaches to perform inter-request analysis for more optimization opportunities. As another example, we also envision that our specifications can allow information-flow analysis to detect security vulnerabilities and malicious flows across multiple requests.

4.5.3 Dynamic Taint Tracking

Our tracing approach is similar to dynamic taint tracking, which incurs a non-negligible runtime overhead. Dytan, a generic taint tracking system [36], reports an overhead from 3000% to 5000%. Phosphor, a recent system for Java [21], incurs an average overhead of 53.31% and 220% at worst. We consider the overhead of our approach to be on a par with these systems.
In this dissertation, we have presented various data-driven techniques working on logs and traces for understanding and debugging complex software systems. Depending on specific analysis purposes and system characteristics, our techniques convert logs and traces into suitable models for further analysis, and then use proper representations to present analysis results. We have discussed various kinds of data modeling and result representations in our techniques targeting at different systems. In particular, we use a graph-based model for calling dependencies and lock contentions in Windows device drivers, an automaton-based model for execution workflows across distributed components with both synchronous and asynchronous communications, and a sequence-based model for object accesses in complex framework-based applications. Based on these modeling providing key information, we design offline or online algorithms to analyze the data. Depending on analysis purposes, we present analysis results in proper representations to either enhance human understanding of system behaviors and related problems, or tool capabilities on hard-to-analyze system components.

With the presented techniques, we are able to show the usefulness of data-driven analy-
sis in understanding and debugging complex software systems. On real-world execution traces, we can effectively conduct performance analysis to identify performance problems hidden deeply in the Windows kernel. CLOUDSeer allows us to monitor the whole cloud infrastructure in the workflow level with only general logs, and CLOUDSeer provides structured workflow information to facilitate problem diagnosis. With a customized tracing, DATAFLOW TUNNELING infers data dependencies that are hidden by complex request-based frameworks and client-server communications, and the resulting dependencies can enhance existing program analysis tools to work better on request-based applications.

5.1 Future Work

With our experience of applying data-driven analysis on software systems, we foresee and discuss the following future directions.

5.1.1 End-to-end Modeling of Cross-component Flows

Through this dissertation, we have discussed how various component interactions in complex software systems can cause different kinds of problems, and how data-driven analysis on runtime data can complement conventional program analysis in understanding such interactions and their related problems.

As a fundamental part of data-driven analysis, modeling control and data flows across components is a direction that is worth continuously exploring. Our existing work has addressed some parts in the direction: (1) calling and synchronization-related behaviors in device drivers [136], (2) task workflows across distributed components [138], and (3) data dependencies between highly modular request-handler methods [137]. A further step into this direction can be including and combining more components into the modeling. Ultimately, we expect to achieve end-to-end control and data flows across components from high-level applications to low-level operating systems.

Along this direction, we need to address some significant challenges coming from open systems, in which interconnected and interdependent components may provide different kinds of runtime data. Currently we rely on our expertise in such system components to select and model useful data. However, that expertise-based approach may not scale well to the prospective end-to-end modeling due to difficulties in understanding all the involved components before modeling. A better approach is automatically determining what data
can be useful without prior knowledge. Some existing work has attempted such automation for different purposes. For example, Log20 [143] adds log statements in optimal places for better debugging effectiveness and efficiency. PerfGuard [66] automatically determines program units under workload and their performance assertions.

With useful runtime data determined, extracting required information from data noises in open systems is also challenging. Unlike what we have done in this dissertation, more sophisticated machine learning techniques are required to address this challenge. Recently there have been advancements of constructing flows from log data using energy optimization with Monte Carlo Markov Chain [130] and Long Short-Term Memory [40]. In future, we plan to investigate along this direction to develop techniques for automated modeling with high precision.

5.1.2 Reproducing Problematic System Behaviors

With our techniques, we can identify problematic behaviors in device drivers that lead to performance problems, and components that are likely to cause failures in cloud infrastructures. However, it is not straightforward to reproduce such problems when developers have the need of examining and repairing buggy code. A missing piece is about what and how external characteristics (e.g., environments and input) can cause observed problems.

A further step is identifying and inferring connections between external characteristics and problematic system behaviors. Such connections can be useful to guide developers in reproducing problems. Although there have been techniques leveraging symbolic execution to reproduce program failures, such as BugRedux [58], they may not scale well to complex systems due to their intrusive nature. Recently Pensieve [142] has set a good start to use logs to guide the search of failure program paths. Our future work can adapt this non-intrusive idea.

5.1.3 Applications of Inter-request Analysis on Web Applications

Dataflow Tunneling reveals the hidden data dependencies between request-handler methods. With proper inter-request analysis, we believe that such dependencies can benefit multiple areas in improving web applications.

Testing Separation of Front-end and Back-end. Testing web applications involves extra complexities due to the co-existence front-end web pages and back-end application logic.
The current practice treats front-end web pages as starting points, and tests each feature on every web page as thorough as possible. These page features would in turn invoke back-end application logic during testing. The combination of front-end pages and back-end logic could lead to a low coverage on the back-end, especially in the state-of-the-art application design that highly separates the front-end and the back-end. That is because there is no longer a clear correspondence between a feature in the front-end and a part of application logic in the back-end. With Dataflow Tunneling and the inferred data dependencies, we expect to see techniques that can split test cases into front-end tests, which test the web pages only; and back-end tests, which use inter-request data dependencies to directly orchestrate related request-handler methods in different ways to achieve a high coverage.

**Inter-request Performance Optimization.** As we have discussed in Chapter 4, the modular design of request-handler methods could introduce side effects on application performance due to repetitive or uncoordinated operations, such as expensive database queries. To address such performance problems, new techniques based on inter-request data dependencies may reorganize expensive queries spread across related request-handler methods. There can be different strategies of reorganization, such as query prefetching, which executes an expensive query asynchronously before its result is required; and query batching, which groups related queries together to reduce overhead from multiple round trips between the application and the database. Some previous techniques have adapted query prefetching [27, 28, 83, 92], but they mostly work within methods that process a single request. Future work can explore further optimization in the inter-request scope.
BIBLIOGRAPHY


Bodden, E. Soot - A framework for analyzing and transforming Java and Android Applications.


Cheung, A., Madden, S. & Solar-Lezama, A. “Sloth: Being Lazy is a Virtue (when Issuing Database Queries)”. Proceedings of the 2014 ACM SIGMOD International


Winterhalter, R. *Byte Buddy*.


Yu, X., Han, S., Zhang, D. & Xie, T. “Comprehending Performance from Real-world Execution Traces: A Device-driver Case”. *Proceedings of the 19th International


