Generalized techniques for using system execution traces to support software performance analysis

Thelge Manjula Peiris
Purdue University

GENERALIZED TECHNIQUES FOR USING SYSTEM EXECUTION TRACES TO SUPPORT SOFTWARE PERFORMANCE ANALYSIS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Thelge Manjula Peiris

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2016

Purdue University

West Lafayette, Indiana

To my parents, my wife *Kanchana* and our daughter *Senadi*.

ACKNOWLEDGMENTS

First, I would like to express great thanks to my advisor and my great mentor Prof. James H. Hill. I am very proud to say that I am his first Ph.D. student and I am really lucky to have worked under him throughout my tenure at IUPUI. I have undoubtedly learn more from Prof. Hill than from any other single person throughout my life. His guidance on doing research, writing papers, presenting the work, writing clean code, overcoming roadblocks and much other advice brought me to where I am now. His confidence and trust in me always encouraged me to get closer and closer to the goal.

Secondly, I would like to acknowledge Prof. Xiangyu Zhang for serving as the co-advisor in my dissertation committee. His advise and suggestions helped significantly in improving the research work proposed in this dissertation.

I would also like to thank Prof. Mohammad Al Hasan for serving as a committee member in my dissertation committee. His advice on applying certain data mining techniques were invaluable given that my background was not in the data mining field, especially at the start of this dissertation work.

I would also like to thank Prof. Buster Dunsmore for serving as a committee member in my dissertation committee. The advice and guidance he provided on dissertation definitely help me to improve the quality of work.

Most of my research work is sponsored by Australian Department of Defence. Therefore, I am really thankful to them for the funding provided especially during the early stages of my research. I would also like to thank all the faculty and non-faculty members at Department of Computer Science at IUPUI for the financial and non-financial support given. The teaching and research assistantships helped me to learn many new things and get invaluable experience. Special thanks should go to Nicole for all the assistance provided on non-academic matters.

I would also like to thank my friends at the SEDS lab for invaluable comments and help provided to achieve my research goals. I would specially like to acknowledge Dennis, Lahiru and Dimuthu for their friendship and also for great suggestions to shape my research. Also, I will never forget all my other friends in USA and in Sri Lanka, who helped me especially during tough times.

I would also like to express my deepest gratitude to Dr. Sanjiva Weerawarna (who himself is a PhD graduate from Purdue Computer Science Department), CEO and Chairman of my previous company WSO2. He encourages me to pursue computer science PhD from a graduate school in USA. Without his recommendations and guidance, this would have been a dream.

I will never forget all my teachers in my primary/high schools, University of Moratuwa and at IUPUI. Every single thing I learned from them has contributed to my thinking process and for this dissertation work.

I would not have come this far in education without the support of my father, mother, and sister. We had real hard times financially when I was at high school and also when I was in the college during my undergraduate studies in Sri Lanka. My father and mother always assured that none of those hard situations stopped my journey in education. Their patience, unconditional love and encouragement has been one of the main motivations throughout my life.

Last but not least, I would like to express my great thanks to my wife Kanchana and my daughter Senadi. If anyone has shared all the difficulties during the last five years, that's my wife. Her never ending love, patience and the sacrifice is hard to imagine. She assured that I am physically and mentally tough during whole time. My one year old daughter Senadi always kept me busy, wanting me to play with her, read, walk and many other things, which help me to get a break from work and to overcome some stressful periods. I cannot imagine a world without these two ladies.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

DBI	Dynamic Binary Instrumentation
XML	Extensible Markup Language
ORB	Object Request Broker
SOAP	Simple Object Access Protocol
UNITE	Understanding Non-functional Intensions via Testing and Experimentation
CUTS	Component Workload Emulator (CoWorkEr) Utilization Test Suite
SETAF	System Execution Trace Adaptation Framework
DMAC	Dataflow Model Auto Constructor
EMAD	Excessive Dynamic Memory Allocations Detector

ABSTRACT

Peiris, Thelge Manjula PhD, Purdue University, May 2016. Generalized Techniques for Using System Execution Traces to Support Software Performance Analysis. Major Professor: James H. Hill.

This dissertation proposes generalized techniques to support software performance analysis using system execution traces in the absence of software development artifacts such as source code. The proposed techniques do not require modifications to the source code, or to the software binaries, for the purpose of software analysis (non-intrusive). The proposed techniques are also not tightly coupled to the architecture specific details of the system being analyzed. This dissertation extends the current techniques of using system execution traces to evaluate software performance properties, such as response times, service times. The dissertation also proposes a novel technique to auto-construct a dataflow model from the system execution trace, which will be useful in evaluating software performance properties. Finally, it showcases how we can use execution traces in a novel technique to detect Excessive Dynamic Memory Allocations software performance anti-pattern. This is the first attempt, according to the author's best knowledge, of a technique to detect automatically the excessive dynamic memory allocations anti-pattern. The contributions from this dissertation will ease the laborious process of software performance analysis and provide a foundation for helping software developers quickly locate the causes for negative performance results via execution traces.

1 INTRODUCTION

Performance is a crucial non-functional property in any software system. Slow and inefficient software frustrate users and can cause financial losses [1]. In certain real time systems, failure to achieve performance requirements can be interpreted as functional failures of the system. On the other hand, improving software performance is also critical, even when the software is meeting its performance requirements. This is because efficient software systems can attract many users and generate lots of revenue. Therefore, it is very important to develop techniques to analyze the performance of software systems. Analyzing software performance includes not only analyzing performance properties such as response times, service times and throughputs of a software system; but also providing feedback to software developers about software performance results.

System execution traces or simply execution traces[1] are one artifact that has been used by researchers and practitioners to analyze functional and non-functional properties of software systems. There are several methods to generate execution traces such as:

1. Compiling the original source code with an instrumentation code and executing the combined executable [2–4];

2. Collecting execution traces from dynamic binary instrumentation where the instrumentation is performed at run time on the compiled binary files [5];

3. Collecting log messages (*i.e.*, traces generated from log statements in the source code) during system execution [6–10]; and

[1]System execution traces and execution traces refer to the same entity throughout this document

4. Registering for certain events of the target system and generating messages whenever the events occur [11–13].

Execution traces have been used on functional aspects of the system, such as detecting system failures [14], operational profiling [15], and website usage patterns based on user sessions [16]. However, recent research efforts have focused on using system execution traces for software performance analysis [2, 3, 6, 9, 11, 17] .

One benefit of using execution traces to analyze software system performance properties is execution traces provide a comprehensive view of the system's behavior and state throughout the system's execution lifetime. This is opposed to a single snapshot of the system at a given point in time, such as a global snapshot [18]. Likewise, they provide system testers with a rich set of data for analyzing data trends associated with a given performance property, *i.e.*, how a given performance property changes with respect to time. Moreover, analyzing execution trace data has the advantage over analyzing performance counter data [19–23] because the latter does not provide behavioral aspects of the system. Furthermore, it is hard to assume that every system provides a complete enough set of performance counters for analysis. The Appendix A presents techniques for analyzing performance counter data in system execution logs to detect performance anomalies.

Most systems however have some kind of tracing technique that can be utilized for performance analysis. Therefore, the next section summarizes different research efforts and some unresolved challenges of using system execution traces for software performance analysis.

1.1 Trends and Challenges of Using Execution Traces for Performance Analysis

Recent research efforts [2, 6, 9, 11] have tried using execution traces for software performance analysis. However, following challenges still remain and need to be addressed when using execution traces for software performance analysis.

1. **Relaxing the assumptions about the execution trace and the system generated it.** Some approaches for using system execution traces to analyze software performance rely on methods that require access to the system's original source code [2,3]. Therefore, it is hard to apply these approaches when the source code of the system is not available, which is the case for most systems built from third-party off the shelf components.

   Another set of approaches for validating performance of the system using execution traces are tightly coupled to system's architecture and technology [11–13]. Finally, the approaches that are not implementation dependent require execution traces to be generated in a certain format [6,7]. Moreover, such approaches are not trying to utilize system log messages, but rather enforce the system implementers to use the provided logging mechanisms or convert the system logs to an intermediate format.

   Understanding Non-functional Intensions through Testing and Experimentation (UNITE) [9] is a tool and a technique that does not rely on the requirements mentioned in the above paragraph to analyze software system performance properties. UNITE accomplishes this feat by using dataflow models that describe causal relationships between event types—not event instances—in the system. This allows UNITE to operate at a higher level of abstraction that remains constant regardless of how the underlying software system is designed, implemented, and deployed (*i.e.*, the mapping of software components to hardware components). More details on UNITE is provided in Chapter 3.

   Although it is possible to analyze performance properties via system execution traces using tools like UNITE (without any restriction on log message format), it is assumed that execution traces contain several properties, *e.g.*, identifiable keywords, unique message instances, enough variations among the same event types to support performance analysis. Moreover, the dataflow model used to analyze the system execution trace must contain several properties, *e.g.*, identifiable log formats and unique relations between different log formats. If planned early enough in the software lifecycle, it is possible to ensure these properties exist in both the dataflow model and generated system execution trace.

Unfortunately, it is not possible to *always* ensure that system execution traces contain the properties required to support performance analysis via UNITE. Therefore, developing techniques to analyze software performance properties using execution traces, which are not tightly coupled to (1) the source code of the system; (2) the architecture and platform of the system; (3) the structure of the messages in the execution trace; and (4) certain properties in the execution trace will be very useful. This will help software testers to use execution traces for performance analysis (1) with no or minor modifications to the analysis tool and (2) with no modifications to the target system.

2. **Minimizing the manual analysis of system execution traces to construct dataflow models.** Although UNITE provides a foundation for analyzing software system performance properties via execution traces, it is both tedious and time-consuming for software system testers to define the required dataflow model. Likewise, as software systems increase in both size (*i.e.*, source lines of code and number of software/hardware components) and complexity (*i.e.*, set of features), it becomes *harder* to manually define the dataflow model. This is because software testers have to analyze the entire system execution trace before defining the dataflow model, and may have to refer the source code of the system to gain needed knowledge about the system.

   Therefore, the task of manually specifying the dataflow model can limit the applicability of dataflow model based performance analysis approaches like UNITE. Because dataflow model captures the abstract event types and the causal relationships between the event types, it will be very useful for software testers to understand the runtime behavior of the system at a very abstract level.

3. **Using the system execution traces in detecting excessive dynamic memory allocations.** Even though summarizing software performance property values using summary statistics such as means, variances and probability distributions provides valuable insights about the system performance, software developers and architects are interested in root causes to performance problems [24, 25].

   One such root cause is excessive dynamic memory allocations. Even though dynamic memory allocations provide software developers with memory flexibility at runtime, it is an expensive operation [26]. Allocation and dellocation (*i.e.*, the process of releasing dynamically allocated memory) of dynamic memory using standard memory allocation/deallocation functions like malloc/free (in the case of C) and new/delete (in the case of C++) require system calls. Too many dynamic memory allocations can have negative consequences on software performance. For example, Smith et al. [27] described and illustrated how excessive dynamic memory allocations is a software performance anti-pattern. *Software performance anti-patterns* [27, 28] are common designs that have negative impact on software performance.

   Because it is known that excessive dynamic memory allocations can have significant impact on software performance, many popular middleware and software applications, such as Apache Web Server [29], GCC compiler [30], and the PHP scripting language [31] use custom memory allocators as a solution to the excessive dynamic memory allocations problem. Moreover, some popular applications servers provide guidelines on how to reduce object creation and deletion. For example, in WebSphere application server some of the best practices to improve the server performance are directly/indirectly related to creation and deletion of objects [32].

It is also important for newly created systems to detect excessive dynamic memory allocations anti-pattern before adapting an existing solution approach to resolve this anti-pattern. On the other hand, failure to detect and remove excessive dynamic memory allocations will cause system developers to seek alternative methods for resolving it, such as adding more memory and processing power to coup with slow performance.[2]

Because excessive dynamic memory allocation can have negative impact on performance, it needs to be detected and resolved. Unfortunately, the most prominent (and reliable) method for detecting and resolving excessive dynamic memory allocation—and actually any software performance anti-pattern—is (manual) source code analysis [33]. This approach, however, requires expert domain knowledge. More importantly, it requires access to the original source code to support the necessary analysis.

There are existing approaches for detecting software performance anti-patterns that do not rely on source code. However, these approaches have categorized excessive dynamic memory allocations as an undetectable software performance anti-pattern [24, 25, 34]. This is mainly because these approaches are either architecture dependent [34] or rule-based approaches [24, 25], which do not consider the behavioral aspects of software performance anti-patterns at runtime. These forms of analysis make it *hard* to detect implementation level anti-patterns like excessive dynamic memory allocations [35]. On the other hand execution traces can be used to understand behavioral aspects of the systems. Therefore, techniques need to be developed to detect excessive dynamic memory allocations anti-pattern utilizing execution traces.

[2]This thesis advisor Professor James Hill experienced this first hand as a visiting researcher at EBay, Inc in 2007. He was responsible for optimizing the backend search engine for EBay, Inc to address known performance issues. The proposed solution was to remove excessive dynamic memory allocations that were requesting 0 bytes of memory. The solution resulted in 99% improvement in performance for test scenarios that were missing their deadline and 10-15% improvement in performance for scenarios that were not missing their deadline. This allowed EBay, Inc. to address the root cause of the problem.

1.2 Research Approach.

To address the challenges identified in Section 1.1, and for better use of execution traces, we advance the current state of the art of using execution traces in software performance analysis by,

1. **Improving the current techniques of using system log messages to evaluate software performance properties.** The novel contributions of this aspect include an adaptation technique of dataflow models, associated with execution traces, *i.e.* when execution traces do not contain the properties required for performance analysis and a method to auto-generate the dataflow model associated with execution traces.

2. **Providing a novel technique to detect excessive dynamic memory allocations software performance anti-pattern.** This will enable software developers to quickly locate the excessive dynamic memory allocations without doing tedious source code analysis.

In the following, we briefly summarize the three different but synergistic contributions proposed by this dissertation:

1. **Adapting the dataflow models associated with execution traces,** which includes a technique that enables, using system execution traces for software performance analysis when the execution traces does not contain the properties discussed in Section 1.1.

   The novel contributions of this research includes (1) a framework for adapting dataflow models associated with the execution trace without changing the source code of the software system or the execution trace; and (2) a domain specific language for software testers to write different adaptation modules. Chapter 4 discusses this adaptation technique in detail.

2. **Auto-constructing dataflow models from execution traces,** which includes algorithms and techniques for constructing the dataflow model given an

execution trace, with minimum user intervention. This research effort provides capabilities for (1) finding abstract event types of an execution trace, and (2) finding the causal relationships among abstract event types and their variable parts.

The novel contributions of this research include two algorithms for finding abstract event types and causal relationships using data mining techniques and probabilistic approaches. Chapter 5 discusses these techniques in detail.

3. **Detecting excessive dynamic memory allocations anti-pattern using system execution traces,** which includes a technique to detect excessive dynamic memory allocations anti-pattern without using the source code of the system.

   The novel contribution of this research include an algorithm for constructing the execution call graph of a runtime system using a system execution traces generated from Dynamic Binary Instrumentation (DBI) and a method to find routines in the source code that perform excessive dynamic memory allocations.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 discusses existing research related to the algorithms, analytics, patterns, and tools we present in later chapters; Chapter 3 discusses UNITE performance analysis tool, because some of the concepts and techniques proposed in this proposal have emerged from UNITE and the dataflow model concept it has proposed. Chapter 4 discusses adapting execution traces for software performance analysis. Chapter 5 discusses auto-generating dataflow models from execution traces. Chapter 6 discusses detecting excessive dynamic memory allocations software performance anti-pattern using execution traces. Finally, Chapter 7 provides concluding remarks, lessons learned and future research directions.

2 RELATED WORK

This chapter discusses current research related to our research on techniques for using system execution traces to support software performance analysis. Some of our research work on utilizing system execution traces to evaluate software performance properties is built on top of techniques and concepts mentioned in [9]. Therefore, we described UNITE's technique in a separate chapter (Chapter 3). The remainder of this chapter describes other related work and is organized as follows: Section 2.1 discusses related research on using execution traces to evaluate software performance properties; Section 2.2 discusses existing techniques for building models for software analysis using execution traces. Section 2.3 presents current research related to detecting excessive dynamic memory allocations software performance anti-pattern.

2.1 Approaches of Using Execution Traces to Evaluate Performance Properties

This section compares our research on adapting execution traces to support software performance analysis with other related research on using execution traces to achieve similar goals. Our research on adapting system execution traces to support software performance analysis is discussed in detail in Chapter 4.

2.1.1 Intrusive Instrumentation Approaches.

We categorize intrusive instrumentation as altering the source code of the system to collect traces for the purpose of software analysis. Several intrusive approaches can be found in the literature that use traces from instrumented source code for software performance analysis [2–4]. These approaches instrument the source code of the target system using methods defined in the performance analysis tool. A main requirement for these approaches therefore is the availability of the source code. Moreover, these approaches add overhead to the system being analyzed. In contrast, our research work uses log messages the system outputs during the system execution. We do not modify the logging infrastructure (source code related to the logging module) of the system for the purpose software performance analysis. This allows our approach to be applied when the source code of the originating system is not available.

TimeToPic [6] is a tool that can be used to visualize a system execution log. It also provides facilities to analyze different locations, such as points of interests in the visualization graph. This visualization graph can be used to analyze different performance properties. The main limitation in TimeToPic is, it relies heavily on log message format. Either the developer has to use the common message format TimeToPic has defined or implement the logger of the application using TimeToPic's logging API. This approach is easy to apply on newly developed systems, and *hard* to apply on existing systems because it requires changes to the existing systems. In contrast, our research technique on adapting execution traces does not have any restrictions on the log message format and tries to utilize the system log as much as it can.

2.1.2 Non-Intrusive Instrumentation Approaches.

Nagaraj et al. [7], propose a technique to comparatively analyze system logs to diagnose performance problems. Nagaraj's technique uses two execution logs: one that is called the baseline log and an erroneous log it assumes to have performance related issues. Nagaraj's technique compares the erroneous log with the baseline log and provides a report on the locations of performance problems. The main limitation of Nagaraj's approach is that the perl scripts that process to logs must be modified each time when it encounters a new system. This is because the execution logs of different systems are heterogeneous in nature. In contrast, our research does not depend on such comparative analysis and also the performance analysis tool doesn't need to be change for each different log.

Han et al. [8] propose a method called StackMine that uses stack traces collected by the Microsoft application monitoring tools to debug performance. Their performance monitoring tools send traces only when the response time of a method is above a certain threshold. Furthermore, Han's analysis is offline. The main difference between StackMine and our technique is that StackMine is operating system dependent because the performance monitoring tools they use are platform dependent. Other non-intrusive approaches for performance monitoring are architecture dependent. For example, previous research efforts have focused on proposing methods for performance monitoring in Enterprise Java Beans (EJB) applications [11–13]. These approaches, however, require users to deploy performance monitoring beans and are tightly coupled to the J2EE architecture.

2.1.3 Approaches to Overcome the Imperfectness of Log Messages

Cinque et al. [36], propose a technique for dependability evaluation of complex software systems using system execution traces. In their approach, they propose to follow a minimal set of logging rules during the design and development time. This set of rules guarantees that the system logs contain the required properties for dependability analysis of complex systems. For example, the main types of rules that Cinque et al. propose are enabling logging for service start/end and interaction start/end events. Even though this approach contributes to the generation of well-structured logs, it is highly unlikely that every system follows these rules. Current software development methodologies highly depend on software reuse. It therefore is highly unlikely to assume that this kind of structure is followed in complex software systems. Instead, We focus on tackling this problem of uncertainty in system execution traces using an adaptation technique.

Yang et al. [37], use system execution traces for dynamic inference. Similar to our research approach, Yang et al. also accepts the fact that system execution traces are imperfect. Likewise, Yang uses coding conventions to prune large number of unimportant properties for developers. We however, do not rely on such coding conventions. Instead, we provide a framework to capture execution semantics that are not reflected in the execution trace.

2.2 Techniques of Using Execution Traces to Build Models for Software Analyis

This section compares our research on auto-constructing dataflow models to support software performance analysis with other related research. The main research goals of below mentioned approaches are, (1) Abstracting event types from execution logs, similar to finding log formats in our work, (2) Building intermediate models which are similar to dataflow models defined in UNITE 3 for software system analysis. Our research on auto-constructing dataflow models to support software performance analysis is discussed in detail in Chapter 5.

2.2.1 Abstracting Event Types from Execution Logs

Qiang et al. [38] describes a technique to detect execution anomalies in distributed systems using unstructured log analysis. Similar to log formats in UNITE 3, Qiang et al. has a concept called *log keys*. They also consider the variable parts of log keys as parameters. They use empirical rules to identify log key parameters. Their intuition is that in log messages, log key parameters are often in the forms of numbers, URIs, and IP addresses. Qiang et al. also clusters similar log messages together. Edit distance (*i.e.*, the number of edit operations required to transform one word sequence to another) is used as the criteria for determining similarities between log keys. In contrast our technique does not rely on such assumptions when finding log formats. Instead, we employ an algorithm based on frequent-sequence mining to identify log formats and distinguish between variable and static parts.

Xu et al. [14] describes a technique for mining console logs for large-scale software system problem detection. Xu et al. extracts static and variable parts of a log message. Their approach, however, requires the original source code combined with the console log. Nagappan et al. [39] describes a technique to find log formats, without the original source code. They apply a frequency based approach to separate variable parts from static parts in log messages. The main drawback with their approach is variable parts are not identified when they have a frequency above the threshold frequency. We address this issue in our technique by iteratively employing the algorithm after pruning set of log messages using already found log formats.

Safyallah et al. [40] uses frequent-sequence mining to identify common functionality associated with feature-specific task scenarios based on data captured in a system execution trace. Safyallah et al. also uses frequent-sequence mining to discover relationships between different feature specific tasks in inter/intra modules. Their approach defines a frequent-sequence pattern (*i.e.*, execution pattern) as a contiguous part of an execution trace. This method however does not count the situations where, variable parts appearing in between static parts.

2.2.2 Building Intermediate Models from Execution Logs

Lou et al. [41] propose an approach on identifying dependences among events in distributed systems. In their work, they use a learning approach based on Bayesian decision model to construct a dependency graph using abstract log formats. They have used the trained model to detect dependency based errors in Hadoop. The training based approach makes it harder to apply their technique for broader set of software systems. Because, in most of the software systems it is hard to find enough training data (*i.e.* execution logs) to train a model. This will result in over fitting of models and low accuracies when applied to new data. Instead our approach is based on an evidence combination approach, which uses Dempster-Shafer theory and it does not require up-front training data.

Fischer et al. [42] describe a technique to track system evolution by analyzing the system execution traces. Their method locates execution patterns in the traces to determine how the software has evolved between different revisions (or versions). Their method also uses relational data model to store the traces and assign unique ids for analysis purposes. Our work differs from their work, because our work analyzes the performance properties of the system which are runtime properties as opposed to static properties like how software is changed between versions. Further, their method requires the system execution traces to have a particular format; however, we do not assume that kind of a format in our technique.

Safyallah et al. [40] describe a method of mining system execution traces to find out common functionality associated with feature-specific task scenarios (*i.e.*, core functions that implement software features) to discover relationships between different feature specific tasks in inter/intra modules (*i.e.*, inside a single software component and between different software components). Their approach does not need any particular format in the system execution traces. The main difference between Safyallah et al.'s work and our work is that we try to find causal relationships among different event types instead of relationships between feature specific tasks.

Voigt et al. [43] present a trace visualization technique for analyzing method calls and object access. Their main purpose is to understand the large execution traces as a sequence of object activities. They have tried to find out relationships in the traces, which are mapped to relationships as activities between objects. Our research effort differs from this work, because we are trying to do analysis at a higher level than object activities. Also our main focus is not on visualization.

2.3 Techniques Related to Detecting Excessive Dynamic Memory Allocations

This section compares our research on detecting excessive dynamic memory allocations with other related research efforts. Our research on detecting excessive dynamic memory allocations is discussed in detail in Chapter 6.

Automated approaches for detecting excessive dynamic memory allocations cannot be found in literature. Likewise, existing approaches for detecting software performance anti-patterns have categorized excessive dynamic memory allocations as an undetectable software performance anti-pattern [24, 25, 34]. Although there are several approaches for detecting memory leaks and memory access errors using Dynamic Binary Instrumentation (DBI) [44–47], excessive dynamic memory allocation problem has not been attacked by the research community.

2.3.1 Using DBI in Detecting Memory Related Issues and Performance Analysis

Chen et al. [48] have developed a tool called *MemBrush* that can be used to detect memory allocation/deallocation functions in stripped binaries. Their goal is to detect custom memory allocation routines and reverse engineer the memory management APIs. They have identified a set of characteristics for custom memory allocation routines and use DBI to identify these routines. Their approach is useful in detecting memory leaks and memory access errors, but they do not discuss detecting excessive dynamic memory allocations. Our technique expects a particular signature for allocation/deallocation routines. By combining the *MemBrush* approach with our approach, it may be possible to relax our assumptions about allocation/deallocation routines.

DBI has been used to identify the root causes of performance anomalies. For example, Attariyan et al. [49] proposes an approach to detect root causes of performance anomalies using DBI. They use DBI to monitor the software application as it is executing. The root causes they try to detect are human errors (*e.g.*, misconfigurations), not software performance anti-patterns (*e.g.*, excessive dynamic memory allocation). Menon et al. [50] uses DBI to diagnosis performance overheads in Xen virtual machine environments. The root causes they try to detect are related to I/O handling in virtual machine environments—particular related to TCP connections in virtual networks.

Detlefs et al. [26] provides a method to calculate the dynamic memory allocation costs of large C/C++ programs. Their measurements include CPU overhead and memory usage of different allocators. They also consider number of instructions executed by different allocation routines in their analysis. Their goal is to show the cost of dynamic memory allocations for different memory allocation algorithms. They, however, do not discuss on detecting excessive dynamic memory allocations from a performance perspective.

There are several research efforts on finding the object life times in garbage collected languages [51–53] using DBI techniques. In garbage collected languages the garbage collection process can happen at any time, therefore the timestamp at which an object is deleted cannot be used alone to approximately calculate the object lifetime. We believe that by integrating our approach with these precise object lifetime calculation techniques, we can extend our technique for applications created using garbage collected languages.

2.3.2 Techniques of Detecting Software Performance Anti-patterns

There have not been many research efforts on automatically detecting software performance anti-patterns. According to the study done by Din et al. [33], most of the approaches for detecting software anti-patterns are based on source code analysis. The few efforts, which have tried to detect software performance anti-patterns, without the use of source code are model based approaches [24, 25, 34]. The main characteristic of these approaches is that, they use annotated software architectures to detect software performance anti-patterns. The annotation is done using rules, which are similar to logical predicates. The annotated software architectures are simulated using models, such as Queuing networks, or sometimes using simulated code. The rules for describing software performance anti-patterns use performance matrices, such as resource usages and queue sizes, not behavioral patterns. Even though these approaches suggest several rules to detect many performance anti-patterns, they have failed to provide rules to detect excessive dynamic memory allocations. This is mainly because excessive dynamic memory allocations is a source code level architecture independent software performance anti-pattern.

Parsons et al. [54] used *association rule mining* to identify performance anti-patterns in Enterprise Java Bean (EJB) applications. They use deployment descriptors, instrumentation data, and rules to automatically identify the software anti-patterns. Their approach is language and architecture dependent (*i.e.* EJB). They collect and use three types of data to detect performance anti-patterns in EJB applications—information from the running application, information related to the server resources and contextual data. Anti-patterns are described as rules using the three different kinds of data. The rules are applied to analysis data to detect performance anti-patterns. The main limitation of Parsons et al.'s approach is it cannot be applied to detect architecture independent software performance anti-patterns like excessive dynamic memory allocations.

3 AN OVERVIEW OF UNITE

Understanding Non-functional Intensions through Testing and Experimentation [9] is a method and tool for analyzing execution traces and validating software system performance properties. UNITE's analytical techniques are not tightly coupled to (1) system implementation *i.e.*, what technologies are used to implement the system; (2) system composition, *i.e.*, what components communicate with each other; and (3) deployment, *i.e.*, where components are located. This is opposed to processing execution traces using simple scripts where those scripts are typically hard-coded for a specific use case, system, and/or problem. Another major advantage of UNITE, over other research efforts of utilizing execution traces for performance analysis is UNITE does not rely on a global identifier, such as a global clock, for event correlation. This is because UNITE uses data within the event trace that is common in both cause and effect messages, thereby removing the need for a global clock and ensuring that log messages (or events in a trace) are associated correctly.

Because UNITE is not a well-known tool, the remainder of this chapter provides a detailed overview of UNITE's capabilities and its current limitations.

Table 3.1.: An example execution trace displayed in table format as if being stored for offline analysis in a database.

Time of Day	**Hostname**	**Message**
2012-01-25 05:15:55	node1	Config: sent event 1 at 120394455
2012-01-25 05:15:55	node2	Planner: sent event 2 at 120394465
2012-01-25 05:15:55	node2	Planner: received event 2 at 120394476
2012-01-25 05:15:55	node1	Config: received event 1 at 120394480
2012-01-25 05:15:55	node3	Effector: sent event 3 at 120394488
2012-01-25 05:15:55	node3	Effector: received event 3 at 120394502

Table 3.1 presents an example execution trace from a software system. As shown in this table, the execution trace has log messages that correspond to sending/receiving events between components in a software system. The log messages in this example

contain time of the event, event id, and name of the component where the event occurred.

Software system testers use UNITE to analyze performance properties from the execution trace in Table 3.1 by first identifying what log messages to extract from the execution trace. These log messages should contain metrics of interest that support desired performance analysis. Once the log messages are identified, software system testers convert the common log messages into *log formats*. A log format is a high-level representation of a log message that captures both the static and variable portions of its corresponding log message. The static portions are those that do not change between different log messages. The variable portions are those that change between different log messages.

Given the execution trace in Table 3.1, Listing 3.1 shows the log formats that represents the different log messages in the trace. As shown in this listing, each log format (*e.g.*, LF_1 and LF_2) contains static and variable portions for extracting metrics from its corresponding log message in the execution trace. For example, LF_1 contains the variables: `cmpid`, `eventid`, and `sent`. The `sent` variable is used to extract the sending time. The remaining variables in the log format are used for correlating messages, which is explained next.

```
LF1: {STRING cmpid} sent event {INT eventid} at {INT sent}
LF2: {STRING cmpid} received event {INT eventid} at {INT recv}
```

Listing 3.1: Log formats for analyzing execution trace presented in Table 3.1.

A log format LF_i can have zero or more variables. We define the set of variables of log format LF_i as V_i. Similarly, the set of variables of log format LF_j is V_j. A *causal relation* $CR_{i,j}$ between two log formats LF_i and LF_j is denoted as $LF_i \rightarrow LF_j$ where LF_i is the cause log format and LF_j is the effect log format. This kind of a causal relation is also called a *log format relation*. A causal relation $CR_{i,j}$ can have zero or more variable relations. A *variable relation* $VR_{C,E}$ of a causal relation $CR_{i,j}$ is defined as $v_C = v_E$, where $v_C \in V_i$ and $v_E \in V_j$. A execution trace can have many log formats, many causal relations between the log formats, and many variable relations for each causal relation.

For the purpose of performance analysis, system testers can use subsets of the log formats, causal relations and variable relations, which we call a *dataflow model*. We formally define a dataflow model $DM = (LF, CR, VR)$ as:

- A set LF of log formats where each log format represents a set of log messages useful for analyzing a performance property;
- A set CR of causal relations that specify order of occurrence and causality among the log formats LF; and
- A set VR of variable relations attached to causal relations CR.

Like dataflow models in program analysis [55] where they relate variables across different source lines, dataflow models in UNITE relate log format variables across different log messages (or application contexts). The dataflow model then enables reconstruction of execution flows in the system (1) irrespective of system complexity and composition and (2) without a need for a global clock to ensure causality [18]. This is because the relations between the log formats preserve causality. For the execution trace in Table 3.1, Listing 3.2 illustrates the dataflow model.

```
Log Formats:
 LF_1: {STRING cmpid} sent event {INT evid} at {INT sent}
 LF_2: {STRING cmpid} received event {INT evid} at {INT recv}

Log Format Relations:
 LF_1 → LF_2

Variable Relations:
 LF_1.cmpid = LF_2.cmpid
```

```
LF_1.evid = LF_2.evid
```

Listing 3.2: Dataflow model for analyzing execution trace presented in Table 3.1.

The dataflow model illustrated in Listing 3.2 is a higher level abstraction of the execution trace being analyzed. Using this dataflow model and the execution trace, UNITE creates a variable correlation table based on variables defined in the dataflow model. A variable correlation table is a set of tuples $(d_1, d_2, .., d_i, .., d_n)$ where each tuple $d_i (i \leq n)$ contains instance values for all the variables of log formats defined in the dataflow model. The correlation of these values (*i.e.*, the values occur together in a single tuple) is defined by the variable relations in the dataflow model.

The previous work on UNITE [9] illustrates an algorithm based on topological sorting directed acyclic graphs and constructing variable correlation table for a given execution trace and a dataflow model. The log formats and corresponding log messages are processed in a certain order. This ordering is defined by the reverse topological ordering of the log formats in the dataflow model. This is because for a causal relation if we can find a log message instance representing an effect log format, then we should be able to find the corresponding cause log message instance from the same execution trace.

The reverse, however, is not always true. For example, there can exist a "sent" event without a "received" event. On the other hand, if there is a "received" event, then the corresponding "sent" event must exist in the same execution trace. Once the log formats in the dataflow model are topologically ordered, the source node (cause log format) comes first in the list. Because UNITE only considers cause log message instances that have the corresponding effect log message instance, UNITE has to process effect log message instances first.

For example, for any $LF_i \rightarrow LF_j$ log format relation, UNITE first processes the log messages that correspond to LF_j, and then log messages that correspond to LF_i. While processing in this order, UNITE populates the variable correlation table based on the values of the variables of each log message instance. For example, Table 3.2

shows the variable correlation table for the execution trace in Table 3.1 and dataflow model in Listing 3.2.

Table 3.2.: Variable correlation table for the execution trace shown in Table 3.1.

LF1.cmpid	**LF1.evid**	**LF1.sent**	**LF2.cmpid**	**LF2.evid**	**LF2.recv**
Config	1	120394455	Config	1	120394480
Planner	2	120394465	Planner	2	120394476
Effector	3	120394488	Effector	3	120394502

The variable correlation table shown in Table 3.2 enables software system testers to analyze performance properties. For example, Listing 3.3 highlights the expression for evaluating average event round trip time.

```
AVG(LF2.recv - LF1.sent)
```

Listing 3.3: Expression for analyzing round trip time using UNITE.

Based on this expression, UNITE can generate SQL queries that can aggregate performance results captured from the variable correlation table. Likewise, if the aggregation function (*i.e.*, `AVG`) is removed from the expression, then UNITE will present the data trend for the performance property undergoing analysis. Lastly, UNITE provides facilities to group aggregated results—similar to grouping in SQL.

4 ADAPTING EXECUTION TRACES FOR SOFTWARE PERFORMANCE ANALYSIS

In Chapter 1 we presented, the need to relax the assumptions about the structure of the execution trace. As described in Section 2.1 of Chapter 2, existing approaches of using system execution traces to evaluate software performance properties are based on several assumptions about the execution traces. They assume either the source code is available or messages of the execution trace are in a certain format. These assumptions limit the applicability of the existing approaches described in Section 2.1 of Chapter 2.

Although UNITE's analysis technique does not depend on the above mentioned assumptions to some extent, it has not addressed other challenges when analyzing heterogeneous system execution traces, which we are going to describe in this section. These challenges motivated us to come up with a tool and technique called System Execution Trace Adaptation Framework (SETAF). In this chapter we first present the challenges, which motivated SETAF and then we describe the SETAF technique in detail. Finally, we present the experimental results of applying SETAF combined with UNITE to several open source software systems.

4.1 Unresolved Challenges in UNITE

Although UNITE enables analysis of software system performance properties using execution traces, UNITE's methodology has not resolved the following challenges:

- **Challenge 1: Correlating log formats that have non-unique instances.** As mentioned in Chapter 3 the variable correlation table is a set of tuples. Each tuple is a set of values for log format variables. Some of these values represent a time stamp for a particular event (*e.g.*, LF1.sent, LF2.recv). Let us define the subset of a tuple that does not represent time as F. UNITE assumes that the set F of any tuple in the variable correlation table to be unique (*i.e.*, any instance of a particular log format is different from any other instance of the same log format apart from the timestamps). We call this *uniqueness* among the log messages.

As shown in Table 3.1 and Listing 3.1 of Chapter 3, the event ids are different in different log formats. It, however, is possible for the same log message to reoccur without a unique id. When this situation occurs, the relation between the two log messages is considered *non-unique*. Consequently, analysis of an execution trace with non-unique relations typically yields incorrect results.

Table 4.1.: Example execution trace that does not contain unique ids.

Time of Day	**Hostname**	**Message**
2011-02-25 12:00:55	node1	Started doing task A at 12.00
2011-02-25 12:01:55	node1	Finished doing task A at 12.01
2011-02-25 12:02:55	node1	Started doing task A at 12.02
2011-02-25 12:03:55	node1	Finished doing task A at 12.03

For example, Table 4.1 illustrates an example execution trace where the different instances of the same log format are similar. Only the variable parts related to time, change in different instances. It is therefore *hard* to know what start/finish messages are associated with each other without human intervention. Moreover, when an example similar to the one presented in Table 4.1 is analyzed by UNITE, it will yield incorrect results (see Section 4.3.2 for supporting results). It is therefore critical that UNITE be able to handle such situations in generated execution traces.

- **Challenge 2: Correlating log formats with hidden relations.** Execution traces typically capture a variety of events that occur in different software components. When there are repetitive events as shown in Table 3.1, it is easy to identify the relations between log formats. In other cases, there may be no repetitive events in the system. When this occurs, there are no *true* variable parts (other than the log message time) for defining causality between log formats. When this occurs, we say the dataflow model and system execution trace contain *hidden relations*.

Table 4.2.: Example execution trace that contains hidden relations.

Time of Day	**Hostname**	**Message**
2011-02-25 10:00:55	node1	Initializing the system at 10.00
2011-02-25 10:10:55	node1	Start Monitoring components at 10.10
2011-02-25 10:11:55	node1	Finish Monitoring components at 10.11
2011-02-25 10:40:55	node1	Shutting down the system at 10.40

For example, consider the execution trace in Table 4.2. Time is a variable in each log format and each log message is unique, however, there is no explicit variable for determining causality between the log messages. The execution trace in Table 4.2 therefore cannot be analyzed using UNITE, but it is critical that UNITE be able to handle such situations.

- **Challenge 3: Associating values of newly added log format variables.** One of UNITE's main assumptions is that values for a given log format variable are populated using data from its corresponding log messages. Correlating log formats in Table 4.1 and Table 4.2, however, requires adding new log format variables to the dataflow model while preserving the relationship between different log formats. This process is sometimes as simple as adding a monotonically increasing id. Other times it requires coordinating values from other log messages. There is no uniform way to associate data for the newly added log format variables, but UNITE must be able to handle such situations.

The challenges listed above illustrate the heterogeneity among different execution traces in software systems. Although system execution traces vary from system to system, it is possible to use a general-purpose approach for adapting them so that it will help to correctly correlate events in the system. This kind of correct correlation will be useful to support performance analysis of the system. The next section therefore explains our solution called, System Execution Trace Adaptation Framework (SETAF) which is created to addresses the challenges outlined above to enable performance analysis using execution traces.

4.2 Solution Approach : System Execution Trace Adaptation Framework (SETAF)

This section describes the design and functionality of SETAF. This section also uses concrete examples to illustrate concepts realized in SETAF.

4.2.1 Design Approaches for Adapting Execution Traces

Before discussing the details of SETAF, it is necessary to understand different approaches for adapting system execution traces and dataflow models to support performance analysis—in particular with UNITE. The following therefore is a list of approaches for realizing the adaptation:

- **Approach 1: Change execution traces directly.** In some cases, execution traces may not contain certain properties that enable performance analysis. The execution semantics of those systems, however, can be used to define dataflow models as defined in Section 3. If the system's source code is available, then the source code can be changed so that the execution trace reflects the execution semantics. For example, the source code that generates the execution trace can be changed such that each log format has variables for capturing unique relations.

  The advantage of this approach is that UNITE—as is—can directly analyze generated system execution traces. This approach, however, has several disadvantages. First, this approach requires software system testers to have access to the source code so they can make the necessary updates. Moreover, it requires software system testers to be familiar with the source code—its implementation—to make the necessary updates. Secondly, this approach is not practical because updating the source code accordingly can be a costly, error prone, and time consuming task—especially when dealing with a large code base. Finally, the actual source code should not be changed just to analyze performance properties because such changes may impact existing functional and performance properties of the system.

- **Approach 2: Adapting the dataflow model inside UNITE.** It is possible to adapt the dataflow model directly by modifying UNITE's source code. For example, if the adaptation requires adding a new log format variable and a relation then UNITE could be updated to add new variables to an existing dataflow model.

  The advantage in this approach is that the source code of the actual software system does not need to change. Unfortunately, it is not possible to adapt each dataflow model in the same manner. This is because the dataflow model is associated with the given system that generates the execution trace under performance analysis. A dataflow model therefore can only be reused for different executions of the same system. Moreover, this implies that UNITE must be updated to accommodate new dataflow models that need adaptation.

- **Approach 3: Adapting the dataflow model using user-defined external adapters.** This is similar to Approach 2, *i.e.*, adapting the dataflow model inside UNITE, but now the mechanisms for adapting the dataflow model reside in an external specification. The external specification is then loaded by UNITE when analyzing the corresponding execution trace. This approach allows software system testers to write their own adaptation specification according to the system domain without modifying the software system's existing source code.

  The disadvantage of this approach is that software system testers must be aware of the dataflow model's limitations. Software system testers also need to identify the new log format variables, the relations that must be added to the existing dataflow model so it can be adapted correctly.

Based on the advantages and disadvantages of each approach discussed above, Approach 3 was selected as the approach for adapting execution traces for performance analysis. This approach was selected because it addresses the heterogeneity among different systems in different domains. Moreover, it provides greater flexibility and configurability when analyzing execution traces because UNITE's underlying theory and algorithms can remain constant while allowing the adapter(s) to provide more domain-specific details.

When using Approach 3 (above), software system testers first have to identify the adaptation pattern using domain knowledge about the execution trace. This adaptation pattern then need to be expressed in a manner that UNITE can understand. One possible way of doing this is to implement the adapter(s) in the same programming language as UNITE, which is C++. Although this is possible, software system testers will be required to possess some domain knowledge about underlying architecture and functionality of UNITE. We therefore designed a domain-specific language for expressing the adaptation pattern, which does not require software system testers to know the internals of UNITE. Section 4.2.2 describes this domain-specific language in detail.

4.2.2 Defining the Adaptation Specification

As discussed in the previous section, the approach of using external adapters was selected for adapting system execution traces for performance analysis. Software system testers use SETAF by first *manually* analyzing the generated execution trace. Through this analysis, the tester identifies an adaptation pattern. The *adaptation pattern* captures what properties must be added to the dataflow model to support performance analysis via the execution trace. Each adaptation specification contains the following details:

- **Variables.** The variables are private data points that assist with adapting the corresponding system execution trace. The variables are visible only to the adaptation pattern, and not visible to UNITE—thereby helping to address Challenge 3 introduced in Section 4.1. Moreover, these variables keep the state of the current adaptation throughout the execution trace analysis.

- **Initialization.** The initial values for the variables defined above are specified in this section.

- **Reset.** The state variables defined above may need to be reset at the start of processing each log format. This section contains the values for such resetting.

- **Data points.** The data points are new columns added to UNITE's data table for reconstructing valid system execution flows from the generated execution trace. For example, a data point named `LF1.uid` will become a column name in UNITE's data table. Finally, the data points are used to create new relations in UNITE's data table—thereby addressing Challenge 1 introduced in Section 4.1.

- **Relations.** The relations section of the adaptation pattern inserts new causality relations among log formats into the dataflow model. For example, assume the following two data points named `LF1.uid` and `LF2.uid` are added to the dataflow model. This section is used to define that `LF1.uid` causes `LF2.uid`—thereby addressing Challenge 2 introduced in Section 4.1.

- **Adaptation code.** The adaptation code is where the domain-specific logic resides for the adaptation pattern. The adaptation code is segmented based on the log formats that must undergo adaptation. Each segment dictates how to update variables in the dataflow model, as well as its own private variables—thereby helping address Challenge 3 introduced in Section 4.1.

Realization in SETAF. To show the adaptation specification (capturing an adaptation pattern) defined in SETAF, we are going to use a portion of an example execution trace of Apache ANT (`ant.apache.org`), which is presented in Table 4.1. We selected Apache ANT because its adaptation specification is a simple example for illustrating the concepts previously discussed. We, however, have applied SETAF to more complex examples as explained in Section 4.3.

Apache ANT is a widely used build tool primarily for Java projects, but can be used for other purposes (*e.g.*, build automation, documentation generation, and traditional execution shell). Apache ANT completes different tasks during a build process. A task finish event is the effect of a task start event. Using this domain knowledge of the execution trace, Listing 4.1 illustrates the dataflow model for analyzing the execution time of each task in Apache ANT.

```
Log Formats:
  LF1:Started doing task {STRING taskname} at {INT startTime}
  LF2:Finished doing task {STRING taskname} at {INT finishTime}

Relations:
  LF1.taskname = LF2.taskname
```

Listing 4.1: Dataflow model for Apache ANT execution trace.

When repeating the same task, Apache ANT uses the same task name in different log messages, which will result in identical instances of `LF1` and `LF2` (*i.e.*, different only in the time stamp) in the execution trace. When UNITE is processing the execution traces using the dataflow model shown in Listing 4.1, it first identifies all the log message instances of type `LF2`. Then for each message of that type, UNITE tries to find the corresponding `LF1` message instance (*i.e.*, UNITE is trying to correlate the finish event of an ANT Task with the start event of the same task). As shown in the Listing 4.1, the only possible way to do this is using the `taskname`. Because `taskname` is not always different among different message instances, UNITE cannot do this correlation correctly.

This behavior in UNITE is similar to Challenge 1 described in Section 3. Although ANT's execution trace has this problem, a log message representing the start of a task is *always* preceded by a log message representing completion of the corresponding task. The system tester knows this is the execution semantics of ANT, but it is not completely captured in the execution trace. This observation is therefore used to write a SETAF specification that adapts ANT's dataflow model accordingly. Listing 4.2 highlights the adaptation pattern—written as a SETAF specification—to ensure correct analysis of ANT's execution trace.

```
Variables:
  int id_;

Init:
  id_ = 0;

Reset:
  id_ = 0;

DataPoints:
  int LF1.uid;
  int LF2.uid;

Relations:
  LF1.uid -> LF2.uid;

// Begin adaptation code section
On LF1:
  id_ = id_ + 1;
  [uid] = id_;

On LF2:
  id_ = id_ + 1;
  [uid] = id_;
```

Listing 4.2: Adaptation pattern specification for Apache ANT in SETAF.

As illustrated in this listing, first software system testers define variables needed to adapt the system execution trace. This information is captured in the section labeled *Variables* of the SETAF specification. In this case the variable named *id_* maintains the state of the adaptation. As shown in section labeled *Init* and section, labeled *Reset* the value of this variable is initialized to 0 at the start of the adaptation and reset to 0 each time a new log format is processed. Software system testers then use the *DataPoints* section to specify what data points need to be added to each log format. For example, two data points named `LF1.uid` and `LF2.uid`, which are of integer type, are injected into the dataflow model. These two variables are needed to ensure that the relations are unique between the two log formats named `LF1` and `LF2`.

After defining what data points need to be injected into the dataflow model, software system testers define new relations that should be added to the dataflow model. As illustrated in Listing 4.2, the left side of the arrow represents the cause variable; whereas, the right side of the arrow represents the effect variable. This specification of the relations is similar to how existing relations are defined in UNITE.

The final part of the SETAF specification is defining how to adapt the actual execution trace. This task is completed by stating how the adapter transforms the execution trace for each log format that needs adaptation. As shown in Listing 4.2, the `uid` variable is assigned the current value of `id_` in both `LF1` and `LF2`. In both `LF1` and `LF2` the state variable `id_` is incremented. This ensures that the next occurrence of `LF1` is differentiated from the previous occurrence of `LF1`, as well as `LF2`. Finally, the variable inside the brackets [ ] represents log format variables in UNITE. Writing the variable inside brackets is used to differentiate the adapter state variables from UNITE's log format variables.

Integrating SETAF with UNITE. We extended UNITE to provide a configuration option for specifying the location of the adaptation specification, and a standard interface to support the functionality of the adaptation specification described above. The unified interface of UNITE defines three main methods—`update_log_format(), update_relations(), update_values()`. The implementation of these functions are defined in the adaptation specification. When there is an adapter specification provided with a dataflow model, UNITE calls the three methods above as follows.

1. **update_log_format.** This method is called when UNITE is processing log formats in the dataflow model, and SETAF needs to add data points defined in the adaptation specification to the dataflow model.

2. **update_relations.** This method is called when UNITE is processing the log format relations in the dataflow model and SETAF needs to add log format relations defined in the adaptation specification to the dataflow model.

3. **update_values.** This method is called when UNITE needs to populate the variable correlation table. Moreover, this method is only called for the columns (*i.e.*, log format variables) in the variable correlation table that are added from the adaptation specification because the execution trace does not have values to populate the newly added columns.

4.2.3 Compiled versus Interpreted External Adapters in SETAF

There are two possible ways to use this adaptation specification with SETAF: *compiled adapter* and *interpreted adapter*. As shown in Figure 4.1 when using the compiled adapter technique, SETAF generates C++ source code using the adaptation specification. Software system testers then compile the auto-generated code into an external module. During the execution trace analysis, UNITE loads the external module and invokes required functionality for the adaptation from the external module.

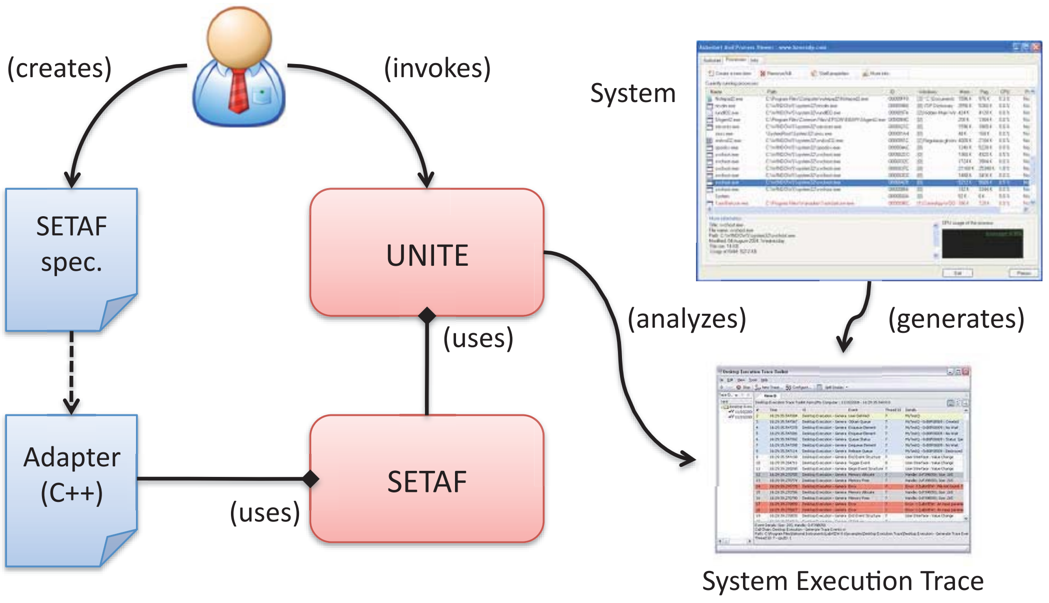

Figure 4.1.: Overall analysis process with SETAF compiled adapter technique.

Listing 4.3 showcases the source code auto-generated for ANT's adapter based on the SETAF specification in Listing 4.2. As shown in this listing, the variables in the *Variables* section of the SETAF specification are mapped into private variables in the adapter. Likewise, the *DataPoints* in the specification are used inside the `update_log_format()` method. More specifically, these data points are used to create new log format variables.

Similarly, the `update_relations()` method uses the relations specified in the *Relations* section of the specification. This method is therefore responsible for creating new relations among log formats with respect to the new log format variables. The `update_values()` method does the actual adaptation. Each adaptation section in the SETAF specification (*i.e.*, `On [name]`) is given its own if statement based on the log format's unique name as defined in the dataflow model.

The identifier `SETAF::int32_vp ()` represents a log format variable casting operator. It is needed because all the variables types in UNITE are derived from a common variable type. This casting operator allows the system tester to narrow the generic variable type to its concrete variable type, such as an integer, to set its value accordingly. SETAF has log format variable casting operators for each variable type supported in UNITE. Lastly, UNITE uses SETAF to adapt its dataflow model to support the analysis of the execution trace using the compiled version of the specified adapter source code.

```
class Ant_Adapter : public CUTS_Log_Format_Adapter {
public:
  void init (void) { this->id_ = 0; }
  void reset (void) { this->id_ = 0; }
  void close (void) { delete this; }

  void update_log_format(CUTS_Log_Format * lfmt) {
    const string & name = lfmt->name ();

    if (name == "LF1")
      lfmt->add_variable ("uid", "int");
    else if (name == "LF2")
      lfmt->add_variable ("uid", "int");
  }

  void update_relations(CUTS_Log_Format * lfmt) {
    const string & name = lfmt->name ();

    if (name == "LF1")
      lfmt->add_relation ("LF2", "uid", "uid");
  }

  void update_values(Variable_Table & vars,
        CUTS_Log_Format * lfmt) {
    const ACE_CString & name = lfmt->name ();

    if (name == "LF1") {
      ++this->id_;
      SETAF::int32_vp (vars["uid"])->value (this->id_);
    }
    else if (name == "LF2") {
      ++this->id_;
      SETAF::int32_vp (vars["uid"])->value (this->id_);
    }
  }
private:
  int id_;
};
```

Listing 4.3: Auto-generated source code for Apache ANT adapter.

The interpreted adapter technique removes the extra overhead of code generation and compilation. As shown in the Figure 4.2, software system testers just need to express the adaptation pattern as a specification. During analysis time of the execution trace, UNITE loads the adaptation specification and SETAF builds an in memory object model of the adapter.

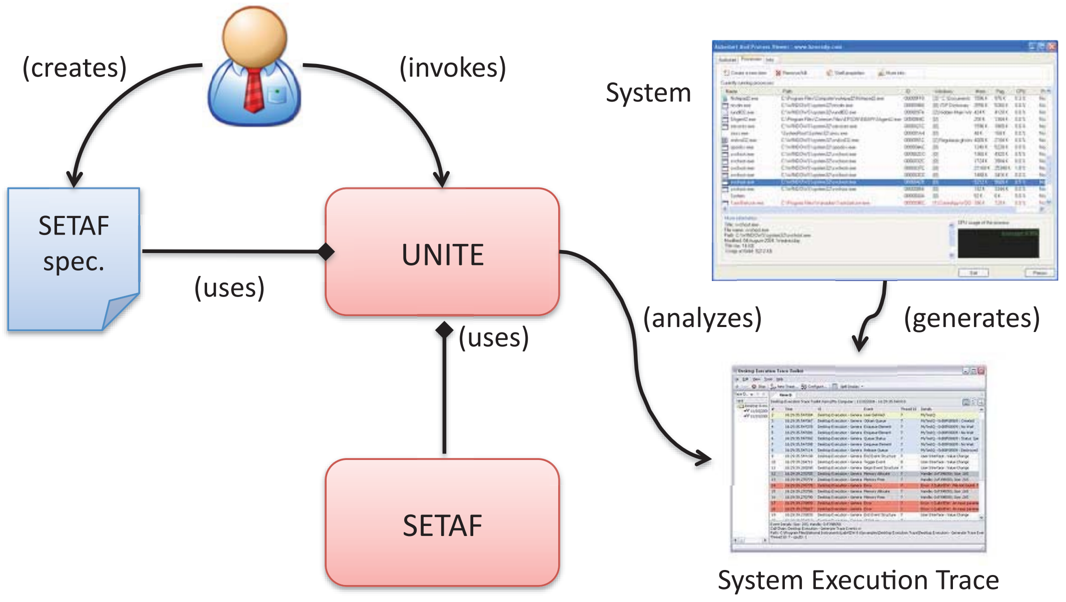

Figure 4.2.: Overall analysis process with SETAF interpreted adapter technique.

For the interpreted adapter technique, the adaptation specification is mapped into a `SETAF_Interpreter` object type. The data points and relations are stored in two list objects where each entry in the list corresponds to a data point and relation, respectively. The methods `update_log_format()` , `update_relations()` and `update_values()` are methods on the `SETAF_Interpreter` object type. The first two methods process the data point and relation list objects, respectively, and update the dataflow model accordingly.

The interpreted adapter technique also defines two classes: `SETAF_Variable` and `SETAF_Command`. `SETAF_Variable` represents different kinds of variables in the adaptation specification, and `SETAF_Command` represents each statement in the adaptation code section of the adaptation specification. Figure 4.3 illustrates the `SETAF_Variable` and `SETAF_Command` classes, their subclasses, and the relationship between the two class hierarchies.

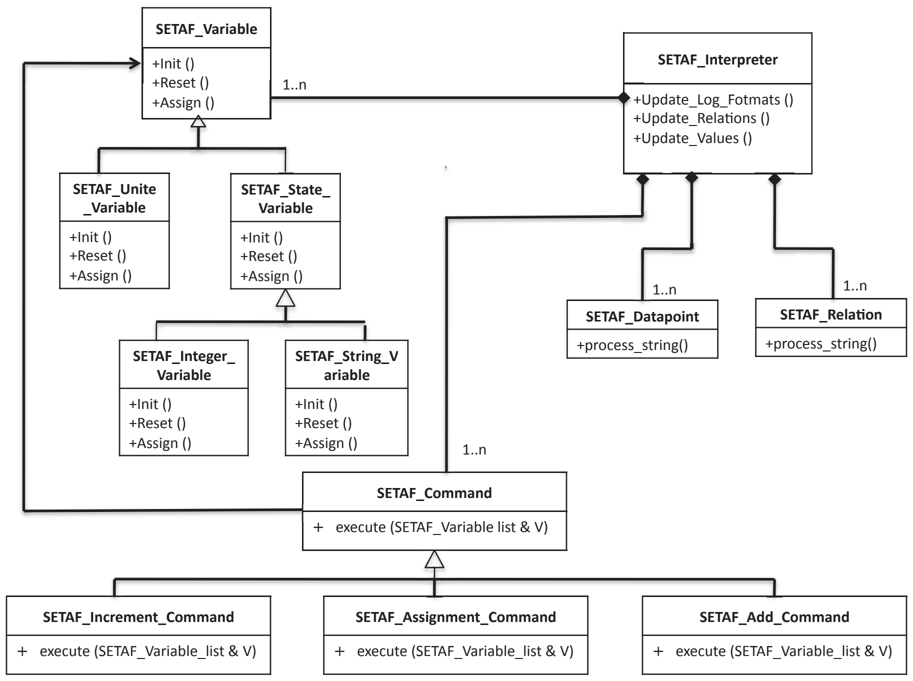

Figure 4.3.: SETAF_Variable and SETAF_Command class hierarchies.

As shown in the figure, the variables defined in the *Variable* section of the adaptation specification are represented using `SETAF_State_Variable` object type. Likewise, UNITE's log format variables (*i.e.*, variable inside "[ ]") are represented using `SETAF_Unite_Variable` object type. Figure 4.3 also shows that SETAF's interpreter adapter technique currently implements three types of statements: assignment, increment, and addition. The three statements, which are an implementation of the Command pattern [56], are represented using `SETAF_Assignment_Command`, `SETAF_Increment_Command`, and `SETAF_Addition_Command`, respectively. SETAF only implements these three statements because they are sufficient to adapt dataflow models of execution traces we have currently used with SETAF. Further these three statements are the building blocks for most of the complex adaptation patterns. Moreover if the adaptation requires a new statement, then it can be easily implemented using the Command pattern.

The actual functionality of the command is implemented in a method named `execute()`. As described before, the adaptation code is segmented based on the log formats. Each log format that appears in the adaptation code is therefore mapped to a `SETAF_Command` object. When UNITE needs SETAF to apply adaptation to the execution trace, it invokes `update_values()` for the corresponding log format. The SETAF interpreted adapter then locates the corresponding `SETAF_Command` object and executes it.

The interpreted adapter technique is easier to use, because the overhead associated with compiling the adapter, such as obtaining UNITE libraries and setting up a development environment, are not present in the interpreted adapter. One therefore may question the use of the compiled interpreter technique. We developed the compiled adapter technique because our initial intuition was that the compiled technique would have better performance. This is because UNITE calls the functionality for adaptation from a compiled module compared to parsing the specification file and building an object model in the interpreted adapter. Section 4.3.6 shows a performance comparison of the two techniques to evaluate whether our intuition is correct.

4.3 Results of Applying SETAF to Open-Source Projects

This section presents results from applying SETAF to several open-source projects that generate execution traces and do not have the properties required for performance analysis via UNITE. It also shows a performance comparison of the two techniques—compiled versus interpreted adapter. Finally, the adaptation specification used for either technique (*i.e.*, compiles vs. interpreted) is the same. The performance analysis results from applying UNITE and SETAF to different open source software systems is therefore the same for both the cases.

4.3.1 Experimental Setup

To determine applicability of SETAF's technique, we applied SETAF to the following open-source projects:

- **Apache ANT.** Apache ANT, which was previously introduced in Section 4.2.2, is a widely used Java library and a command line tool mainly used to build Java-based software systems.

- **Apache Tomcat Web Server.** Apache Tomcat (`tomcat.apache.org`) is an implementation of the Java Servlet [57] and JavaServer Pages [58] technology. It is also one of the most widely used Java web-based application servers. Finally, Tomcat is embedded in many enterprise application servers that serve very high volumes of requests.

- **ActiveMQ Java Messaging Server (JMS) Broker**. Apache ActiveMQ (`activemq.apache.org`) is a widely used message broker that implements Java Messaging Services (JMS) [59]. Apache ActiveMQ supports implementation of enterprise integration patterns such as publisher-subscriber. It is also integrated into a variety of Enterprise Service Bus (ESB) [60] middlewares in order to support message mediation. It is also designed for high performance clustering, client-server and peer based communication.

- **Deployment And Configuration Engine (DAnCE)**. DAnCE [61] is an implementation of the Object Management Group (`www.omg.org`) Deployment & Configuration (D&C) [62] specification for deploying and configuring component-based distributed systems.

These open-source projects were selected for several reasons. First, we analyzed their execution trace with only UNITE (*i.e.*, without SETAF) and produced invalid results because the execution trace lacked the required properties to support performance analysis via UNITE. Secondly, each open-source project exhibited a different adaptation pattern, which is discussed in their respective result section. Finally, each software application used a logging facility, such as log4j (`http:`

`//logging.apache.org/index.html`) and ACE Logging Facilities [63]. It was therefore possible to use appenders and intercepters, respectively, to capture generated execution traces and store them in a database for adaptation and analysis without making any modifications to the existing source code.

All experiments were conducted on an Intel core 2 Duo 2.1 GHz processor, with 3GB memory and running 32-bit Windows 7 operating system. The execution of either UNITE or SETAF, however, is not bound to a particular operating system as long as the operating system supports the Adaptive Communication Environment (ACE) [64], Boost (`www.boost.org`), and SQLite (`www.sqlite.org`) middelware.

4.3.2 Experimental Results for Applying SETAF to Apache ANT

Table 4.3 shows the correlation table constructed by UNITE when analyzing ANT's execution trace without SETAF. The end goal was to measure the average execution time of each ANT task, which is accomplished by subtracting the `finishTime` from the `startTime`. Unfortunately, Table 4.3 constructed by UNITE will produce incorrect results because some rows are not correlated correctly. For example, the first and third rows have a `startTime` that is greater than the `finishTime`. This means that the task finished before it actually started, which is not the case.

Table 4.3.: Data table reconstructed by UNITE for a subset of ANT's tasks without adaptation pattern specification.

startTime (msec)	**LF1.task**	**finishTime (msec)**	**LF2.task**
1500	property	860	property
1500	property	1704	property
1516	available	1511	available
1516	available	1518	available

The reason for this error lies not in the generated execution trace. Instead, the error lies in the analysis because the relations in the dataflow model used to reconstruct the dataset from the execution trace are not unique (see Listing 4.1). More specifically, UNITE processes the log formats in topological order of the corresponding directed acyclic graph of the data flow model. UNITE therefore first populates the `finishTime` column. Then it uses a *SQL UPDATE* query to update the corresponding data value of the `startTime` column.

In this case, UNITE can only do the correlation using the relation *LF1.task = LF2.task*. This cause UNITE to update multiple rows (as the relation is not unique) and finally ends up with the latest value of the *startTime* for a particular *task*. For example for *task property*, UNITE updates the `startTime` with 1500 by replacing all the previously updated values. Because of the non-unique relations in the dataflow model, the final analysis using only UNITE results in several negative values for the average execution time of different ANT tasks as illustrated in Table 4.4.

Table 4.4.: Results for analyzing reconstructed table in UNITE for ANT without adaptation specification.

Task	**Average Execution Time (msec)**
available	-630.34
delete	0.0
macrodef	140.0
mkdir	-25.125
path	297.0
patternset	-9.77
property	-241.4
Total evaluation time (sec)	0.11994

To correct the errors in UNITE's current analysis, we defined a SETAF specification as described in Listing 4.2 for adapting ANT's generated system execution trace and the corresponding dataflow model. Table 4.5 therefore highlights the dataset reconstructed by UNITE after using SETAF to apply the adaptation pattern to the reconstruction process. As shown in this table, `startTime` and `finishTime` are now correlated correctly because of the unique id added by SETAF. In this table, `startTime` is always less than `finishTime`, which is the expected result.

Table 4.5.: Improved table reconstruction using SETAF and UNITE for a subset of ANT's tasks.

LF1.uid	**LF1.task**	**startTime**	**LF2.uid**	**LF2.task**	**finishTime**
1	property	766	1	property	860
2	property	1500	2	property	1704
3	available	1500	3	available	1511
4	available	1516	4	available	1518

Finally, Table 4.6 illustrates the updated final results for analyzing task execution time after using SETAF to adapt the execution trace as UNITE analyzed it. As shown in this table all the service times for different ANT tasks have positive values, which produce the expected (and correct) analysis results.

Table 4.6.: Final results for adapting UNITE's analysis using SETAF for a subset of ANT's tasks.

Task	**Average Execution Time (msec)**
available	93.67
delete	55.0
macrodef	79.0
mkdir	2.0
path	390.0
patternset	6.0
property	17.975
Total evaluation time of compiler technique (sec)	0.210
Total evaluation time of interpreter technique (sec)	0.220

4.3.3 Experimental Result for Applying SETAF to Apache Tomcat

To further validate SETAF's method for adapting system execution traces for analysis via UNITE, we applied SETAF and UNITE on Apache Tomcat. To obtain a considerable amount of log messages for performance analysis, we had to set the log level to a high value (*i.e.*, DEBUG) to produce a more verbose execution trace. This has some impact on the system performance, but the purpose of this experiment is to validate SETAF's applicability to a variety of applications—not to validate performance.

When the Tomcat server starts up, it outputs the total time of the startup process. Our aim was to compare this value with the value calculated from analyzing its generated system execution traces using UNITE. The log messages related to this case study, however, do not contain any variable parts other than the timestamp of the event being captured in the execution trace. This means that SETAF is needed to adapt the execution trace.

After analyzing the execution trace, we identified twelve independent events (or log formats) associated with Tomcat's startup process. Although there were no variable parts in the log formats for explicitly identifying causality in the dataflow, the desired causality can be defined by injecting a common id.

```
Variables:
 string server_name;

Init:
 server_name = "Tomcat";

Reset:
 server_name = "Tomcat";

DataPoints:
 string LF1.cid;
 string LF2.cid;
 // ...
 string LF12.cid;

Relations:
  LF1.cid->LF2.cid;
  LF2.cid->LF3.cid;
  // ...
  LF11.cid->LF12.cid;

// Begin adaptation code section
On LF1:
  [cid] = server_name;

```

```
On LF2:
  [cid] = server_name;
// ...

On LF12:
  [cid] = server_name;
```

Listing 4.4: SETAF specification for Apache Tomcat.

Listing 4.4 therefore highlights a portion of the SETAF specification for Tomcat. As illustrated in this listing, a variable called *cid* is added to all the log formats to expose the hidden relation.

Table 4.7.: Results for adapting Tomcat execution trace using UNITE and SETAF.

Method	**Time (msec)**
Server startup time from Tomcat instrumentation	68799.0
Server startup time from UNITE w/o SETAF	N/A
Server startup time from UNITE w/ SETAF	68802.0
Total evaluation time of compiler technique (sec)	8.297
Total evaluation time of interpreter technique (sec)	8.329

Table 4.7 shows the results for comparing the server startup time calculated by UNITE with and without SETAF against the server startup time given by the server itself. As shown in this table, it was not possible to analyze the system execution trace using UNITE alone because there are no variable parts for defining causality between log formats (*i.e.*, Challenge 2 in Section 3). When we analyzed the same system trace using UNITE and a SETAF adaptation specification, the resulting analysis is relatively close (*i.e.*, a 0.00436% difference). The reason for the difference in time is because the instrumentation points in the Tomcat source code are not the same as the two points where the log messages are generated. More importantly, however, this experiment shows that SETAF and UNITE can be used to produce results similar to direct instrumentation. This, however, is dependent on how far a generated log message is from the real instrumentation points-of-interest.

4.3.4 Experimental Result for Applying SETAF to Apache ActiveMQ

In Apache ActiveMQ, each message broker uses a local file for persistent storage. This persistent store is updated periodically in order to prevent message lost during a system crash. This process is called *checkpointing*. When checkpointing, ActiveMQ generates a message with the content “checkpoint started”. At the end of the checkpointing task, ActiveMQ generates another message with the content “checkpoint done”. Because ActiveMQ checkpoints periodically, the checkpointing messages occur frequently in the generated system execution trace.

Unfortunately, when we first tried to evaluate ActiveMQ’s average checkpointing time using UNITE for one scenario, we learned that average checkpointing time was -27235.333 msec. This result was clearly not correct because checkpointing time cannot be a negative number. We then realized that ActiveMQ’s execution trace cannot be analyzed as is using UNITE because ActiveMQ’s dataflow model does not contain unique relations.

ActiveMQ’s execution trace, however, is similar to Apache ANT’s execution trace. This is because each log message that represents the start of checkpointing is preceded by a finish checkpointing message before another start checkpointing message occurs. Because of this fact, the same adaptation specification is used as in Listing 4.2 to adapt the execution trace generated by ActiveMQ.

After executing UNITE with the SETAF adapter for ActiveMQ, we were able to evaluate that average checkpointing time was 115.917 msec for the scenario discussed above. Software system testers therefore can use UNITE and SETAF to determine whether there are any performance problems with the checkpointing module of ActiveMQ without making any modifications to the existing source code to perform such analysis. This will be very useful when ActiveMQ is running in thirdparty systems, because in those types of system only the server log is available for analysis.

4.3.5 Experimental Result for Applying SETAF to DAnCE

The goal of analyzing DAnCE is to evaluate the amount of time it takes to deploy a set of components on a given node in the generated deployment plan. For this case study we used the BasicSP scenario provided with DAnCE. The BasicSP scenario has four different components mapped into four different nodes. After manually analyzing DAnCE's system execution trace for the BasicSP scenario, the following dataflow model was constructed for DAnCE.

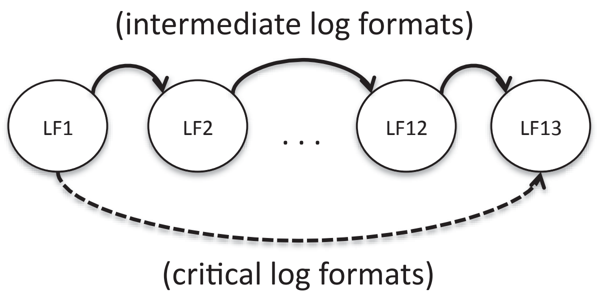

Figure 4.4.: Log formats associated with DAnCE.

As shown in Figure 4.4, the dataflow model contains 13 different log formats (out of 50+ log formats) that depend on each other. The DAnCE deployment process can be divided into three phases [61]: (1) deployment preparation phase; (2) start launching phase; and (3) finish launching phase. Each of these phases are driven by remote method calls between different components in DAnCE. The first 5 log formats represent the analysis of the preparation phase. The next 4 log formats represent the start launching phase. The last 4 log formats represent the finish launching phase. If the software testers want to isolate different phases for performance testing, the tester can use log formats within each phase for that purpose.

Because of different execution flows in DAnCE and its distributed functionality, it is not possible to use only the first and last log formats for the analysis. Instead, each intermediate log format between the first and last log format must be considered to ensure correct correlation. Unfortunately, the relation between the intermediate log formats is not unique.

Because component deployment is done according to a deployment plan, it is possible to use a common id named *planid* to correlate the messages among different components and deployment plans. Moreover, another id named *nodeid* can be used to correlate messages that are generated from the same node. Listing 4.5 therefore presents the SETAF specification for adapting DAnCE's generated system execution traces for analysis using UNITE.

```
Variables:
  string planid_;
  int lf12_count_, lf13_count_, nodeid_;

Init:
  lf12_count_ = lf13_count_ = nodeid_ = 0;

Reset:
  lf12_count_ = lf13_count_ = nodeid_ = 0;

DataPoints:
  string LF1.planid; string LF2.planid;
  string LF5.planid; string LF6.planid;
  string LF9.planid; string LF11.planid;
  int LF12.nodeid; int LF13.nodeid;

Relations:
  LF1.planid->LF2.planid; LF5.planid->LF6.planid;
  LF6.planid->LF7.planid; LF8.planid->LF9.planid;
  LF9.planid->LF10.planid; LF10.planid->LF11.planid;
  LF11.planid->LF12.planid; LF12.nodeid->LF13.nodeid;

// Begin adaptation code section
```

```
On LF1:
  [planid] = planid_;
On LF2:
  [planid] = planid_;
On LF5:
  [planid] = planid_;
On LF6:
  [planid] = planid_;
On LF9:
  [planid] = planid_;
On LF11:
  [planid] = planid_;
On LF12:
  [nodeid] = lf12_count_;
  lf12_count_ = lf12_count_ + 1;
  plan_id_ = [planid];
On LF13:
  [nodeid] = lf13_count_;
  lf13_count_ = lf13_count_ + 1;
```

Listing 4.5: SETAF specification for DAnCE.

As illustrated in Listing 4.5, all the log formats in the SETAF specification for DAnCE, other than `LF12` and `LF13`, use the private variable `planid` to get an adaptation value. The value of this private variable is set by `LF12` because it is the first log format SETAF processes. This is because UNITE processes the log messages in the topological order based on the dataflow model to achieve $O(mn)$ runtime complexity where m is the number of log formats in the dataflow model and n is the number of log messages in the execution trace.

In order to correlate `LF12` and `LF13`, a newly added id named `nodeid` is used. The instance counts of this log format are kept in state variables `lf12_count_` and `lf13_count_`. These variables are used to populate the value of `LF12.nodeid` and

`LF13.nodeid`. This allows us to differentiate the similar instances of the same log format. The scenario we tested with DAnCE has nodes named `EC`, `BMDevice`, `BMClosedED` and `BMDisplay` where each contains a set of component instances.

Table 4.8.: Results for adapting DAnCE's execution trace using SETAF to measure deployment time (DT).

Node	**DT w/o SETAF**	**DT w/ SETAF (sec)**
EC	N/A	5.0
BMDevice	N/A	5.0
BMClosedED	N/A	5.0
BMDisplay	N/A	5.1
Total evaluation time of compiler technique (sec)	N/A	0.237
Total evaluation time of interpreter technique (sec)	N/A	0.272

Table 4.8 illustrates the results of analyzing deployment time for each node after adapting DAnCE's generated system execution trace. As shown in this table, we were not able to analyze DAnCE's system execution trace using only UNITE because some of the log formats were lacking variable parts to define causalities (*i.e.*, Challenge 2 in Section 4.1) and some newly added log format variables required analyzing other log messages to populate their corresponding value (*i.e.*, Challenge 3 in Section 4.1). When we used both UNITE and SETAF to analyze DAnCE's system execution trace, we learned that all four nodes take approximately equal time to deploy. More importantly, however, these results show that with careful analysis of the generated system execution trace, SETAF and UNITE can be used to analyze such complex dataflow models as found in DAnCE without modifying the existing source code.

4.3.6 Performance Comparison of SETAF Compiled and Interpreted Techniques

We compared the load time, processing time, total evaluation time and percentage difference in total evaluation time for the compiled and interpreted adapters in SETAF. This performance analysis is based on the following equation:

$$\text{Total Evaluation Time} = \text{Load Time} + \text{Process Time}$$

In the compiled adapter, the load time is the amount of time taken to load the compiled adapter module. In the interpreted adapter, the load time is the amount of time taken to parse the adapter specification and create the object model that represents an adapter. In both cases, the process time is the amount of time taken to evaluate the dataflow model using either adapter.

Figure 4.5 shows the load times for both the techniques. As shown in the figure, the compiled adapter has a much lower load time when compared to the interpreted adapter. This is expected because the interpreted adapter has to parse the specification and build an object model at run-time. In the compiled adapter, this object model is already in binary form and UNITE only has to load the compiled adapter into memory.

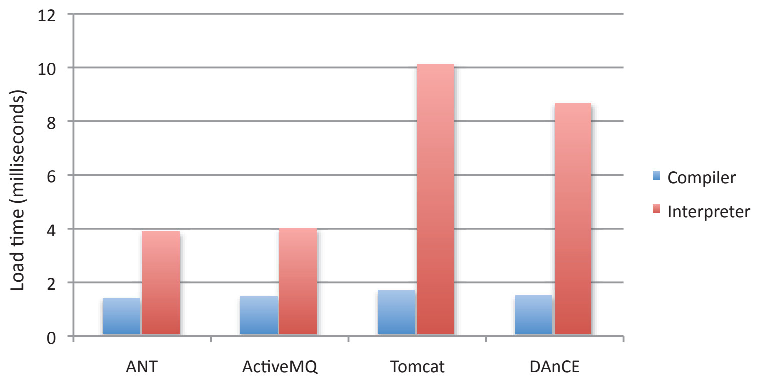

Figure 4.5.: Load times of SETAF interpreted and compiled adapter.

Although the load times are significantly different between the compiled and interpreted adapters, Figure 4.6 shows that there is not much difference in total evaluation times. This is because total evaluation time for both techniques is dominated by the processing time (*e.g.*, more than 96% as shown in Table 4.9). The reason for the dominance in processing time is the number of database operations executed during the analysis stage of UNITE.

For example, for each log format defined in the dataflow model UNITE needs to find all the corresponding log message instances [9]. This needs to be done because

UNITE needs to extract values for the data points in each log format . Therefore it needs to iterate through the system execution trace each time it finds a new log format. This complexity is the same for both the compiled and interpreted adapter, and reflected in Figure 4.6.

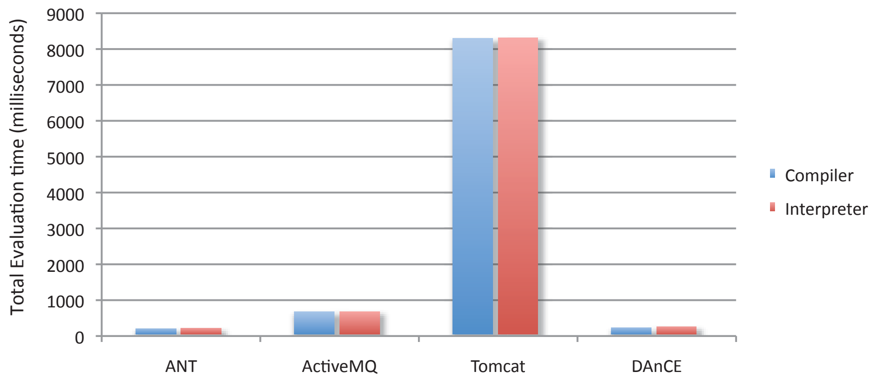

Figure 4.6.: Total evaluation time of SETAF interpreted and compiled adapter.

Table 4.9.: Process time as a percentage of total evaluation time.

Open-source Project	**Size of the trace (KB)**	**Compiled Adapter**	**Interpreted Adapter**
ANT	4032	99.33%	98.23%
ActiveMQ	2242	99.79%	99.41%
Tomcat	27880	99.97%	99.87%
DAnCE	576	99.36%	96.80%

Final discussion. The four case studies previously discussed three different adaptation patterns. The ANT and ActiveMQ case study are similar in that a unique id is added to the two log formats. In the DAnCE case study, the new log format variables were populated using the values extracted from previously found log format variable values after analyzing the execution trace. In the Tomcat case study, a common id is added to log formats. This therefore showcases the flexibility and extendibility of SETAF's approach to support adaptation of execution traces for performance analysis.

The performance comparison results of the two adaptation techniques show that there is little difference between a compiled and interpreted adapter. We therefore recommend that software system testers use interpreted adapters because as mentioned in Section 4.2.2 it is easier to use. Moreover, the interpreted adapter provides more flexibility and usability without sacrificing the performance.

4.3.7 Threats to Validity

One major assumption in these experiments is the collected log messages correctly show the time of occurrence of events. This means system developers must insert the log statement temporally adjacent to the program statement that is responsible for the particular event. If the log statement is temporally far away from the event statement, then the analysis can give incorrect results. This form of "good practice" is not difficult to adhere when developing software. It is therefore not a big threat for using SETAF and UNITE for performance analysis.

Another important aspect of these experiments is that they are tightly coupled to the generated execution trace. For example, the same system can be executed in different modes, such as with different configuration settings and different operating conditions depending on user requirements. In these situations, the generated execution trace may differ from mode to mode, and therefore performance analysis results may also differ. Different executions under the same mode, however, will produce similar results, because executions under the same mode always generates the same execution trace but with different timestamps.

4.4 Chapter Summary

In this chapter we have described the System Execution Trace Adaptation Framework (SETAF) which enables software testers do performance analysis with minimal assumptions about the system execution trace. We first described the limitations of UNITE, which motivated us to create SETAF. Then we described several design approaches to address those limitations and the overall design, functionality of SETAF. When doing performance analysis using SETAF and UNITE software testers do not need to have access to the source code of the system. The described technique does not assume any structure in the log messages or certain properties such as unique identifiers and relations in the system execution trace. We showcase the validity of our approach by applying it to four open source software systems. Following are the key research contribution from this research work.

- A framework for adapting dataflow models associated with the execution trace, without changing the source code of the software system, the execution trace or the performance analysis tool (*i.e.* UNITE)

- A domain specific language for software testers to write different adaptation modules, where each adaptation module captures an adaptation pattern.

5 AUTO-CONSTRUCTING DATAFLOW MODELS FROM EXECUTION TRACES

Although UNITE and SETAF can be used to analyze system execution traces using the concept of dataflow models, defining the dataflow model is done manually. In Chapter 1, we briefly described requirements and advantages of auto-constructing the dataflow model with minimum user intervention. In Section 2.2, we have already discussed the related research work on constructing intermediate models from system execution traces. This chapter first describes challenges associated with auto-constructing dataflow models from execution traces. Then, we describe Dataflow Model Auto Constructor (DMAC), which is a tool and technique for auto-constructing dataflow models [65] in detail. Finally, we describe results of applying DMAC to different open source software systems.

5.1 Challenges Addressed by Proposed Approach

For trivial system execution traces, it is not hard to manually define a dataflow model. This, however, is not the case for large and complex system execution traces. In this situation, it is ideal to automatically construct a dataflow model from its corresponding system execution trace. Based on this need, we have identified the following challenges for automatically constructing a dataflow model:

- **Challenge 1: Correctly identifying valid log formats.** As mentioned in Chapter 3 Log formats contain both static and variable parts. The variable parts are used to define causal relations between log formats and define expressions that evaluate a performance property. Failure to identify the correct static and variable parts can reduce the number of possible variables available for usage in an expression. Moreover, it can inhibit our ability to define correct and complete relations in the dataflow model. It is therefore important to correctly identify complete log formats to create a comprehensive dataflow model. As described in Section 2.2 the current approaches for abstracting event types from logs either (1) Depends on empirical rules [38, 40], (2) Applies frequency based approaches, however does not provide solutions to infrequent situations [39, 40] or (3) Require source code [14].

- **Challenge 2: Correctly identifying causal relations between log formats.** When all the events are in a single execution context (*i.e.*, no concurrency), we can assume causality is defined by the order of occurrences for the events. This is because the representative log messages generated for these events occur from a source code that is executed serially. In this case we can generate the dataflow model based on their execution order. The difficult task, however, is identifying cause-effect relationships between events that occur in different execution contexts because it introduces multiple local clocks, which may not be synchronized. Even if we assume a single clock (*i.e.*, a global clock), there may, or may not, be explicit relationships between events that occur in different execution contexts. As described in Section 2.2 the current approaches for detecting causality relationships between abstract event types are supervised learning techniques [41], which have a higher risk of model overfitting.

- **Challenge 3: Correctly identifying relationships between variables.** After identifying the log formats and relations of the dataflow model, one must identify relations between the variables such that their values are equal for all the instances. Similar to the previous challenges, it is important to identify correct relations between variables to ensure correct correlation of data points.

The next section describes Dataflow Model Auto-Constructor (DMAC), which has been created to address the above mentioned challenges.

5.2 Solution Approach : Dataflow Model Auto-Constructor (DMAC)

Figure 5.1 illustrates DMAC's workflow for constructing a dataflow model from a system execution trace. As shown in this figure, the process consists of two major steps: (1) identifying log formats for the dataflow model of the corresponding system execution trace; and (2) identifying causal relationships between different log formats.

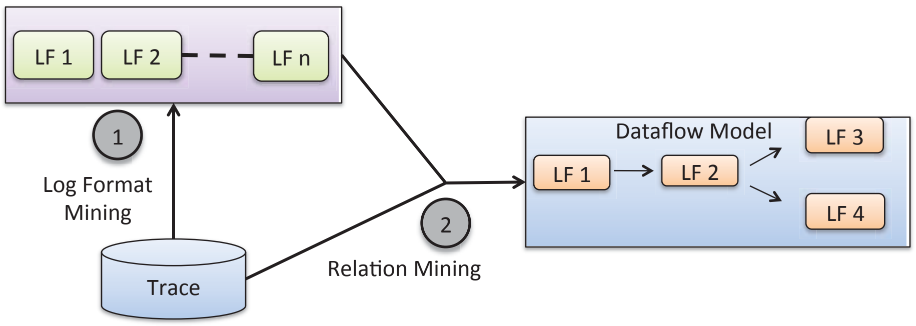

Figure 5.1.: DMAC's workflow for constructing a dataflow model.

The log format mining step (*i.e.* step 1) is based on frequent-sequence mining. This section therefore gives a brief overview of frequent-sequence mining and then describes the functionality of DMAC's log format mining algorithm.

5.2.1 Overview of Frequent-Sequence Mining

Given a collection of sequences S and a support threshold $\sigma \in (0,1]$, frequent-sequence mining locates all the sequences that are sub-sequences of at least $\sigma \times |S|$ sequences. The support of a sequence t is defined as the number of sequences in S such that t is a sub-sequence. Mathematically,

$$\sum_{\delta \in S} I(t \sqsubseteq \delta) \tag{5.1}$$

where $\sqsubseteq$ denotes a sub-sequence and I is an indicator random variable. Lastly, the support threshold σ is called *minimum support* and denoted by *min-sup*.

In the context of system execution traces, we define sequences as words separated by a delimiter (*e.g.*, space, tab, and newline). There are several methods for frequent-sequence mining, *e.g.*, *Generalized Sequential Pattern (GSP)* [66] and *Sequential PAttern Discovery using Equivalence classes (SPADE)* [67]. DMAC uses *SPADE* because of its efficiency [67].

DMAC considers all *k-word* sequences where $k > 1$, because DMAC performs multiple levels of mining (discussed later) to select candidate sequences and remove sub-sequences from larger ones. Moreover, most log formats have at least two words as their static part. Considering two or more word sequences therefore is sufficient because most single word sequences appear in two or more word sequences as sub-sequences. Likewise, single word sequences that are not part of any higher-order sequence should correspond to one word log formats.

Table 5.1.: An example system execution trace to illustrate the DMAC concepts.

ID	**Time**	**Hostname**	**Severity**	**Thread**	**Message**
1	13:15:56	Host-A	INFO	1	A sent message 1 at 10
2	13:16:10	Host-A	INFO	1	A received message 1 at 20
3	13:16:20	Host-A	INFO	1	Got the authentication for request at 30
4	13:16:30	Host-B	INFO	2	B sent message 2 at 40
5	13:16:40	Host-B	INFO	2	B received message 2 at 50
6	13:16:55	Host-B	INFO	2	Access denied at 60
7	13:17:10	Host-C	INFO	3	C sent message 3 at 70
8	13:17:21	Host-C	INFO	3	C received message 3 at 80
9	13:18:35	Host-C	INFO	3	Got the authentication for request at 90

To better understand how frequent-sequence mining relates to DMAC, applying frequent-sequence mining with *min-sup* value of 0.33 to the log messages in Table 5.1 produces the set of sequences shown in Listing 5.1. Reducing the *min-sup* to 0.2 will generate all the combinations with `A,B,C,1,2,3` in addition to the set of frequent-sequences shown in Listing 5.1.

```
sent message, sent at, message at, received message,
received at, sent message at, received message at
```

Listing 5.1: Frequent-sequences for *min-sup* value of 0.33.

DMAC then uses the resulting frequent-sequences to identify log formats of a system execution trace. When an event occurs multiple times during the system execution, its corresponding log message will appear in the system execution trace with variations in its variable parts; whereas, its static parts do not change. This implies that the static parts have a relatively high frequency compared to its variable parts. DMAC therefore assumes frequent-sequences correspond to words in static parts of a log format.

Because of DMAC's assumption, it is important to select a relevant *min-sup* value. Finding the appropriate *min-sup* value is a challenging problem in the field of data mining [68]. If the min-sup is set to a very low value, variable parts of a log format will be filtered as static parts (as discussed in the example above). Likewise, if the *min-sup* is set to high, then SPADE will generate a smaller set of frequent-sequences. This can cause DMAC to miss log formats. DMAC therefore uses an iterative process that enables it to start with a relatively high *min-sup* value and still identify a large number of log formats present in the system execution trace.

5.2.2 Identifying Log Formats.

Identifying the set of log formats in a dataflow model is a challenging process. This is because it requires differentiating static parts from variable parts, and identifying their correct positions within the log format. For a given system execution trace, finding the set of log formats is a two step process. In the first step, DMAC uses a user-provided *min-sup* value. This step is similar to the process described in Section 5.2.1.

The next step is constructing the log formats from the identified frequent-sequences. Based on the assumption described in Section 5.2.1 where frequent-sequences are candidate static parts of the log formats, DMAC uses the frequent-sequences to identify the variable parts of a log format. More specifically, DMAC uses the following sets to assist with constructing valid log formats from identified frequent-sequences:

- **Frequent-sequences (F).** The set F is the frequent-sequences generated by SPADE for a given *min-sup* value. For example, Listing 5.1 is the set F for Table 5.1 with *min-sup* value of 0.33.
- **Maximal-sequences (M).** The set M is the maximal-sequences for F. A set of frequent-sequences is called *maximal* if it is not a subset of any other set of frequent-sequences. For example, Listing 5.2 shows the set M for Listing 5.1. Maximal sequences are important because a maximal sequence contains all the static parts for its corresponding log format.

```
sent message at
received message at
```

Listing 5.2: Maximal-sequence set for Listing 5.1.

- **Position vector (p).** The position vector p is an integer vector that tracks the position numbers of words in a maximal sequence (*i.e.*, i^{th} word of the log message) that appears in an actual log message. A position vector is always associated with a maximal sequence $m \in M$. Each value in p represents the position of the corresponding word of its associated maximal sequence m when m appears in a log message. There can also be different position numbers for m in different log messages. Listing 5.3 shows the position vector for M in Listing 5.2. We denote the set of all position vectors associated with a maximal sequence m as p_m.

```
   sent message at - (2, 3, 5)
   received message at - (2, 3, 5)
```

Listing 5.3: Position vectors for the maximal sequences in Listing 5.2

DMAC first uses SPADE to construct the set F. DMAC then executes the following steps to construct log formats for a system execution trace:

1. For the set F, DMAC first generates the set M.

2. $\forall m \in M$, DMAC calculates the set of position vectors p_m. DMAC does this by checking if m is a sub-sequence of any log message in the system execution trace. If it is a sub-sequence of a log message and $p \notin p_m$, then a new position vector p is created and added to the set p_m. When a new p is added to p_m, the size of the log message is also recorded. This is used to construct the log formats as explained next.

3. $\forall m \in M$, log formats are created for each member of p_m. As shown in Figure 5.2, a dummy log format is first created for the message where each word in the dummy log format is initialized with an empty placeholder ($\{\}$). The number of words in the dummy log format is equal to the number of words recorded in previous step. Likewise, the position vector contains the positions of the static parts and the actual words for the static parts are contained in m. Placeholders that have not been replaced with actual words at the end of this process are assumed to be the log format's variable parts.

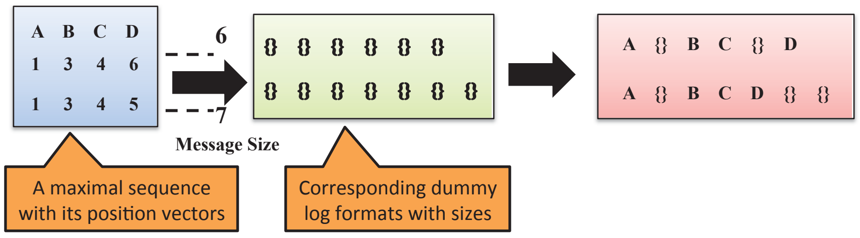

Figure 5.2.: Log format construction from a maximal-sequence.

Algorithm 1 highlights the steps discussed above. During the first iteration, DMAC uses SPADE to mine the system execution trace. The identified frequent-sequences are used to construct candidate log formats. The candidate log formats are used to prune the current system execution trace—generating a new one (line 15). The new system execution trace is then used in the next iteration. This process continues until a satisfiable coverage percentage (*i.e.*, a measure of the number of log

Algorithm 1 Algorithm for constructing log formats from system execution traces.

```
1: procedure FindLogFormats($(D, R, \sigma)$)
2:     $D$: The system execution trace (Initial system execution trace)
3:     $R$: Required coverage
4:     $\sigma$: Initial min-sup
5:     $L$: Final log formats
6:     $|D|$: Number of messages in D
7:
8:     $d \leftarrow D, L \leftarrow \emptyset, C \leftarrow 0$
9:     $i \leftarrow 1, \mu \leftarrow \sigma$
10:
11:    while $C \leq R$ or $i \leq MaxIterations$ do
12:        $\mu \leftarrow \sigma \times \frac{1}{i} \times \frac{|D|}{|d|}$
13:        $F \leftarrow MineSequences(d, \mu)$
14:        $l \leftarrow FindLogFormats(d, F)$
15:        $d \leftarrow CreateNextSystemExecutionTrace(d, l)$
16:        $L \leftarrow L \cup l$
17:        $C \leftarrow C + CalculateCoverage(d, l, D)$
18:        $i \leftarrow i + 1$
19:    end while
20:    return $L$
21: end procedure
```

messages covered by the identified log formats) is met or a predetermined number of iterations is exceeded.

For example using the frequent-sequences in Listing 5.1, DMAC identifies the log formats shown in the Listing 5.4 where empty placeholders ({}) represent variable parts.

```
LF1  -  {} sent message {} at {}
LF2  -  {} received message {} at {}
```

Listing 5.4: Log formats for the first iteration.

This set of log formats is now used to prune the original system execution trace in Table 5.1 and create the new system execution trace shown in Listing 5.5.

```
Got the authentication for request at 30
Access denied at 60
Got the authentication for request at 90
```

Listing 5.5: Example trace for iteration 2.

One challenge in the iterative process is calculating the *min-sup* value for the upcoming iteration. DMAC addresses this challenge by dividing the initial *min-sup* value by the current iteration number and scaling it to a higher value by multiplying it from the ratio of the sizes of the original system execution trace to the pruned one (line 12). This guarantees that mining process starts from a relatively higher *min-sup* value in each iteration.

In the example, DMAC applies frequent-sequence mining with *min-sup* 0.49 ($0.33 \times \frac{1}{2} \times \frac{9}{3}$) to the system execution trace in Listing 5.5 for its second iteration. This produces the frequent-sequences in Listing 5.6.

```
Got the, Got authentication, Got for, Got at,
the authentication, the for,
the at, authentication for, authentication at, for at,
Got the authentication, Got the for, Got the at,
Got authentication for, Got authentication at, Got for at,
Got the authentication for, Got the authentication at,
Got the for at, Got the authentication for at,
Got authentication for at, the authentication for,
the authentication at, the for at, the authentication for at,
authentication for at
```

Listing 5.6: Frequent sequences for the iteration 2.

If the new *min-sup* exceeds 1 it sets it to 1 for the mining process. If it cannot find any new frequent sequences in any iteration the *min-sup* is decreased and mining process continues. When DMAC applies to the frequent-sequences of the system execution trace in Listing 5.5, it identifies the log format in Listing 5.7.

```
LF3  -  Got the authentication for request at {}
```

Listing 5.7: Resulting log formats from iteration 2.

Again, the log format in Listing 5.7 is used to prune the current system execution trace, which creates the new system execution trace shown in Listing 5.8.

```
Access denied at 60
```

Listing 5.8: Example trace after iteration 3.

Finally, another iteration of mining with *min-sup* value of 1 (Using the equation in the algorithm) is applied to the latest system execution trace. This process results in the log format shown in Listing 5.9.

```
LF4 - Access denied at 60
```

Listing 5.9: Resulting log formats from iteration 3.

DMAC's iterative process has two advantages over using a single iteration with a lower *min-sup* value. First, it reduces the possibility of variable parts being selected as static parts in the final set of log formats. This is because starting with a higher *min-sup* value and pruning log messages after each iteration guarantees that lower *min-sup* values are used on a smaller system execution traces. On the other hand, if DMAC uses a single, low *min-sup* value, then DMAC runs the risk of selecting variable parts as static parts. This issue is reflected in Nagappan et al. [39] approach when we integrated it into DMAC for identifying log formats. Secondly, it enables achieving a high coverage. This is because the iterative mining process increases coverage after each iteration since infrequent sequences in an iteration become frequent in successive iterations.

5.2.3 Mining Causal Relationships.

The end goal of mining causal relationships is to find the cause-effect graph from a given set of log formats. Given a pair of log formats ($LF_i \rightarrow LF_j$), they can either occur in the same execution context or in two different execution contexts. Because of this DMAC employs a two step process when mining causal relationships.

Step 1. In the first step, DMAC identifies causal relationships among log formats that occur in the same execution context using the algorithm shown in Algorithm 2.

The steps in Algorithm 2 can be summarized as follows:

1. For each execution context, all its log messages are compared with the identified log formats to produce an execution order, such as `<LF1, LF2, LF3, LF1, ..., LF3>`. DMAC only analyzes adjacent log format pairs in the execution order because it is possible to order events using transitivity. For example, if ($LF1 \rightarrow LF2$) and ($LF2 \rightarrow LF3$) are valid relations, then ($LF1 \rightarrow LF3$) is a valid relation.

2. For each adjacent log format pair in the execution order (*i.e.*, OL_i and OL_{i+1}) the earliest position of OL_i in the execution order should always be less than

Algorithm 2 Algorithm for mining causal relationships in a single execution context.

```
procedure MineCausalOne($D, G, LF, E$)
    $D$: The system execution trace (Initial system execution trace)
    $G$: Directed Acyclic Graph
    $LF$: Final log formats
    $E$: Current Execution Context
    $OL$: Execution Order

    $OL \leftarrow CreateLogFormatOrderList(E, LF, D)$

    for all $(OL_i, OL_{i+1}) \in OL$ do
        $m \leftarrow FirstPositionOf(OL_i)$
        $n \leftarrow FirstPositionOf(OL_{i+1})$
        if $m \leq n$ then
            $ExtendGraph(OL_i, OL_{i+1}, G)$
        end if
    end for
end procedure
```

the earliest position of OL_{i+1} to be considered a valid relation. This ensures no cycles exist and directed acyclic graph properties are not violated.

3. Finally, the `ExtendGraph` function in line 14 adds the log format pair to the graph if they are not already in the graph and do not have an edge between them.

To provide a better understanding of the steps above, let us reconsider the system execution trace from Table 5.1. For this example, we assume that they are generated from the same execution context. This system execution trace produces an execution order similar to the following:
$< LF1, LF2, LF3, LF1, LF2, LF4, LF1, LF2, LF3 >$
In this case, DMAC will identify $(LF_1 \rightarrow LF_2)$, $(LF_2 \rightarrow LF_3)$, $(LF_2 \rightarrow LF_4)$ as valid relations. DMAC, however, will not identify $(LF_3 \rightarrow LF_1)$ and $(LF_4 \rightarrow LF_1)$ as valid relations because it forms a cycle.

Step 2. In the second step, DMAC identifies causal relationships among log formats that occur in different execution context. This, however, is a challenge when compared to Step 1 because any two events that happen in different execution contexts can have a causal relationship between each other [69]. This implies that there is a level of uncertainty associated with causal relationships between events that occur in different execution contexts.

To address this challenge, DMAC uses a probabilistic approach since probabilistic frameworks are most suited for causality mining [70]. More specifically, DMAC uses Dempster-Shafer (DS) theory [71], which is is a mathematical theory of evidence, to mine causal relationships for log formats that occur in different execution contexts. The advantage of using Dempster-Shafer theory instead of Bayesian decision model [41] is that it does not rely on a trained model of some prior known relationships. Instead DS theory increases the belief on a particular decision depending on the evidences it has collected.

The Section 5.2.3.1 provides a brief overview on Dempster-Shafer (DS) theory.

5.2.3.1 A Brief Overview on Dempster-Shafer (DS) Theory

In DS theory, a set of mutually exclusive and exhaustive set of hypothesis, $\Theta = \{h_1, ..., h_k\}$, are referred to as Frame of Discernment (FoD). A hypothesis h_i is referred to as a singleton. The *basic belief assignment, (bba)* function, also called the *mass distribution function* m, distributes belief over the power set of the FoD and is defined as follows:

$$m : 2^{\theta} \rightarrow [0, 1] \tag{5.2}$$

$$\forall x \subseteq \Theta, m(x) \geq 0 \tag{5.3}$$

$$m(\emptyset) = 0 \tag{5.4}$$

and

$$\sum_{x \subseteq \Theta} m(x) = 1 \tag{5.5}$$

The belief function is defined as

$$\forall x \subseteq \Theta, Bel(x) = \sum_{y \subseteq x} m(x) \tag{5.6}$$

$$Bel(\emptyset) = 0 \tag{5.7}$$

and

$$Bel(\Theta) = 1 \tag{5.8}$$

The belief function is a measure of how much confidence we have for a certain hypothesis to be true, whereas the *bba* specifies the weight (mass) of a particular evidence source has to support a given hypothesis. Note that $Bel(x) = m(x)$, if x is a singleton.

Because there can be many sources of evidences for a particular proposition DS theory provides Dempster's rule to combine the evidences. Let's say h_1 and h_2 are two propositions and h is the resulting proposition of intersecting h_1 and h_2.

$$m(h) \equiv (m_1 \oplus m_2)(h) = \frac{\sum_{h_1 \cap h_2 = h} m_1(h_1).m_2(h_2)}{1 - K} \tag{5.9}$$

$$K = \sum_{h_1 \cap h_2 = \emptyset} m_1(h_1).m_2(h_2) \tag{5.10}$$

is referred to as the *conflict* because it indicates how much the evidences of the two sources are in conflict.

Now we describe how we have used DS theory in the context of mining causal relationships among log formats. The Frame of Discernment (FoD *i.e.* the set of mutually exclusive hypothesis) in mining causal relations is $\{(LF_i \rightarrow LF_j) = Yes, (LF_i \rightarrow LF_j) = No\}$. We define evidence $m((LF_i \rightarrow LF_j) = Yes)$ for supporting the relationship according to a particular source of evidence, and evidence $m((LF_i \rightarrow LF_j) = No)$ for disqualifying the causal relation according to a particular evidence source. $m((LF_i \rightarrow LF_j) = Yes, (LF_i \rightarrow LF_j) = No)$ is the measure of uncertainty, or the ignorance, that a particular evidence source has about the relation. Unlike traditional probability theory if $m((LF_i \rightarrow LF_j) = Yes) = p$, then it does not necessarily mean that $m((LF_i \rightarrow LF_j) = No) = 1-p$. This is because sources of evidence only support $\{(LF_i \rightarrow LF_j) = Yes\}$ and its ignorance about $\{(LF_i \rightarrow LF_j) = No\}$ should be assigned to $\{(LF_i \rightarrow LF_j) = Yes, (LF_i \rightarrow LF_j) = No\}$.

We define the belief function as $Bel((LF_i \rightarrow LF_j) = Yes)$ in causal relation mining. It denotes the confidence we have to support the causal relation between LF_i and LF_j after combining evidence values from different evidence sources. DMAC uses the Dempster's rule to combine evidences and to handle conflicting evidences.

Algorithm 3 illustrates the major steps DMAC uses to apply DS theory.

As shown above, DMAC first verifies if the two log formats occur in different execution contexts (line 9). DMAC then checks if the corresponding nodes for these

Algorithm 3 Algorithm for mining causal relationships between log formats in different execution entities.

```
1: procedure MineCausalTwo($(G, LF, EL, \lambda)$)
2:     $G$: Directed Acyclic Graph
3:     $LF$: Final log formats
4:     $E$: Execution Context List
5:     $\lambda$: Belief Threshold
6:     $B$: Belief value
7:
8:     for all $LF_i, LF_j$ do
9:         if $HasCommonEE(LF_i, LF_j, E)$ then
10:            $Continue$
11:        end if
12:        if $AreReachable(LF_i, LF_j, G)$ then
13:            $Continue$
14:        end if
15:        $B \leftarrow CalculateBelief(LF_i, LF_j)$
16:        if $B \geq \lambda$ then
17:            $ExtendGraph(LF_i, LF_j, G)$
18:        end if
19:    end for
20: end procedure
```

log formats in the current graph are reachable from each other (line 12). This is necessary because it implies a transitive relationship.

Lastly, DMAC evaluates evidence from different sources and combines it using Dempster's rule to associate a single belief value $b \in [0,1]$ to any log format pair (line 15). The belief value is a measure of confidence about the causal relation between the two log formats. Currently, DMAC uses the following three evidence sources to calculate the belief value of a particular causal relation $LF_i \rightarrow LF_j$:

1. **Time evidence.** Time evidence is based on the observation that two events can be causally related if they occur within a certain time window [41]. To calculate the time evidence value, DMAC first checks whether there is at least one LF_j log message occurs within the time window for each LF_i log message, and calculates a ratio of identified LF_j to the number of LF_i log messages. The ratio is then multiplied by a weight based on the ratio between actual time difference between the two events and the time window, that reflects how "easy" for two events satisfy the time window requirement. For example, the closer the time difference is to 0, the closer the weight is to 1.

   Time evidence only assigns a *bba* to the $\{(LF_i \rightarrow LF_j) = Yes\}$ hypothesis. Failing to satisfy the timing window does not necessarily mean $\{(LF_i \rightarrow LF_j) = No\}$. DMAC therefore assigns $1-p$ value to the hypothesis $\{(LF_i \rightarrow LF_j) = Yes, (LF_i \rightarrow LF_j) = No\}$ Lastly, when calculating the time evidence for any candidate causal relation $\{(LF_i \rightarrow LF_j) = Yes\}$, we assume that clocks in the different execution contexts are synchronized. If there is high clock drift between the different execution contexts, then we decrease the calculated evidence value by multiplying it with a weight that decreases the confidence about this event source.

2. **Variable evidence.** Variable evidence is based on the observation that $LF_i \rightarrow LF_j$ can be true if they both have variable parts that match across many occurrences of the log messages. Based on this observation DMAC calculates the

bba for the hypothesis $\{(LF_i \rightarrow LF_j) = Yes\}$ using the approach presented in Algorithm 4.

Algorithm 4 Algorithm for calculating Variable evidence.

```
1: procedure CalculateVarEvidence($LF_i, LF_j$)
2:     $LF_i$: Candidate Cause log format
3:     $LF_j$: Candidate Effect log format
4:     $V_i$: Set of variable values of $LF_i$
5:     $V_j$: Set of variable values of $LF_j$
6:     $M$: Map of $\{(v_m, v_n), c\}$
7:     $v_m \in V_i, v_n \in V_j$
8:     $c$: Count of each matching $(v_m, v_n)$
9:
10:    for all $LF_i messages$ do
11:        for all $LF_j messages$ do
12:            for all $(v_m, v_n)$ do
13:                if $v_m = v_n$ then
14:                    increment corresponding $c$ in $M$
15:                end if
16:            end for
17:        end for
18:    end for
19:    $S \leftarrow Max(c)$
20:    $n \leftarrow |LF_i|$
21:    return $\frac{S}{n}$
22: end procedure
```

As shown in this algorithm, if LF_i has m variables and LF_j has n variables, then for any two messages it requires $m \times n$ comparisons. The comparison of variables can be computationally expensive, but most log formats do not have more than 4 variable. Within the $m \times n$ iterations, the algorithm checks whether the values of the two variables are equal (line 13). If the variable values are equal, then the algorithm increments the counter that tracks the number of instances satisfying the variable relation candidate in line 13. After iterating over all the variable parts, the algorithm outputs portions of the messages that satisfy the log format variable relationship. Lastly, the algorithm calculates the *bba* for $\{(LF_i \rightarrow LF_j) = Yes\}$ hypothesis using the maximum count of the $m \times n$ variable relation candidates (line 19).

3. **Domain-specific evidence.** The two evidence sources previously discussed are common across all system execution traces. There can be cases when time and variable evidence is not enough to construct a valid dataflow model. When this situation occurs, we rely on domain-knowledge to integrate domain-specific evidences (*i.e.*, the evidence that pertains only to the dataflow model and corresponding system execution trace). Because of the need to incorporate domain-knowledge into DMAC, we have created a framework that allows testers to integrate their own domain-knowledge about the system execution trace as another evidence source. Figure 5.3 shows the general architecture for how domain-

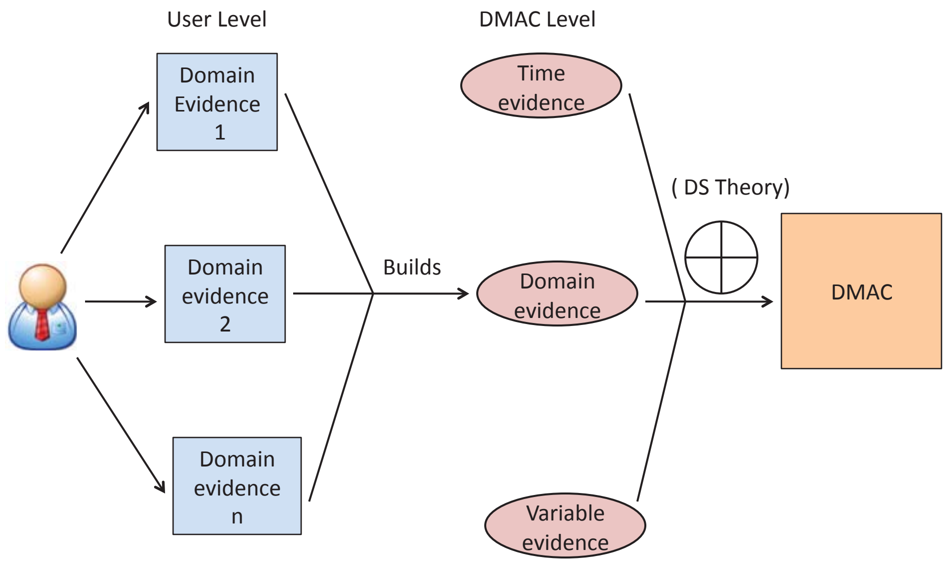

Figure 5.3.: Combing evidences for causal relation mining process.

specific evidence is integrated into DMAC. As shown in this figure, the user can specify domain-specific evidence at the DMAC-level, or the user-level, which is converted to DMAC-level evidences. At the user-level, the tester specifies knowledge about causal relationships between log messages using a natural language and associates a quantitative value between 0 and 1 with each piece of knowledge. For example, $(\text{“}send\text{”} \rightarrow \text{“}receive\text{”}, 0.6)$ means that corresponding log formats that match log messages with either “send” or “receive” are causally related with 0.6 certainty. At the DMAC-level, the tester specifies knowledge about causal relationships between log formats. For example, $(LF_i \rightarrow LF_j, 0.6)$ means that LF_i causes LF_j with 0.6 certainty.

DMAC then uses the DMAC-level domain-specific evidence—along with the time and variable evidence—to auto-construct the dataflow model. It is worth noting that domain-specific evidence is not required for DMAC to work correctly. It, however, is useful to incorporate domain-specific evidence if the time and variable evidences are not producing an accurate dataflow model. Section 5.3 discusses results related to this observation.

5.2.4 Identifying Causal Relationships among Variables.

The final part of constructing the dataflow model is identifying relationships between variable parts of different log formats. For each log format variable, DMAC keeps all the values extracted from its corresponding log messages. This is called the *value set*. When a valid relation is identified, the *value sets* are compared using an algorithm similar to the one shown in Algorithm 4. This happens during the causality mining process between log formats described above.

Finally, using information gathered from the multiple mining phases described above, DMAC generates a dataflow model for the entire system.

5.3 Results of Applying DMAC to Open-Source Projects

This section discusses experimental results from applying DMAC to several open-source projects, and the accuracy of its constructed dataflow models.

5.3.1 Experimental Setup

Similar to the case studies described in Section 4.3 We applied DMAC to the system execution traces generated by Apache ANT, Apache Tomcat, Apache ActiveMQ and DAnCE. We first manually constructed their dataflow model after going through the system execution trace and the source code. We therefore have "ground truth" for these projects' dataflow models. Another reason is that these projects have diverse system execution traces—giving DMAC a range of case studies. For example, Apache Tomcat's system execution trace is dense compared to Apache ANT, Apache ActiveMQ, and DAnCE. DAnCE's system execution trace, however, has less reoccurring patterns compared to ANT and ActiveMQ.

5.3.2 Experimental Results for Execution Traces without Domain-specific Evidence

5.3.2.1 Experimental Results for ANT

ANT's system execution trace used in this experiment contained 8100 log messages. We used an initial *min-sup* value of 0.8. As shown in Listing 5.10, DMAC identified 7 log formats, which cover 94.85% of total system execution trace. DMAC was able to correctly identify the static and variable parts in all 7 log formats. DMAC identified `LF3` although it has occurred only once in the system execution trace. This is because its corresponding candidate sequence is same as that of `LF2`, which occurs frequently. Because of the difference in position vectors, DMAC was able to distinguish between the two different log formats. We did not use domain-specific evidence for ANT's system execution trace because it has only one execution context. DMAC identified 12 relations as shown in Listing 5.10. One of the identified relations has a cause-effect relationship between its variables (*i.e.*, $LF4.1 = LF5.1$). This means that variable 1 in `LF4` has a cause-effect relationship with variable 1 in `LF5`.

```
LF1 = {} skipped - don't know how to handle it {}
LF2 = {} omitted as {} is up to date. {}
LF3 = omitted as {} is up to date. {}
LF4 = Task {} started. {}
LF5 = Task {} finished. {}
LF6 = adding directory {} {}
LF7 = adding entry {} {}

LF1->LF2; LF1->LF6; LF1->LF7; LF2->LF6; LF2->LF7;
LF2->LF3; LF4->LF5; LF4->LF1;LF4->LF2; LF4->LF7;
LF4->LF6; LF6->LF7

LF4.1 = LF5.1

Total Records = 8100
LF1 - Count = 1783; Percent = 22.0123%
LF2 - Count = 3488; Percent = 43.0617%
LF3 - Count = 1; Percent = 0.0123457%
LF4 - Count = 159; Percent = 1.96296%
LF5 - Count = 158; Percent = 1.95062%
LF6 - Count = 294; Percent = 3.62963%
LF7 - Count = 1800; Percent = 22.2222%
Total coverage : 94.8518%
```

Listing 5.10: Results for applying DMAC to ANT's system execution trace.

ANT's dataflow model constructed by DMAC is useful for analyzing its performance properties. For example, `LF4` and `LF5` can be used to evaluate the execution time of different ANT tasks, which we have explained in Section 4.3.2.

5.3.2.2 Experimental Results for ActiveMQ

ActiveMQ's system execution trace contained 3650 log messages. Although ActiveMQ's system execution trace contained fewer messages than ANT, it contained more log formats. DMAC's frequent-sequence mining step produced few sequences for relatively high *min-sup* value. For example, an initial *min-sup* of 0.16 generated only 2 frequent-sequences; whereas *min-sup* of 0.03 produced 6000 frequent-sequences.

In the latter case, executing DMAC for one iteration produced a dataflow model with 12 log formats and 50.96% coverage. DMAC's iterative mining-process improved when starting from a *min-sup* of 0.16 by producing a dataflow model with 80.23% coverage. This dataflow model contained of 21 log formats and 31 relations.

5.3.2.3 Experimental Results for Apache Tomcat

Tomcat's system execution trace contained 101700 log messages produced by six different threads. We started DMAC with an initial *min-sup* value of 0.05. After 7 iterations, DMAC identified 89 log formats that covered 61.95% of the system execution trace. DMAC also identified 318 relations. We stopped DMAC after 7 iterations because after 7 iterations DMAC started identifying variable parts of the log messages as static parts. Finally, we validated that the dataflow model constructed by DMAC could analyze the same performance properties of Tomcat that were analyzed using a manually constructed dataflow model in prior work [72].

5.3.2.4 Measuring the Auto-constructed Dataflow Model's Accuracy

For ANT, ActiveMQ, and Tomcat, we were able to use the constructed dataflow model to analyze performance properties that were analyzed using a manually constructed dataflow model. Although the auto-constructed dataflow models were correct, we need to validate the accuracy of its log formats and relations in relation to the original source code.

We evaluated the accuracy of a log format with respect to (w.r.t.) its counterpart in the original source by comparing the static and variable parts of the log format correspond with static and variable parts in the originating log message from source code. We also evaluated the accuracy of a log format with respect to the system execution trace by evaluating that (1) the static part of a log format is constant in all corresponding log messages; and (2) the empty placeholder ({}) corresponds to at least two possible values in all the corresponding log message instances in the system execution trace.

Table 5.2 shows the accuracy results for the Apache, ANT, and ActiveMQ results previously discussed. Most of the inaccuracy presented in Table 5.2 is because variable parts were identified as static parts. This has a direct relationship with the structure of the system execution trace. For example, ANT's system execution trace is very succinct; whereas, ActiveMQ and Tomcat's system execution trace is verbose.

Table 5.2.: Accuracy of auto-constructed log formats.

Item	**ANT**	**ActiveMQ**	**Tomcat**
# of identified log formats	7	21	89
# of identified log formats checked for correctness	7	21	25
LFs correct w.r.t. source	7	14	18
LFs correct w.r.t. execution trace	6	15	20
source code accuracy	100%	66.7%	72%
execution trace accuracy	85.7%	71.4%	80%

We evaluated the dataflow model's relation accuracy similar to how we evaluated the log format's accuracy. More specifically, we evaluated $(LF_i \rightarrow LF_j)$ accuracy by comparing whether they actually occur in the original source code. When the relation is from the same execution context we check whether the two log formats are generated from adjacent log statements in the source code. When the relations are in different execution contexts we check whether the two log statements represent initiation or completion of a remote procedure call in the source code.

We were able to conclude that the identified relations are 100% accurate when the log formats are from the same execution context. The next section describes an experiment that was conducted to evaluate relation accuracy when the two log formats are from different execution contexts.

5.3.3 Experimental Results for Execution Traces with Domain-specific Evidence

The DAnCE system execution trace was small compared to that of Tomcat, ANT and ActiveMQ. Most of the words in the DAnCE's log messages that describe an event have a low frequency value. Furthermore, the frequency values of log message metadata, *e.g.*, log message severity, was greater than the frequency values for actual log message content. DMAC therefore interprets the metadata as static parts and the remaining content of the actual message as variable parts.

```
[LM_TRACE] - - plan,  [LM_TRACE] - - for,
[LM_TRACE] - - instance,  [LM_TRACE] - - plugin,
 - - artifact,[LM_TRACE] - - from,  [LM_TRACE],
 - - installation, [LM_TRACE] - - successfully,
 [LM_TRACE],  - - to,  - - handler
```

Listing 5.11: Frequent sequences for DAnCE's system execution trace.

For example, Listing 5.11 shows frequent-sequences identified by DMAC when the intermediate *min-sup* is 0.17. These candidate sequences are not sufficient to build a log format. Because of this, DMAC considers remaining parts of the log format as variable parts. Listing 5.12 shows some of the log formats identified by DMAC for DAnCE. From this experiment, we concluded that DMAC does not work well with this system execution trace because it does not have a high frequency value for the words that describe an event when compared to other parts of the log message.

Moreover, it is harder to construct DAnCE's dataflow model since many of the log formats occur in different execution contexts—unlike our previous experiments.

```
LF1 = {} [LM_TRACE] - {} - {} {} {} - {} {} {}
for name {}
LF2 = {} [LM_TRACE] - {} - {} - {} {} {}
for name {}
LF3 = {} [LM_TRACE] - {} - {} - {} {}
for name {} {} {} {}
LF4 = {} [LM_TRACE] - {} - {} {} {} - {} {}
for name {} {} {} {}
```

Listing 5.12: Some of the log formats identified by DMAC for DAnCE's system execution trace.

Because we were not able to auto-construct a valid dataflow model from DAnCE using only time and variable evidence, we added domain-specific evidence to the auto-construction process. Based on our domain-knowledge of DAnCE, we defined domain-specific evidence at the DMAC-level such that we specified a uniform *bba* value for each causal relation between log formats. More specifically, if our confidence about the domain knowledge is 0.8, then we assigned a *bba* value of 0.8 for each relation $LF_i \rightarrow LF_j$, *i.e.*, $(\{(LF_i \rightarrow LF_j) = YES\}, 0.8)$. Likewise, we assigned a *bba* value of 0.8 for the hypothesis $\{(LF_i \rightarrow LF_k) = NO\}$ for any relation we knew could not occur in the dataflow model.

We then used DMAC with the added domain-specific evidence to auto-construct the dataflow model for DAnCE. Because the domain-specific evidence is designed to produce more accurate results, we evaluated the effect of domain-specific evidence on affecting *true-positives (TP)* and *false-positives (FP)* in the auto-construction process. Figure 5.4 shows the effect that the confidence level of domain-specific evidence has on TPs and FPs. As shown in this figure, as we increase our confidence level, the number of TPs increases and the number of FPs decrease. Likewise, when we reduce

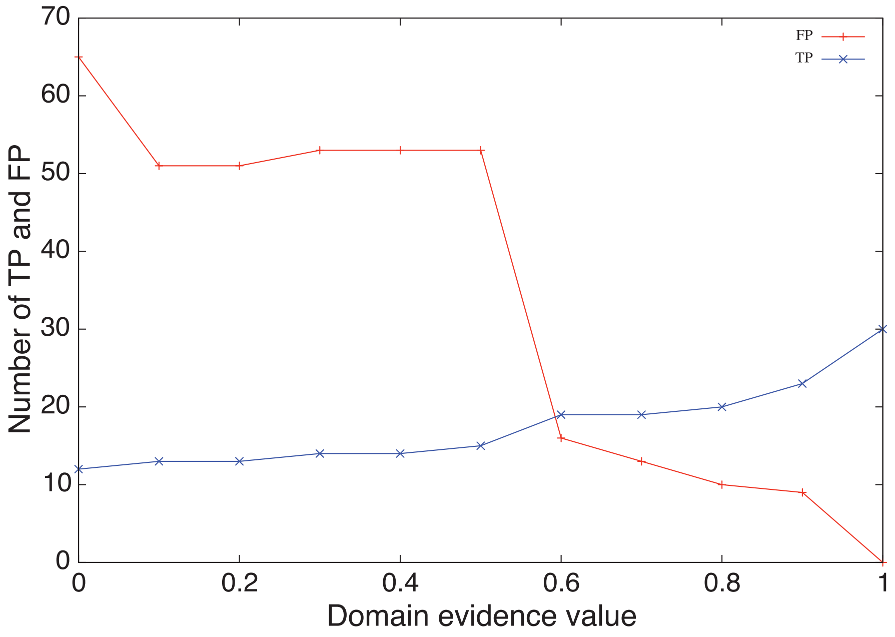

Figure 5.4.: Effect on domain-knowledge on TPs and FPs in the dataflow model auto-construction process.

our confidence level, the opposite occurs. For example, when the confidence level is 1 (*i.e.*, the tester has complete knowledge and confidence), then DMAC produces the most accurate results. Likewise, a confidence level of 0 produces results similar only using time and variable evidence in the auto-construction process.

The DAnCE results show two evidences (*i.e.*, time and variable) are not always enough to correctly auto-construct a dataflow model. In some cases, it may be necessary to integrate domain-specific evidence. As illustrated in the DAnCE results, DMAC is able to successfully integrate domain-specific evidence into the dataflow model auto-construction process. It is therefore the responsibility of the tester to identify the domain-specific evidence and quantify it correctly to reduce the number of FPs.

5.4 Summary of Contributions

In this chapter we presented the Dataflow Model Auto Constructor (DMAC), which is a tool and technique to auto-construct dataflow models from system execution traces. The auto-constructed dataflow models can be used to do software performance analysis and to reason about software performance results. The following are the key contributions of DMAC:

- A frequent-sequence mining based iterative algorithm to identify abstract event types from execution trace data
- An evidence theory based causality relationship mining algorithm, which can be used to identify causal relationships between abstract event types of the system execution trace.

6 DETECTING EXCESSIVE DYNAMIC MEMORY ALLOCATIONS ANTI-PATTERN USING SYSTEM EXECUTION TRACES

In Chapter 1 we described the importance of detecting excessive dynamic memory allocations anti-pattern. In Section 2.3 of Chapter 2, we have already discussed the related research on detecting dynamic memory allocations anti-pattern. This chapter first describes challenges associated with detecting excessive dynamic memory allocations anti-pattern. Then, we describe Excessive Dynamic Memory Allocations Detector (EMAD), which is our novel contribution for detecting excessive dynamic memory allocations anti-pattern with minimal user intervention. Finally, we describe results of applying EMAD to different open source software systems.

6.1 Challenges Addressed by Proposed Approach

Excessive dynamic memory allocation is a common problem known to degrade the performance of a software system. Because of this reason, many popular software systems and libraries adopt solutions that amortize the cost of allocating/deleting memory, such as allocating memory from memory pools (custom memory allocators) or free lists [73]. Another solution is to use the Flyweight software design pattern [74]. Although these promising solutions are available, it is *hard* to apply them if one cannot detect the excessive dynamic memory allocation anti-pattern. Unfortunately, detecting the excessive dynamic memory allocation anti-pattern poses several challenges:

1. **Inapplicability of source code analysis techniques.** As mentioned in Section 1, the prominent approach for detecting a software performance anti-pattern is source code analysis. Understanding dynamic memory allocations by just analyzing the source code, however, is *hard*. This is because key information

like frequency of object allocation, the size of the object being allocated, and the lifetime of an object are hard to determine at compile time. Moreover, such analysis requires time-consuming code analysis involving experts of complex software systems [75].

Another limitation of this approach is that it requires source code to be available. Nowadays, most software systems are built using off the shelf software components and libraries. It is therefore ill-conceived to assume that source code is available for analysis at every situation. Even if the source code is available (as with open-source projects), one must still be able to understand the source code (and its intent) in order to search for excessive dynamic memory allocations.

2. **Limitations of software performance anti-pattern detection techniques based on architectural models.** Another approach for detecting software performance anti-patterns is defining rules on performance metric data (*e.g.*, response time and throughput) and/or resource usage data (*e.g.*, CPU and network usage) and then detecting rule violations [24, 25, 34]. These rules are defined on architectural models of the system and rule violations are analyzed by simulating the architectural models. Excessive dynamic memory allocation, however, happens at software implementation level. This makes it hard to model the minute details of an implementation, and detect the excessive dynamic memory allocation anti-pattern [34].

   On the other hand, resource usage data (*e.g.*, high memory footprint) is not a direct indicator of excessive dynamic memory allocations. This is because a function can do a large allocation at once and then use it subsequently throughout the application lifetime. This is all while not doing any frequent allocations.

3. **Ill definition of excessive dynamic memory allocation problem.** The problem of detecting excessive dynamic memory allocations is ill defined compared to other dynamic memory associated problems like memory leak detection

and invalid memory access detection. For example, memory leak detection can be defined as finding dynamic memory allocations that are no longer accessible to the program [76]. Likewise, memory access errors can be defined as detecting invalid reads/writes from/to memory locations.

```
struct Foo {
  int x;
};

int main (int argc, char * argv[]) {
  Foo * foo = new Foo ();

  // Do someting with foo
  // ...

  return 0;
}
```

Listing 6.1: A simple program that has a potential memory leak.

For example, Lisiting 6.1 illustrates a simple program that has a potential memory leak. As shown in the program, we can conclude that a memory leak exists by examining whether the object *foo* is, or is not, released when the main function returns. Although this examination process can be complex, the problem of detecting the memory leak is well defined.

```
struct Foo {
  int x;
};

int main (int argc, char * argv[]) {
  for (int i = 0; i < 1000000; i ++) {
    Foo * foo = new Foo ();

    // Do something with foo
    // ...

    delete foo;
  }

  return 0;
}
```

Listing 6.2: A simple program that has a potential excessive dynamic memory allocation.

> Excessive dynamic memory allocations, however, cannot be defined in such a precise manner. The word “excessive” depends heavily on the context of the allocation. For example, Listing 6.2 illustrates a simple program that has a potential excessive dynamic memory allocation issue because of the high frequency at which *foo* is being created and deleted. It, however, is hard to determine whether this simple example exhibits excessive dynamic memory allocations by only examining the number of times object *foo* is being created and deleted. This is because excessive dynamic memory allocation is not only based on how many allocations/deallocations occur, but also on the lifetime of those allocated objects.

As discussed above, these challenges make it hard to create automated approaches for detecting excessive dynamic memory allocation anti-pattern. The reminder of this chapter will therefore discuss how EMAD helps address these challenges–providing software developers with an improved approach to detect the excessive dynamic memory allocation anti-pattern. This will allow software developers to detect and resolve the anti-pattern problem faster and improve the performance of their software application.

6.2 The Approach of EMAD

Our approach for detecting the excessive dynamic memory allocations anti-pattern is supported by Dynamic Binary Instrumentation (DBI). DBI [5, 77] is the process of instrumenting a software application at runtime as opposed to recompiling the software application with the instrumentation software. DBI does not require the source code of the system being instrumented because instrumentation logic is injected into the target application while the program's binary is executing. By using DBI, we are able to create an approach that overcomes the challenge of needing the source code to detect the existence of excessive memory allocation. Moreover, DBI allows us to trace an application and therefore capture its behavior, which is one of the main weaknesses in most of the exisiting software performance anti-pattern detection techniques. The bigger challenge now is understanding how to apply DBI to actually detect excessive dynamic memory allocations in an existing application, or middleware. Our proposed technique is based on the intuition that this anti-pattern occurs when the software applications have many short-lived high-frequent dynamic memory allocations.

Figure 6.1 illustrates EMAD's workflow for detecting the excessive dynamic memory allocations anti-pattern. As shown in the figure, the process consists of 3 major steps: (1) instrumenting the software application using DBI to collect an execution trace; (2) constructing a call graph of the software from the collected execution trace; and (3) analyzing the call graph to detect excessive dynamic memory allocations. We discuss each step in detail throughout the remainder of this section.

6.2.1 Instrumenting the Software Application

EMAD uses Pin [5] along with Pin++ [78] as the underlying DBI framework to instrument an application and collect the needed execution trace. Sidebar 1 provides

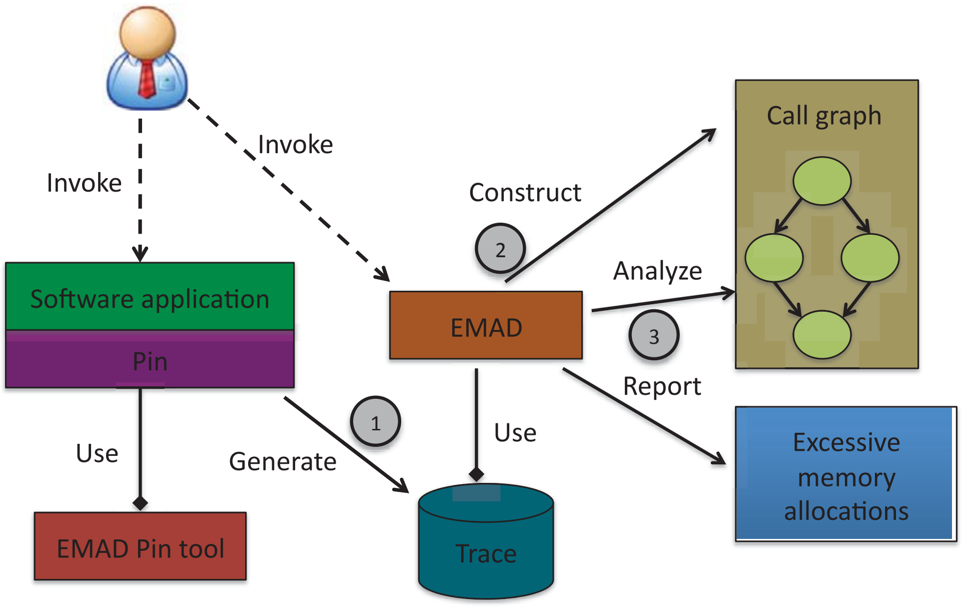

Figure 6.1.: Conceptual overview of EMAD's workflow.

a brief overview on Pin and Pin++. EMAD uses Pin++ to implement a Pintool that instruments a program at routine level. The Pintool instruments[1] each routine call at start (*i.e.*, invocation) and at exit (*i.e.*, return to caller). The Pintool then generates an execution trace during the execution of the application that has messages similar to the one shown in Figure 6.2.

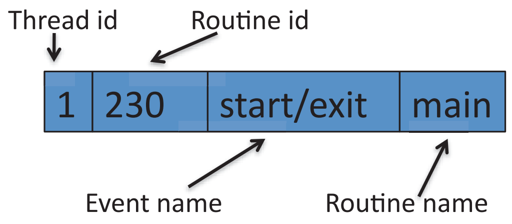

Figure 6.2.: Format of a message in the execution trace

As illustrated in this figure, EMAD records the following information for each instrumented routine:

- **Thread id**. The thread id is a unique identifier of the thread calling the routine under analysis. This is important because the caller-callee relationships between routines are determined on a per thread basis when constructing the call graph. The thread id therefore is used to uniquely identify the thread.

- **Routine id.** The routine id is a unique id of the routine assigned by Pin. This piece of information is important because the routine name is not unique if the same routine is in different image or if it is overloaded in the same class. This allows EMAD to uniquely identify each routine it instruments.

- **Event name.** The event name represents the type of event that is occurring. For EMAD, the event name is either *start* or *exit*. Start represents the beginning of a routine call and exit represents the return of a routine call. This piece of information is important because it determines what subprocedures (*i.e.*, the

[1]By instrument, we mean insert hooks to call analysis routine at point of insertion.

Sidebar 1: Pin and Pin++

Pin is a DBI tool for IA-32 and X86-64 instruction-set architecture. Pin provides a framework to implement analysis tools called Pintools. Pintools can be implemented to analyze several aspects of programs, such as program faults, program behavior, root causes, performance profiling. Pintools also analyze a program at different levels of granularities: binary image level, routine level, and instruction level.

Pintools are implemented independently from target programs and are compiled into separate shared libraries. Pintools are not compiled (statically) or linked (dynamically) with the program. When the target program needs to be instrumented, it is executed by Pin providing the Pintool as an argument. This way the target program can be instrumented with different Pintools, and vice versa, without requiring any modifications to the source of the program under instrumentation.

Even though Pin provides several facilities to instrument programs, the Pintools implemented using Pin are fragile, rigid, hard to extend/reuse, and difficult to understand [78]. Pin++ provides an object-oriented, template meta programming approach to writing Pintools that handle the above mentioned software engineering issues. Moreover, Pintools implemented using Pin++ have a reduction in cyclomatic complexity, do not induce additional overhead, and improves the Pintools performance in certain cases. For example, Hill et al. [78] have shown that Pin++ can have a 54% reduction in complexity, increase its modularity, and up to 60% reduction in instrumentation overhead when compared to Pintools implemented the traditional way.

sub-procedure for receiving a start event or the sub-procedure for receiving an exit event) to call in Algorithm 5.

- **Name.** The name represents the undecorated name of the routine under instrumentation (or being analyzed). This piece of information is important because this allows EMAD to report the human readable name of an routine when it identifies the location(s) of excessive dynamic memory allocations.

Because EMAD eventually constructs a call graph (see Section 6.2.2) that also records dynamic memory allocations and deallocations, EMAD assumes signatures with the patterns shown in Listing 6.3 for dynamic memory allocation and deallocation routines. The patterns in this listing are the common signatures for most of the general-purpose memory allocation/deallocation routines in both standard libraries [79,80](*e.g.*, malloc/free and new/delete) and third-party libraries that implement its own memory management strategy [63, 81].

```
// Pattern expected for memory allocation routine.
void * [allocation_method] (size_t size);

// Pattern expected for memory deallocation routine.
void [dellocation_method] (void * location);
```

Listing 6.3: Allocation/Deallocation method signatures.

EMAD also collects the following additional details for allocation/deallocation routines in the execution trace:

- **Allocation size.** This is the input parameter at the start of the allocation routine, which is the size of the allocation. This piece of information is important when characterizing a memory allocation.
- **Address of the allocation.** This is the return value at the exit of the allocation routine, which is the allocated memory location address. This piece of information is important when correlating memory allocations and deallocations.
- **Allocation timestamp.** This is the timestamp when exiting from the allocation routine. It specifies the time when the memory allocation was active. This

piece of information is important when calculating the lifetime of a particular memory allocation.

- **Deallocation timestamp.** This is the timestamp when exiting from the deallocation routine. It specifies the time when the memory allocation was deactivated. This piece of information is important when calculating the lifetime of a particular memory allocation.

The execution trace (*i.e.*, the data discussed above) is recorded by the Pintool while the program under instrumentation is executing. Listing 6.4 shows a portion of an example execution trace the EMAD Pintool will generate. Once the execution trace is recorded, the remainder of EMAD's analysis is done offline.

```
0 19 start main
0 20 start Initialize
0 22 start malloc 32
0 22 exit malloc 842c008 141677579
0 20 exit Initialize
0 34 start operation1
0 22 start malloc 64
0 22 exit malloc 9786cd0 14167757886
0 35 start operation2
0 23 start free 9786cd0
0 23 exit free 14167757928
0 23 start free 842c008
0 23 exit free 14167757928
0 35 exit operation2
0 34 exit operation1
0 19 exit main
```

Listing 6.4: Example execution trace generated by the EMAD Pintool.

6.2.2 Constructing the Call Graph

EMAD uses the execution trace collected during the instrumentation step (see Section 6.2.1) to construct a call graph [82] of the program. The constructed call graph is a weighted directed graph. Each node in the graph represents an executed routine in the application. Each edge represents a caller-callee relationship. The edge weights represent the frequency of each routine call. The Figure 6.3 illustrates the call graph EMAD will be constructing for the execution trace shown in Listing 6.4.

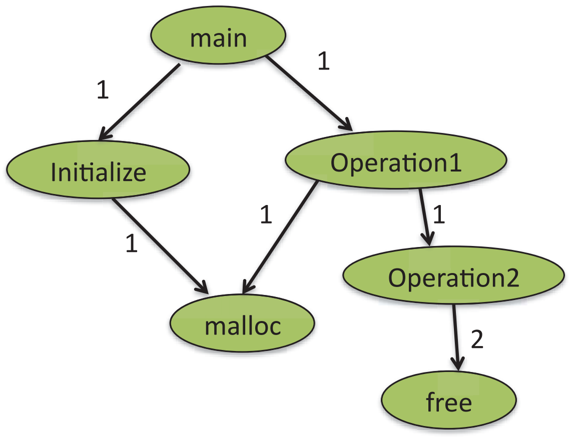

Figure 6.3.: Call Graph for the execution trace in Listing 6.4

The constructed call graph is also a condensed graph [83]. This is because EMAD is not representing each and every call to a routine as its own node and edge as in a detailed call graph. Instead, EMAD is capturing how many times a routine is called. The condensed call graph reduces the amount of resources needed to construct the needed call graph of an application. More importantly, we have learned that a detailed call graph makes it hard to perform the necessary analysis to detect excessive dynamic memory allocations.

Algorithm 5 details EMAD's process for constructing the call graph from an execution trace. The algorithm consists of two sub-procedures. The first sub-procedure handles routine start messages (line 11). The second sub-procedure handles routine exit messages (line 13). It is worth noting that Algorithm 5 maintains a *called routine stack* for each thread in the application being instrumented. This is because caller-

Algorithm 5 General algorithm for constructing the call graph in EMAD.

```
1: procedure ConstructCallGraph($ET$)
2:     $ET$ : set of routine start/exit messages from execution trace
3:
4:     $CG$ : Call graph
5:     $CS$ : Set of stacks of called routines, one per each thread
6:
7:     for all $ET_i \in ET$ do
8:         $j \leftarrow extract\_thread\_id(ET_i)$
9:         $R \leftarrow extract\_routine(ET_i)$
10:        if $ET_i$ is a routine start trace then
11:            $HandleRoutineStartTrace(CG, CS_j, R)$
12:        else if $ET_i$ is a routine exit trace then
13:            $HandleRoutineExitTrace(CG, CS_j, R)$
14:        end if
15:    end for
16:
17:    for all $k \in$ thread ids do
18:        while $CS_k$ is not empty do
19:            $R \leftarrow Top(CS_k)$
20:            $HandleRoutineExitTrace(CG, CS_k, R)$
21:        end while
22:    end for
23:
24: end procedure
```

callee relationships are maintained on a per thread basis when using the condensed graph approach [83]. There, however, will be one call graph that is updated using the relationships maintained in each call stack.

The sub-procedure for handling routine start messages, which is shown in Algorithm 6, is straightforward to understand. Whenever a routine start message is found, the corresponding routine object is pushed onto the stack. A node representing the routine object is also added into the call graph. Because EMAD is constructing a condensed call graph, the *AddNode* statement (line 7) only adds a node to the call graph if and only if the node is not in the call graph.

Algorithm 6 Procedure that handles a routine start trace.

```
1: procedure HandleRoutineStartTrace($CG, cs, R$)
2:     $CG$ : Call graph
3:     $cs$ : The routine stack of a thread
4:     $R$ : The routine
5:
6:     $Push(cs, R)$
7:     $AddNode(CG, R)$
8: end procedure
```

The sub-procedure for handling routine exit messages is not as straightforward when compared to the sub-procedure for handling routine start messages. This is because the instrumentation of routine exits does not work reliably in the presence of tail calls, or when return instructions cannot reliably be detected under Pin [84]. From our experience, a majority of the routine exit messages for the corresponding routine start messages can be found in the execution trace. When a routine exit message cannot be found in the execution trace, EMAD uses Algorithm 7 to resolve the *missing exit message* problem.

As shown in this algorithm, it first checks whether the routine object at the stack top is the same as the routine object represented from the message. If this condition holds true, then this implies that the routine object has both start and exit messages in the execution trace. It also implies that the caller of the routine should be the

Algorithm 7 Procedure that handles a routine exit trace.

```
procedure HandleRoutineExitTrace($CG, cs, R$)
    $CG$ : Call graph
    $cs$ : The routine stack of a thread
    $R$ : The routine

    if $cs$ is not empty then
        if $Top(cs) = R$ then
            Pop(cs)
            if $cs$ is not empty then
                $AddEdge(CG, Top(cs), R)$
            end if
        else
            while $Top(cs) \neq R$ do
                $r \leftarrow Top(cs)$
                $Pop(cs)$
                if $cs$ is not empty then
                    $AddEdge(CG, Top(cs), r)$
                end if
            end while

            $Pop(cs)$
            if $cs$ is not empty then
                $AddEdge(CG, Top(cs), R)$
            end if
        end if
    end if

end procedure
```

stack top element once the current stack top is removed. EMAD therefore creates an edge between the two routines with the correct directionality (line 7-10) if an edge does not already exist. If an edge already exists, its weight is increased by 1.

When the routine object at the top of the stack and the routine object corresponds to routine exit message mismatches, it implies that the routine exit message for the routine object at the top of the stack is missing. The allocation object's caller should be current stack top's adjacent routine object. EMAD therefore saves the stack top, pops an element from the stack, and connects the new stack top with the previous stack top. EMAD continues this process until it finds the routine object represented by the current routine exit message. The sub-procedure for handling routine exit messages therefore guarantees that the correct caller-callee relationship is preserved even when routine exit messages are missing in the execution trace.

Once all messages in the execution trace are processed, there can still be routine objects remaining on the stack. EMAD explicitly calls the *HandleRoutineExitTrace* routine (line 20) while iterating through call stacks of each thread. This is necessary because the routine exit messages of the remaining routine objects are missing. Explicitly calling *HandleRoutineExitTrace* will complete the call graph with any missing edges.

As mentioned in Section 6.2.1, the start/exit messages for allocation/deallocation routines contain extra details such as parameter/return values and timestamps. Algorithm 5 and its sub-procedures discussed above will extract and store this additional information in allocation/deallocation routine objects during the execution trace processing. The data associated with the allocation/deallocation routines is used to create *allocation objects*. The allocation object has three attributes, the size of the dynamic memory allocation; the routine that calls the memory allocation routine to allocate memory; and the routine that calls the memory deallocation routine. In EMAD, each dynamic memory allocation during the lifetime of the application is represented using an allocation object.

An allocation object is distinguishable from another allocation object if any of its attributes is different. One would think it should be possible to use the address of the memory allocation to uniquely represent an allocation object. This, however, is not possible because the same memory address can be reallocated several times during the lifetime of the application. The memory address of an allocation is therefore not unique once we consider the entire lifetime of the application. EMAD therefore uses the memory address of an allocation to match the caller of the allocation routine and caller of the deallocation routine.

Each allocation object also has a frequency. The frequency specifies how many times an allocation object (with same values for above three attributes) occurs throughout the software application lifetime. For each allocation object, we can also calculate its lifetime as follows:

$$T_l = T_d - T_a \tag{6.1}$$

where T_l represents the lifetime of the allocation object; T_d represents the timestamp of the deallocation exit message; and T_a represents the timestamp of the allocation exit message. Each distinct allocation object stores its average lifetime. Lastly, EMAD uses the three attributes of an allocation object, its frequency, its calculated average lifetime, and the constructed call graph to detect the excessive dynamic memory allocation anti-pattern.

6.2.3 Detecting Excessive Dynamic Memory Allocations

As mentioned above our analysis technique for detecting excessive dynamic memory allocations is based on the intuition that this anti-pattern occurs when the software application has many short-lived high-frequent allocation objects. Our intuition comes from studying the two main solutions used to resolve the excessive dynamic memory allocation software performance anti-pattern [27].

The most common solution to resolve this anti-pattern is to use a custom memory allocator [85]. The basic idea of a custom memory allocator is to use a memory pool.

When using a memory pool, a large chunk of memory is allocated during the software application initialization phase. The subsequent requirements for memory allocations are fulfilled by obtaining memory from this memory pool—thereby eliminating the system calls to allocate memory. When the allocated memory is no longer needed, it is released into the memory pool—thereby eliminating the system calls to deallocate memory.

The custom memory allocations approach will not be effective if the allocation objects are in use for long periods of time. This is because when there are many such objects, eventually the memory pool will not be able to fulfill the allocation requests. This will result in acquiring memory from the operating system and the expected performance gain may not be achieved. When the software application has high-frequent short-lived allocation objects, however, the memory pool regains the memory it has given to the application. This improves the performance by rarely allocating memory using general purpose memory allocators.

The other solution for the excessive dynamic memory allocation anti-pattern is to use the Flyweight software design pattern [56]. The Flyweight software design pattern is similar to using a custom memory allocator. Its strategy also based on reusing the already allocated objects. The only difference is the Flyweight design pattern applies the solution at a higher level of abstractions such as reusing particular types of objects. It is also effective only when there are high-frequent short-lived object instances that are reusable.

Based on this intuition, EMAD's main goal in the detection process is to identify short-lived, high-frequent allocation objects. EMAD analyzes the frequency and average lifetime of the allocations objects annotated with the allocation/deallocation routines in the constructed call graph. To understand the analysis process, we introduce a frequency-lifetime diagram as illustrated in Figure 6.4. Each point in the diagram represents a unique allocation object. The x value represents the frequency of the allocation and y value represents the average lifetime of the allocation. We consider points that fall in the low-right quadrant to correspond to short-lived, high-frequent dynamic memory allocations. These are the set of points we want to identify in our analysis.

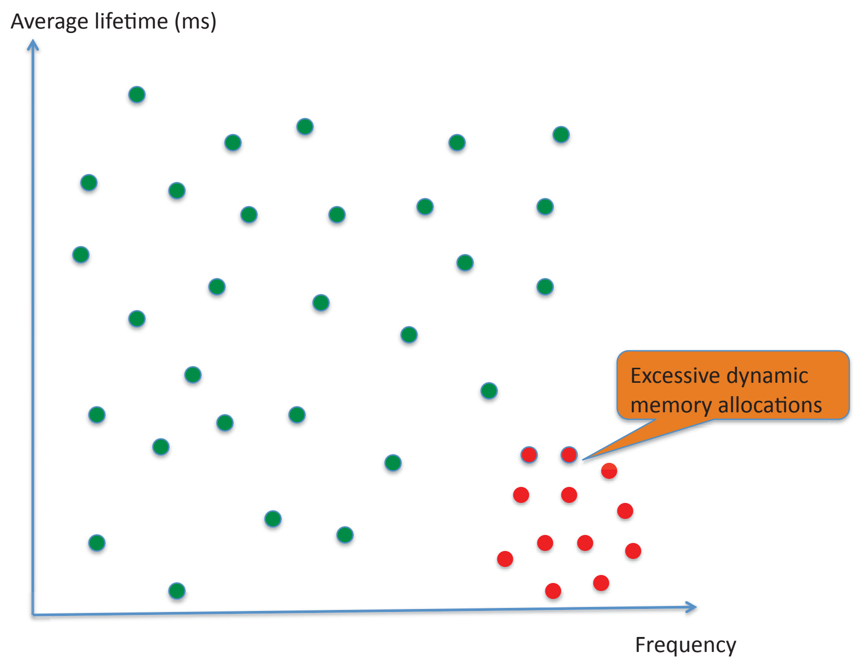

Figure 6.4.: Frequency-lifetime diagram.

Because frequency and lifetime of allocation objects are relative to each software application, it is hard to define thresholds to filter high-frequent, short-lived memory allocations. EMAD therefore provides two different exploratory data analysis techniques: one using K-means clustering, and the other using an outlier detection technique to identify high-frequent, short-lived memory allocations.

6.2.3.1 Using K-means Clustering to Identify High-frequent Short-lived Objects

Clustering is a non-supervised technique that can be used to partition objects based on the quantitative values of their attributes. The goal of clustering is to partition regions of points that have similarities. To accomplish this task, EMAD uses popular K-means algorithm [86] to cluster the allocation objects based on their frequency and average lifetime. Sidebar 2 contains a brief description on K-means clustering.

Sidebar 2: Overview of K-means Clustering

The problem of clustering is to partition n data points $x_i, i = 1 \dots n$ into k partitions. K-means approach in solving this problem is to find k centroids $\mu_i, i = 1 \dots k$ to represent each cluster such that the distance from the centroid to the data points is the minimum. K-means clustering solves,

$$arg \min_{c} \sum_{i=1}^{k} \sum_{x \epsilon c_i} d\left(x, \mu_i\right) \tag{6.2}$$

K-means clustering typically uses square of the Euclidean distance. Therefore above equation can be written as,

$$arg \min_{c} \sum_{i=1}^{k} \sum_{x \epsilon c_i} \left\| x - \mu_i \right\|^2 \tag{6.3}$$

This problem is a NP-hard problem therefore K-means algorithm does not guarantee a global minimum, however it works well in practice and widely used [87]. It uses the following steps to find the k clusters.

1. Initialize the centroid of the clusters
2. Assign the data points randomly to each cluster
3. Evaluate the centroids of each cluster by averaging the data points
4. For each data point calculate the distance to each cluster centroids and assign it to the one with minimum distance
5. Repeat step 3,4 until convergence

The algorithm stops when the assignment does not change from one iteration to the other.

Once the allocated objects are clustered, EMAD then checks whether there is a cluster C that satisfies all the following conditions:

1. The average frequency of C's members is the highest compared to the other clusters. This piece of information is important because if the frequency is high, then there is a potential excessive dynamic memory allocation issue.

2. The average lifetime of C's members is the lowest compared to the other clusters. This piece of information is important because when the allocation object is a short lived object there is a potential excessive dynamic memory allocation issue.

If EMAD can find a cluster that satisfies both the conditions above, then it reports that software application has excessive dynamic memory allocation anti-pattern. The report may contain all the members of that cluster, or a user-defined number n of members. In the latter case, EMAD will report first n members in the descending order of frequency. Because the allocation objects contains the caller information of the allocation, EMAD can also report call hierarchy of the allocation similar to other dynamic memory analysis tools (*e.g.*, Valgrind [44] and Purify [76]). By providing the call hierarchy software developers can quickly locate the excessive dynamic memory allocations anti-pattern in the source code—eliminating tedious and time consuming source code analysis.

On the other hand, if EMAD cannot find a cluster that satisfies the conditions above, then EMAD reports that the software application does not have the excessive dynamic memory allocations anti-pattern. This is because the partitioning indicates that most of the high-frequent allocation objects have a longer lifetime, or short lived allocation objects are not frequent.

Because EMAD's analysis is based on a clustering technique, the user can configure the parameter that controls the number of clusters. This parameter, in turn, controls the number of partitions EMAD has to create from the dataset. Unfortunately, this is one of the limitations in cluster analysis [88, 89]. Likewise, identifying the correct number of clusters may require some trial and error.

6.2.3.2 Using Outlier Detection to Identify High-frequent Short-lived Objects

In this technique, we convert the two dimensional dataset into a one dimensional dataset by calculating the ratio between frequency and average lifetime of each allocation object.

Therefore, the ratio R is defined as,

$$R = \frac{frequency}{lifetime} \tag{6.4}$$

According to the above equation, the value of R is larger when the frequency is high and lifetime is low. Therefore, we consider allocation objects that have relatively high values as potential excessive dynamic memory allocations. Based on this intuition we consider extreme outliers of this one dimensional dataset as potential excessive dynamic memory allocations. We only consider positive outliers that have larger values for R, not the outliers with lower values. To identify these extreme values we use Interquartile Range (IQR) based outlier detection technique [90]. We adopt this technique instead of standard score based outlier detection techniques because we observed that our datasets are not normal distributions [91]. We consider allocation objects that have a value greater than the value obtained from the following expression as potential dynamic excessive memory allocations.

$$Q_3 + \mu \times IQR \tag{6.5}$$

Here Q_3 is the third quartile, IQR is the Interquartile Range, and μ is a user provided parameter. If we increase the value of μ, EMAD may miss potential excessive dynamic memory allocations; and a lower value for μ may cause EMAD to report several false-positives. Therefore, the user has to provide a reasonable value for μ which may requires some trial and error. A good initial value for μ is the value obtained for IQR. Another way to decide on a value for μ is to first view the datasets and see how the value of R is deviating from normal. EMAD outputs this value during the analysis. EMAD also ranks the excessive dynamic memory allocations based on the value of R. Therefore, users can get an idea about the relative significance of excessive dynamic memory allocations after seeing the results. EMAD also provide facilities to view both two dimensional (*i.e.* frequency and lifetime) and one dimensional datasets (*i.e.*, value of R) of allocation objects.

6.3 Experimental Evaluation of EMAD

This section illustrates how we validate EMAD's methodology by applying it to several real world open source systems. Validating EMAD's technique is challenging because once EMAD reports excessive dynamic memory allocations, we need to make sure it is an actual excessive dynamic memory allocation that has an impact on system performance. Therefore, we validate EMAD with following types of experiments:

1. EMAD is applied to a known released software version that has the anti-pattern and then to an anti-pattern resolved newer software version. (see Section 6.3.2)
2. Applying EMAD to an anti-pattern induced software version to see whether EMAD can detect the induced anti-pattern (see Section 6.3.4)
3. EMAD detects the anti-pattern, which is previously unknown, we fix and validate with EMAD again to see whether the problem is actually resolved. (see Section 6.3.3)
4. Applying EMAD to software that does not exhibit the anti-pattern (see Section 6.3.5)

Further to strength our experimental analysis, we evaluate the system performance before and after resolving the anti-pattern if applicable for the above experimental scenarios.

6.3.1 Experimental Setup

Following is a brief description of each open source project we applied EMAD.

- **SQLite** is a widely used SQL database engine that is mainly used in embedded devices, such as mobile phones and web browsers. The power behind SQLite is that it allows developers to access a flat file as if it was a SQL database. We selected SQLite for our evaluation because we were able to search its release history and identify versions of SQLite that were actually impacted by the excessive dynamic memory allocation anti-pattern. This project therefore will serve as a good example to evaluate if EMAD can identify the software performance anti-pattern. More importantly, it will evaluate if EMAD is able to identify the routine that is the source of the problem. (*i.e.* experimental scenario 1)

- **Axis2-C** [92] is a web services framework, which is implemented in C using the popular Axis2 SOAP processing architecture [93]. Axis2-C is used in some of the modern cloud computing infrastructure middleware and also in scripting language based web services engines [94]. We have found a way to induce the dynamic memory allocations anti-pattern into Axis2-C by doing a slight modification to its source code. This allowed us to apply EMAD to Axis2-C to check whether EMAD can detect the induced anti-pattern. (*i.e.* experimental scenario 2)

- **TAO** [95] is a object request broker, which implements the OMG CORBA specification. TAO is used heavily to develop distributed real time and embedded systems. Because TAO is used in real time systems, small percentages of performance improvements matter in practice. Therefore, we have used it as one of our case studies. (experimental scenario 3)

- **Xerces-C++** [81] is a XML parser framework written in the C++ programming language. It can parse, generate, and validate XML documents using the DOM, SAX, and SAX2 APIs. It is one of the most widely used C++ XML parsers. We selected Xerces-C++ because the framework allows developers to integrate custom memory allocators. Moreover the custom memory allocator integrated version and the default version do not show much difference in performance. Therefore it is an indication that Xerces-C++ does not have the excessive dynamic memory allocations anti-pattern. (experimental scenario 4)

All experiments were conducted on an Intel core 2 Duo 3.33 GHz processor, with 4GB memory and running 32-bit Ubuntu 14.04 operating system. We also used Pin 2.13 and Pin++ 1.0.0-beta.

6.3.2 Experimental Results for SQLite

We used the Northwind database [96] as the use case for our experiments with SQLite. We used a single SQL file that had the SQL statements for table creation, data insertion, table updating, and data querying. The SQLite command line interface was used to manipulate all queries in the SQL file. Lastly, the performance of SQLite was measured by recording the total time it takes to process the entire Northwind database SQL file.

According to the SQLite [97] release history, SQLite had the excessive dynamic memory allocations software performance anti-pattern prior to version 3.6.1. This is because versions prior to 3.6.1 created many number of short-lived memory allocations in each database connection. The SQLite documentation states the following related to this excessive dynamic memory allocations problem[2]:

[2]More on the quote can be found at the following location: `www.sqlite.org/malloc.html#lookaside`

> These small memory allocations are used to hold things such as the names of tables and columns, parse tree nodes, individual query results values, and B-Tree cursor objects. There are consequently many calls to malloc() and free() - so many calls that malloc() and free() end up using a significant fraction of the CPU time assigned to SQLite.

As a solution to this issue, SQLite developers implemented a custom memory allocator called *lookaside allocator* that preallocates a large chunk of memory and divides it to fixed size small slots inside each database connection. This is called a *lookaside memory pool*.

We applied EMAD against the Northwind database while using SQLite 3.5.9. We did not use SQLite 3.6.0 because it was not a stable release.

6.3.2.1 Experimental Results with Clustering Method Enabled

We first used EMAD with the clustering method enabled as described in Section 6.2.3.1. From our experiments, EMAD was able to detect 3 locations where SQLite was performing excessive dynamic memory allocations in SQLite 3.5.9. These 3 locations are shown in Table 6.1.

Table 6.1.: Excessive dynamic memory allocation locations in SQLite-3.5.9 identified by EMAD from clustering method.

Caller	**Size**	**Destroyer**	**Freq.**	**Avg. Lifetime**
sqlite3DbMallocRaw	68	sqlite3ExprDelete	29394	929.758 ms
sqlite3DbMallocRaw	32	sqlite3VdbeMemRelease	12918	224.889
pager_write Size	1024	sqlite3BtreeCommitPhaseTwo	6832	1.29202

EMAD was also able to generate the call-tree (to a user-defined depth) for routines that are the source of excessive dynamic memory allocations. For example, Listing 6.5 illustrates the call-tree for the *sqlite3DbMallocRaw* routine. The call-tree shows the routine name and the frequency (inside parenthesis) of each caller-callee relationship. Although there are several call-trees for the *sqlite3DbMallocRaw* routine, Listing 6.5 only shows the call frequencies with maximum edge weights. The call tree for *pager_write* routine is shown in Lisitng B.1 of Appendix B.

```
sqlite3DbMallocRaw (45858)
sqlite3Expr Frequency (26765)
sqlite3Parser Frequency (26436)
sqlite3RunParser Frequency (102322)
sqlite3Prepare Frequency (3450)
sqlite3LockAndPrepare Frequency (3450)
sqlite3_prepare Frequency (3450)
sqlite3VdbeMemRelease (788186)
sqlite3BtreeNext Frequency (378600)
sqlite3VdbeExec Frequency (60800)
sqlite3_step Frequency (14098)
sqlite3_column_name (10659)
```

Listing 6.5: Call-tree for the routine *sqlite3DbMallocRaw*.

As described in the SQLite documentation routines like *sqlite3_column_name* contribute to excessive dynamic memory allocations in SQLite3. As shown in Listing 6.5, EMAD is able to report these routines in the call-tree for *sqlite3DbMallocRaw* routine as a cause (or source) of the excessive dynamic memory allocations.

Figure 6.5 shows the frequency-lifetime diagram for this experiment. This figure also supports the reported excessive dynamic memory allocations. As shown in Figure 6.5, the 3 allocation objects that correspond to excessive dynamic memory allocations have high-frequency (as high as 29394) and short lifetime (as low as 1.29202) when compared to the other allocation objects in the figure.

SQLite releases after version SQLite 3.5.9 implement the solution to the excessive dynamic memory allocations anti-pattern. To verify this, and continue evaluating EMAD, we applied EMAD to SQLite 3.8.5. In this version of SQLite, EMAD could find only one location where SQLite is performing excessive dynamic memory alloca-

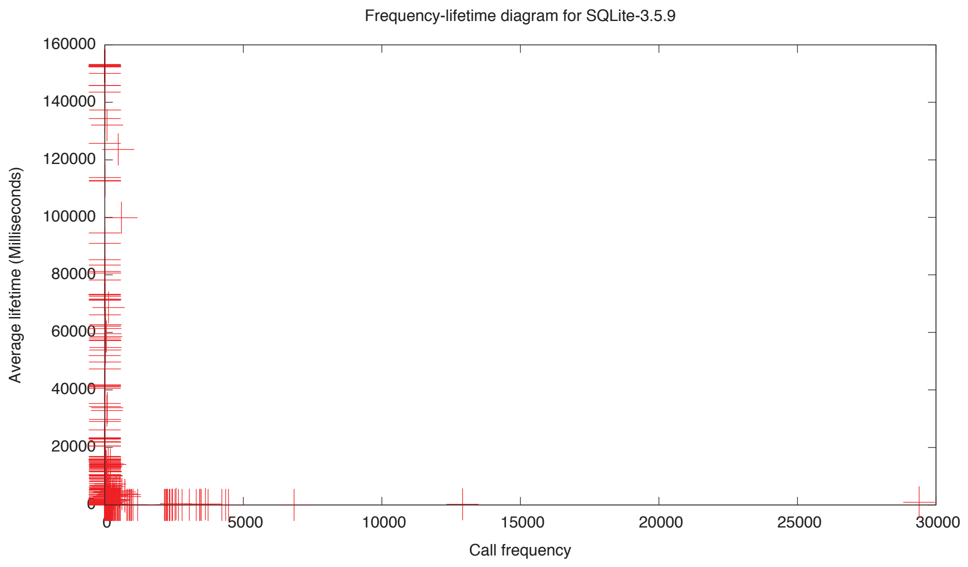

Figure 6.5.: Frequency-lifetime diagram for SQLite-3.5.9

tions. Table 6.2 shows the location identified by EMAD. This location is related to an input/output operation, which has no relation with the excessive dynamic memory allocation problem we found in SQLite 3.5.9.

Table 6.2.: Excessive dynamic memory allocation locations in SQLite-3.8.5 identified by EMAD from clustering method.

Caller	**Size**	**Destroyer**	**Freq**	**Avg. Life-time**
memjrnlWrite	1024	memjrnlTruncate	10596	1.403 ms

The frequency-lifetime digram shown in Figure 6.6 validates the results of EMAD. As shown in the diagram, there is only one allocation object that resides in high-frequency, short-lifetime region.

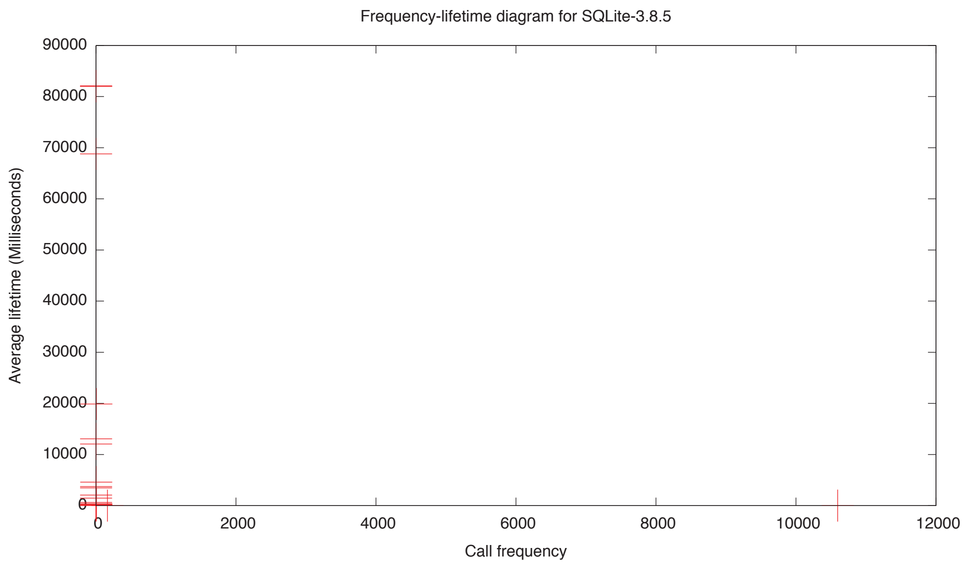

Figure 6.6.: Frequency-lifetime diagram for SQLite-3.8.5

6.3.2.2 Experimental Results with Outlier Detection Method

We also applied EMAD to SQLite 3.5.9 after configuring EMAD to employ outlier detection technique mentioned in Section 6.2.3.2. After using a value of 1000 for μ

in Equation 6.5, EMAD was able to report only one place as shown in Table 6.3 as excessive dynamic memory allocations. The frequency-lifetime ratio chart shown in Figure 6.7 also illustrate this extreme outlier.

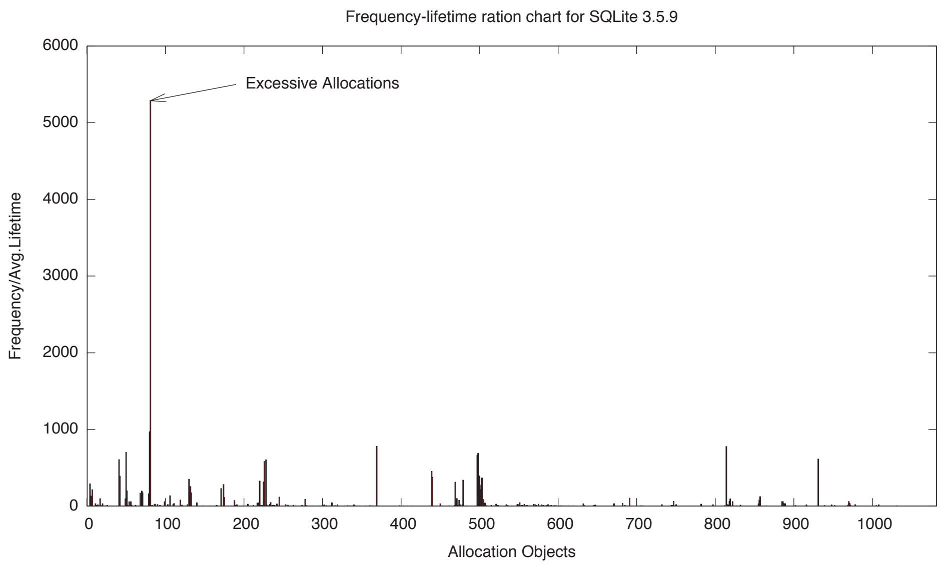

Figure 6.7.: Frequency-lifetime ratio chart for SQLite-3.5.9

Table 6.3.: Excessive dynamic memory allocation locations identified by EMAD in SQLite-3.5.9.

Caller	**Size**	**Destroyer**	**Freq.**	**Avg. Lifetime**
pager_write	1024	sqlite3BtreeCommitPhaseTwo	22832	1.29202 ms

As shown in the Table 6.3 the outlier detection technique didn't categorize some of the high frequency short-lifetime allocation objects as excessive dynamic memory allocations. This is because the IQR (Inter Quartile Range) of the dataset is as low as 1.7211 and we had to use a value as larger as 1000 for μ to filter the outliers. A lower value for μ started producing several false positives. For example, when we lowered the value of μ, EMAD reported allocation objects that have a frequency of 162 and an average lifetime of 0.2075 as excessive dynamic memory allocations. Although the average lifetime of the allocation objects is low in this case, the frequency is also low compared to the frequencies of excessive dynamic memory allocations.

We also applied EMAD after enabling the outlier detection technique to SQLite 3.8.5 version. EMAD reported the same location as shown in Table 6.2, which was identified from the clustering technique.

6.3.2.3 Resolving the Anti-pattern and Performance Improvements

To resolve the identified problem, we used a custom memory allocator (as mentioned in the SQLite documentation[3]) to resolve the performance anti-pattern and improve the performance. According to SQLite documentation, the custom memory allocator preallocates a chunk of memory during the application initialization process. To apply to solution, we re-compiled SQLite-3.8.5 with the custom memory allocator enabled. We then re-ran the same experiment with the enabled custom memory allocator. For our experiments, the custom memory allocator improved performance by 10%.

Table 6.4.: Performance of different versions of SQLite

SQLite Version	**Total Process Time**	**Malloc Calls**
3.5.9	475.01 ms	184859
3.8.5	338.43 ms	58441
3.8.5 with custom allocator	308.53 ms	9706

To summarize our performance results, Table 6.4 shows the total processing time for the Northwind database SQL file when processed by the three versions of SQLite we used in our experiment. As shown in the table, the performance of SQLite improved after we applied each solution to the identifed excessive dynamic memory allocation software performance anti-pattern. For example, SQLite 3.8.5 improved approximately 30% in performance when compared to SQLite 3.5.9. Likewise, SQLite 3.8.5 with custom memory allocator improved approximately 10% when compared to SQLite 3.8.5 without the custom memory allocation. More importantly, the experi-

[3]http://www.sqlite.org/malloc.html

ments shows that EMAD was able to detect the excessive dynamic memory allocations and can assist developers in improving performance.

Lastly, Table 6.4 shows the number of malloc/free routine calls invoked by each version of SQLite we used in our experiments. We collected this data using a Pintool that counts malloc/free routine calls. Our results show that when the excessive dynamic allocation anti-pattern is resolved, there are fewer system-level calls to the malloc/free routine.

6.3.3 Experimental Results for TAO

We instrumented TAO using EMAD's pintool, while sending 10,000 requests to its sample echo service. The collected trace was analyzed using EMAD for excessive dynamic memory allocations. EMAD reported two excessive dynamic memory allocations as shown in the Table 6.5.

Table 6.5.: Excessive dynamic memory allocation locations in TAO.

Caller	**Size**	**Destroyer**	**Freq.**	**Avg. Lifetime**
CORBA::string_alloc	14	CORBA::string_free	10000	0.453467 ms
operator¿¿	0	IOP::ServiceContextList:: ServiceContextList	9999	3.50659 ms

In this case both the clustering technique and the outlier detection technique reported the two locations shown in Table 6.5. The first excessive dynamic memory allocation in Table 6.5 is coming from TAO itself. The second one is coming from the echo service (*i.e.,* the application) when it is echoing the received string. The call tree for the excessive dynamic memory allocation in TAO is shown in Listing 6.6. The complete stack frames for this routine call tree is shown in Listing B.4 of Appendix B. The frequency-lifetime diagram in Figure 6.8 and the frequency-lifetime ratio chart in Figure 6.9 also confirmed EMAD's finding.

```
TAO_GIOP_Message_Gen_Parser_12::parse_header (10000)
TAO_GIOP_Message_Base::process_request (10000)
```

```
TAO_GIOP_Message_Base::process_request_message (10000)
TAO_Transport::process_parsed_messages (10000)
TAO_Transport::handle_input_parse_data (10000)
TAO_Transport::handle_input (10001)
TAO_Connection_Handler::handle_input_internal (10001)
TAO_Connection_Handler::handle_input_eh (10001)
TAO_IIOP_Connection_Handler::handle_input (10001)
ACE_TP_Reactor::dispatch_socket_event (10001)
```

Listing 6.6: Call-tree for the routine *operator¿¿*.

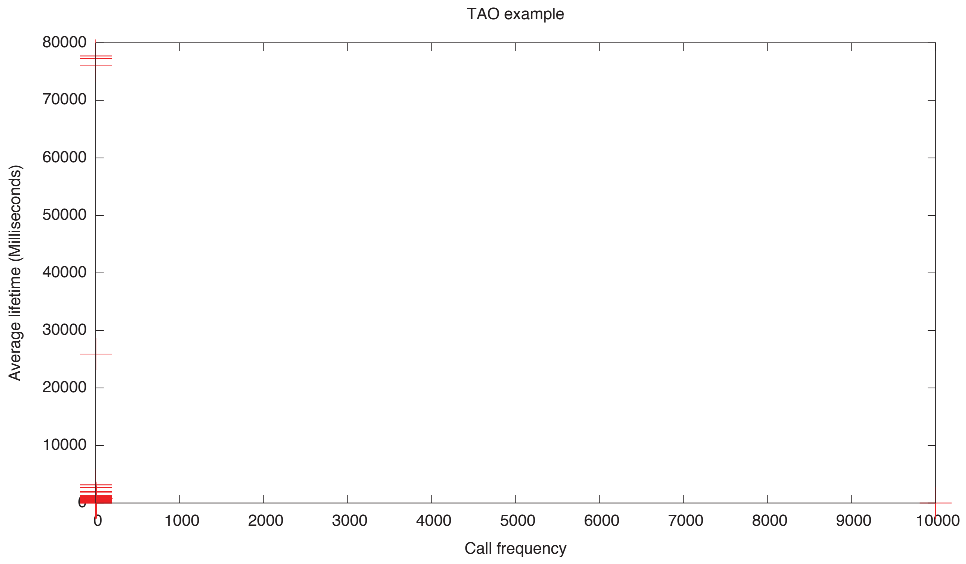

Figure 6.8.: Frequency-lifetime diagram for TAO

Apart from the two excessive dynamic memory allocations, almost all the other allocation objects have a very low frequency. Because of this, only the two data points that corresponds to the excessive dynamic memory allocations are visible in the Figure 6.9.

Our focus was on resolving the excessive dynamic memory allocation that resides in TAO. Because it will impact all the applications that use TAO. This excessive dynamic memory allocation occurs when TAO is doing a zero size allocation using

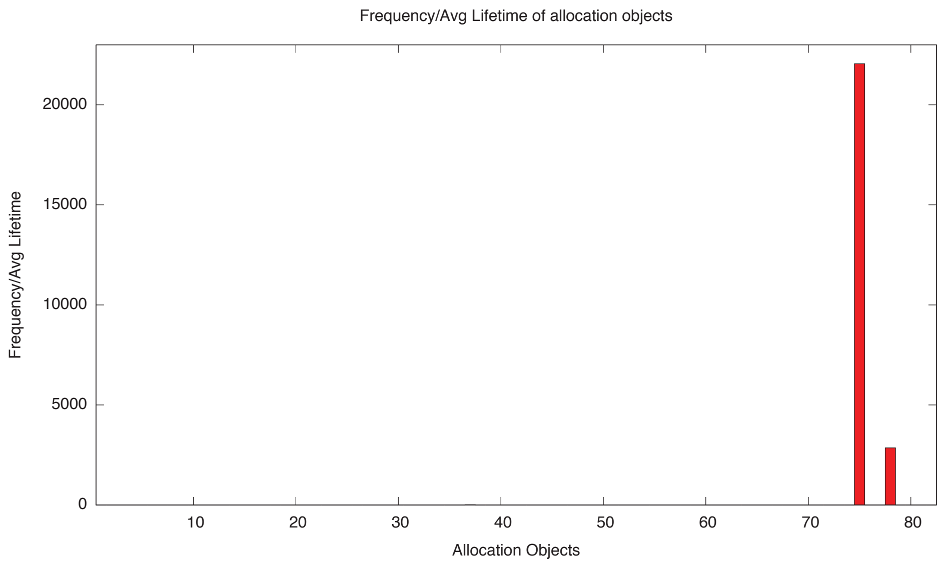

Figure 6.9.: Frequency-lifetime ratio chart for TAO

new[] operator to allocate a list of buffers to keep some service context information. When the same client sends many requests, however the buffer to keep service context information has to be allocated only for the first request. Instead of avoiding allocations for subsequent requests, TAO is doing a zero size allocation with new[] operator. Our simple fix was to return immediately before calling new[] when the requested length is 0.

After this fix we evaluated TAO performance for the simple echo service. We measured the time it takes to process n requests. We observed a 5-10% performance gain for larger number of requests. The performance results are shown in the Table 6.6.

Table 6.6.: Performance of echo service example in TAO.

# of Requests	Before Fix (sec)	After Fix (sec)	Improvement(%)
10K	2.275431	2.25299	0.98
20K	4.589058	4.491926	2.11
30K	6.972080	6.825455	2.1
40K	9.51474	9.419871	0.99
50K	11.487203	11.291216	1.7
100K	22.917998	22.587449	1.44
200K	52.195151	45.445869	12.93
300K	68.968680	63.624066	7.74
400K	91.914805	85.586583	6.88
500K	115.174436	106.963704	7.12

We reported this finding to the TAO mailing list. The TAO developers accepted the patch as it was something they were not aware of. Although it is not a bug, they are willing to fix the problem because even a small improvement in performance is valuable in the context of distributed realtime and embedded systems.

6.3.4 Experimental Results for Axis2-C

Axis2-C is executed via Apace Web Server [29] as an Apache web server module[4]. Axis2-C calls Apache web server's memory pool based dynamic memory allocation/deallocation routines (custom memory allocator) to allocate/deallocate dynamic memory during runtime. To induce the excessive dynamic memory allocations antipattern, we changed the source code of Apace web servers's Axis2-C module to use

[4]http://httpd.apache.org/docs/current/mod/

malloc/free instead of Apace memory pool functions to allocate/deallocate memory. This prevented Axis2-C from allocating memory from Apache's memory pools. After the change, we used Apache Benchmark tool to send 2000 SOAP requests to Axis2-C sample echo service, which is deployed in Apache web server. We then instrumented Apache web server and the Axis2-C while the requests were processed. The collected execution trace was then analyzed using EMAD for excessive dynamic memory allocations.

We found several locations where Axis2-C was doing excessive dynamic memory allocations. Both the frequency-lifetime diagram (Figure 6.10) and frequency-lifetime ratio chart (Figure 6.11) provide evidence for this.

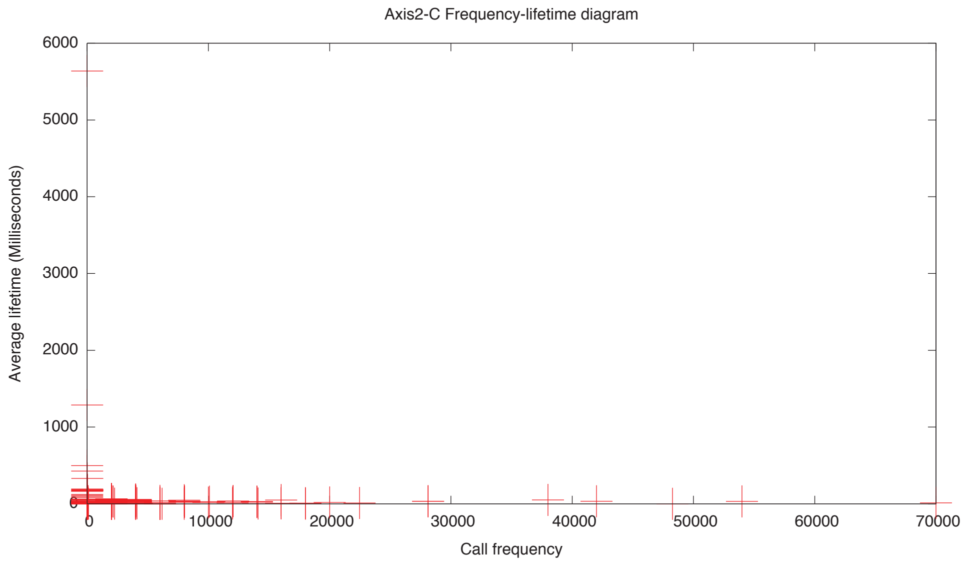

Figure 6.10.: Frequency-lifetime diagram for Axis2-C

Both clustering technique and the outlier detection techniques were able to find several locations that exhibit excessive dynamic memory allocations. These excessive dynamic memory allocations are listed in Table 6.7 and in Table 6.8.

We have shown only the first 5 excessive dynamic memory allocations based on a rank provided by each technique. When using the clustering technique, the results are ranked from highest frequency to lowest frequency in the excessive dynamic memory

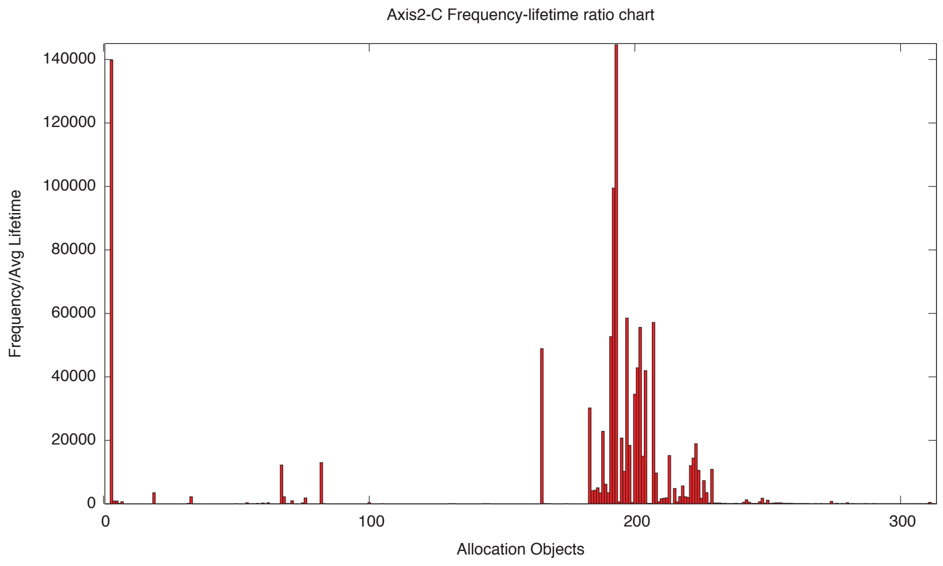

Figure 6.11.: Frequency-lifetime ratio chart for Axis2-C

Table 6.7.: Excessive dynamic memory allocation locations in Axis2-C identified by EMAD from clustering method.

Caller	Size	Destroyer	Freq	Avg. Life-time
axutil_string_create	16	axutil_string_free	70000	12.4372 ms
axiom_node_create	40	axiom_node_free_detached_subtree	54000	30.8595 ms
axutil_hash_first	16	axutil_hash_next	48274	0.345397 ms
axutil_hash_find_entry	20	axutil_hash_free	42000	32.9438 ms
axutil_string_create_assume_ownership	16	axutil_string_free	38000	49.6887

Table 6.8.: Excessive dynamic memory allocation locations in Axis2-C identified by EMAD from outlier detection method.

Caller	Size	Destroyer	Freq	Avg. Life-time
guththila_get_prefix	4	axiom_stax_builder_process_namespaces	8000	0.0553061 ms
axutil_hash_first	16	axutil_hash_next	48274	0.345397 ms
guththila_get_prefix	8	axiom_stax_builder_process_namespaces	6000	0.0603158 ms
axutil_strdup	5	axis2_req_uri_disp_find_op	2000	0.034171 ms
axutil_stracat	22	axutil_qname_to_string	4000	0.0699757

allocation cluster. In the outlier detection technique, the results are ranked based on the frequency and lifetime ratio with the allocation object with the highest ratio coming first. Some of the excessive dynamic memory allocations obtained a higher rank from both the techniques. For example, memory allocation has *axutil_hash_first* a higher rank on both the techniques.

The call tree found by EMAD for this allocation object (*i.e. axutil_hash_first*) is shown in Listing 6.7.

```
axiom_element_free (16000)
axiom_node_free_detached_subtree (36000)
axiom_node_free_tree (4000)
axiom_document_free (2000)
axiom_stax_builder_free (2000)
axiom_soap_builder_free (2000)
axiom_soap_envelope_free (2000)
axis2_msg_ctx_free (4000)
axis2_apache2_worker_process_request (4000)
axis2_handler (2000)
ap_run_handler (2000)
```

Listing 6.7: Call-tree for the routine *axutil_hash_first*.

The call trees for other excessive dynamic memory allocations are presented in Section B.3 of Appendix B. As shown in above listings, Axis2-C's excessive dynamic memory allocations happen mainly because of deep copying of strings. Because Axis2-C is a SOAP engine it performs heavy XML processing for each request. Therefore, it has to do frequent string manipulations. In the real world, Axis2-C is used with QoS support after engaging third-party developed QoS modules. Axis2-C has to pass some parts of the part of the SOAP message as XML objects to these third-party modules. Once these XML objects are passed to the third-party modules, it is hard to determine the ownership of strings. Therefore, Axis2-C uses a safe approach by deep copying the strings. This is the main reason for the excessive dynamic memory allocations. However, when used with Apace web server, Axis2-C

Table 6.9.: Axis2-C performance.

Item	**With memory pools**	**without memory pools**
Time takes to serve 1 million requests	280 secs	304 secs
Mallocs per one request	370	11032

can still perform deep copying when necessary without sacrificing the performance by leveraging Apace's memory pools. Although Apache has several type of memory pools such as request, connection and global, Axis2-C mostly uses the request pool because most of the object creation/deletion happen per request basis. When using Apache memory pools, Axis2-C has 8% of performance improvement for processing 1 million requests as shown in Table 6.9. The table also shows that 96% less calls to Malloc when processing a single request.

6.3.5 Experimental Results for Xerces-C++

We used Xerces-C++ Simple API for XML (SAX) interface to parse a 117 KB XML file that contained 1,318 elements and 71,166 characters via its SAX command-line utility. We then used EMAD to collect the execution trace of the SAX command-line utility while it processed the XML file. Next, we used EMAD to generate the call graph from the execution trace and detect the presence of the excessive dynamic memory allocation software performance anti-pattern.

EMAD could not find any excessive dynamic memory allocations in Xerces-C++ either from the clustering or the outlier detection techniques. We have also checked if a prior version of Xerces-C++ may have had the excessive dynamic memory allocation software performance anti-pattern. We, however, could not find any version after going through Xerces-C++ release notes.

Since Xerces-C++ supports custom memory allocators, we decided to investigate if we could improve Xerces-C++ performance by implementing a custom memory allocator. By default, Xerces-C++ uses the new/delete operators to allocate/deallocate memory. Our custom memory allocator is an implementation of a free list. At the beginning, it allocates a large chunk of memory that is partitioned into small user defined chunks. These small chunks are maintained as two linked lists. The first linked list maintains the memory chunks that are being used in the program. The second linked list maintains the freely available memory chunks.

The allocation function simply returns a memory chunk from the free list and creates a pointer to that chunk from allocated list. The deallocation function simply gives back the deallocated memory chunk to the free list and removes the corresponding pointer from the allocated list. This reduced frequent calls to general-purpose memory allocation/deallocation routines (*i.e.*, new/delete). Lastly, the memory pool calls the general-purpose memory allocation functions if the allocated memory pool is not large enough to service the user request.

Finally, we measured the overall processing time for the XML file using the default memory allocator and the custom memory allocator. Table 6.10 shows the results of this experiment.

Table 6.10.: Performance of Xerces-C++ with a custom memory allocator and default memory allocator.

Xerces-C++ Method	**Avg. Process Time**
w/ default memory allocator	159 ms
w/ custom memory allocator	155 ms

As presented in Table 6.10, even when we plugged in the custom memory allocator we could not observe much performance gain (as small as 2.5%). This is an indication that Xerces-C++ does not exhibit excessive dynamic memory allocations. Figure 6.12 shows the frequency-lifetime diagram for our experiments.

In the diagram, none of the allocation objects resides in the high-frequent, short-lifetime region of the graph. EMAD therefore does not report any excessive dynamic memory allocations. The frequency-lifetime ratio chart shown in Figure 6.13 also confirms this finding. The range of values for frequency-lifetime ratio is as low as 16,

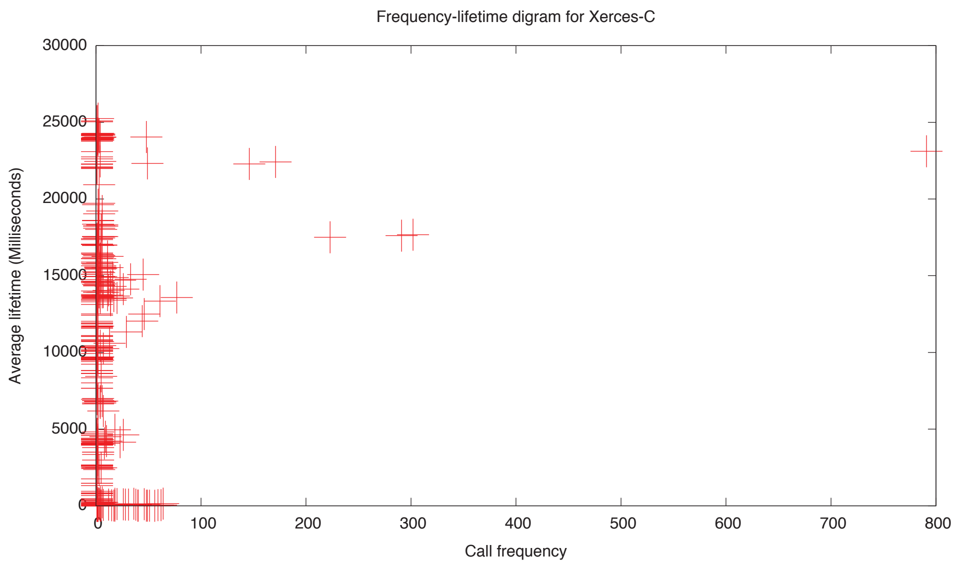

Figure 6.12.: Frequency-lifetime ratio chart for Xerces-C

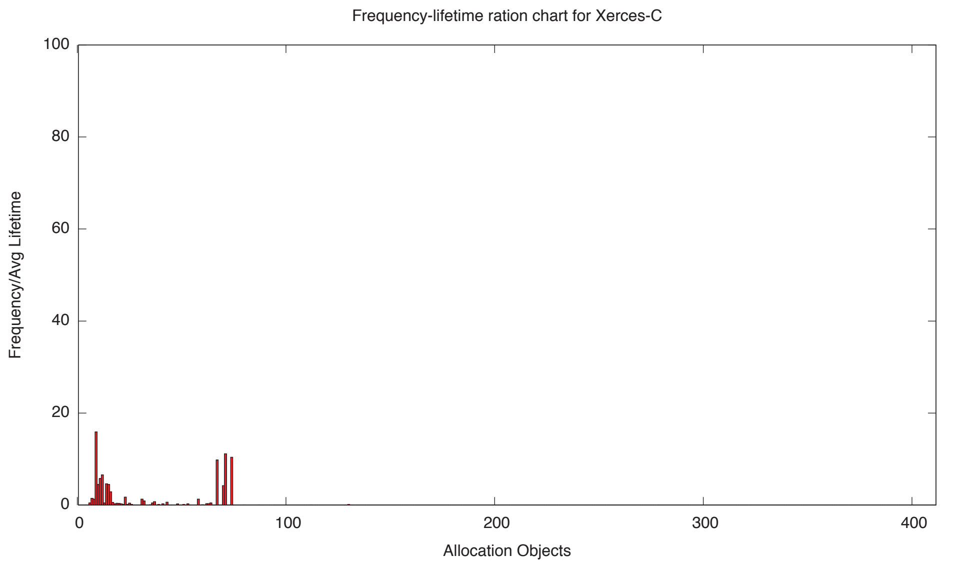

Figure 6.13.: Frequency-lifetime ratio chart for Axis2-C

however in other applications where we found excessive dynamic memory allocations, this ratio has a range as high as $150,000$.

6.3.6 Discussion of Results and Threat to Validity

These experiments show the validity of EMAD's overall approach for detecting excessive dynamic memory allocations. It was able to correctly detect and locate when a software application has the anti-pattern, and when it does not have the anti-pattern. This kind of analysis will help software developers resolve excessive dynamic memory allocations faster. More importantly, it will eliminate the laborious process of detecting the anti-pattern via manual source code analysis.

The main advantage of the clustering technique over the outlier detection method is that it does not categorize allocation objects as excessive dynamic memory allocations when they have a low frequency. In the outlier detection technique, because we consider frequency-lifetime ratio as the analytical value, it can still report extreme outliers when the frequency is low and lifetime of the allocation object is very short. These kind of low-frequent and short-lifetime values may sometime beat some high-frequent short-lifetime objects. However, with the clustering technique, this kind of false positives is not possible as it reports the problem only if it can find the high-frequent short-lifetime cluster.

However, when using the clustering technique, EMAD's users have to provide the number of clusters to use in the analysis phase. There are some advanced data mining techniques [89, 98] for learning this parameter from the dataset itself. EMAD, however, does not employ those techniques in its current implementation. Unfortunately, this can cause EMAD to provide incorrect predictions if the user does not specify a reasonable number of clusters. If the dataset has very clear separable partitions, then the impact of this parameter can still be mitigated. On the other hand, when using outlier detection technique, users have to provide the parameter μ, which may also need some trial and error.

When using clustering technique, EMAD performs quantitative analysis and detects excessive dynamic memory allocations only if high-frequent and short-lived allocation objects resides in the same cluster. A software developer, however, may still think that there are excessive dynamic memory allocations in other clusters by looking at the numbers. In this situation, EMAD's prediction may not be inline with software developer's expectation. EMAD, however, can still be helpful because the software developer can manually analyze the frequency-lifetime diagram or the frequency-lifetime ratio chart to understand the big picture. A recommended way for further analysis is to do a comparative analysis of both two dimensional and one dimensional datasets.

6.4 Summary of Contributions

In this chapter, we have presented the Excessive Dynamic Memory Allocations Detector (EMAD), which is a tool and technique to detect excessive dynamic memory allocations software performance anti-pattern using system execution traces. The following are the key contributions of EMAD.

- An algorithm to construct a dynamic call graph of a program using an execution trace, which may be missing routine exit messages corresponds to tail calls.
- Applying the K-means clustering algorithm [86] and an outlier detection techniques to the data collected from DBI to detect excessive dynamic memory allocations anti-pattern;
- First attempt, to the best of the authors knowledge, of a tool that can automatically detect the excessive dynamic memory allocations software performance anti-pattern; and
- First attempt, to the best of the authors knowledge, of using DBI to detect a software performance anti-pattern.

7 CONCLUDING REMARKS

In this dissertation, we have described generalized techniques to support software performance analysis using system execution traces. We preseneted three novel contributions, which extend current state of the art of using system execution traces for software performance analysis. We first presented the System Execution Trace Adaptation Framework (SETAF). SETAF enables software testers to write adaptation specification and then provides a framework to use those adaptation specifications with our software performance analysis tool called UNITE. We have also described in detail why such an adaption is required. Second, we described DMAC (Dataflow Model Auto Constructor), which is a tool and a technique to auto-construct dataflow models from system execution traces. DMAC uses an iterative frequent sequence mining technique to identify abstract event types from system execution traces and uses an evidence theory based algorithm to detect causality relationships among abstract event types. We then described EMAD an approach for detecting dynamic excessive memory allocation software performance anti-pattern using execution traces generated from dynamic binary instrumentation. We have shown the applicability of SETAF,DMAC and EMAD by describing the results of applying them to several real world open source software projects.

The presented techniques do not require source code of the system to be available and are not tightly coupled to particular system architectures. Moreover, the proposed techniques use non-intrusive system execution traces. Software developers or testers do not need to modify the source code or the binary artifacts of the systems. The following is a summary of lessons learned from the research work presented in this dissertation and some future research directions.

- Complex software systems, such as distributed systems can easily generate system execution traces that are quite large. Therefore analyzing them manually to create adaptation specifications is hard. Although DMAC provides a way to auto-construct dataflow model, adaptation specifications in SETAF are created manually. Therefore, techniques need to be developed to assist in locating adaptation patterns from execution traces and generating adaptation specifications.

- Adaptation specification size does not have much impact on evaluation time of performance analysis. As learned from the performance comparisons of the compiled and interpreted adapters in SETAF, total evaluation time depends on the number of log messages and the size of the dataflow model (*i.e.*, processing time).

- As stated above, system testers have to manually analyze the dataflow models and write the SETAF adaptation specifications. The performance analysis results from UNITE and SETAF therefore cannot be guaranteed if testers do not analyze the model correctly and write a specification correctly. Although this is true, the focus of SETAF is to provide the framework for writing adaptation specifications, and support UNITE when analyzing system execution trace.

- *Min-sup* value plays an important role in log format mining process, when auto-constructing dataflow models. Our log format mining process is based on the user provided *min-sup* value. New parameterless sequence mining approaches have been proposed by the data mining research community [99,100]. Therefore, future research can focus on using these new algorithms to improve DMAC's log format mining process.

- Domain-knowledge is important when constructing dataflow models. We observed that causality mining should not be dependent on only time and variable evidences. This is because the absence or presence of either evidence does not necessarily imply two log formats are—or are not—causally related. Time and variable evidences only increases/decreases confidence levels about the causal relationships. Future work therefore includes developing different techniques for interpreting the user-level domain-specific evidence into DMAC-level domain-specific evidence so it can be better integrated into the dataflow model auto-construction process.

- When auto-constructing dataflow models stopping criteria for the iterative mining process is important. Because DMAC finds more log formats as the number of iterations increases. This, however, may lead to incorrectly identifying variable parts as static parts. In some cases, the *min-sup* value does not decrease during sub-sequent iterations, which is a good indicator of coverage for the auto-generated dataflow model. Future work therefore includes investigating improved techniques for stopping the iterative mining process.

- One assumption DMAC has about the system execution trace is that there are no circular dependencies between log formats. This, however, is not always possible with some of the system execution traces. A circular dependency in the system execution trace may capture very important information such as deadlocks of the system execution. Future research therefore will investigate applying DMAC to system execution traces that have circular dependencies between its log formats.

- In EMAD, our analysis is based on data collected using DBI, and DBI can be used to collect lots of useful information related to an executing program at different granularities. For example, DBI frameworks like Pin allow us to gather information related to routine instructions and program locks. The also allow us to replace routine calls at runtime. This information and functionality can then be used to analyze behavioral aspects of software performance anti-patterns at runtime. As future work, we are planning to continue using DBI to detect other software performance anti-patterns [27], such as God Class, Single Lane Bridge, and Circuitous Treasure Hunt. We believe this approach will improve current state of the art in detecting software performance anti-patterns.

- Although K-means clustering has yielded positive results in EMAD experiments, we have not studied extensively the role clustering algorithms in relation to this problem. Moreover, there are other clustering algorithms [101] that perform better than the K-means algorithm when analyzing noisy data. Future research therefore should investigate comparing the results of different clustering algorithms in the context of the excessive dynamic memory allocations problem.

- EMAD's current technique works only with C/C++ software applications. Programs written in interpreted languages like Java, PHP, PERL have different memory management schemes when compared to C/C++. Moreover, these languages are typically run on virtual machines, and not directly on hardware like C/C++ software applications. Future research therefore will focus on applying EMAD to software applications written from interpreted programming languages to investigate if it is possible to uncover any evidence of excessive dynamic memory allocations.

- As the execution traces become larger both EMAD's and DMAC's analysis times grow from minutes to hours. We experienced this specially when building the intermediate models, such as dataflow models and call graphs from the execution traces. Therefore, it is important to parallelize the algorithms proposed in DMAC and EMAD techniques to speed up the analytical process.

The algorithms, analytics, and techniques described in this dissertation are available in open-source format. SETAF and DMAC has been integrated into CUTS distribution, which can be downloaded from `github.com/SEDS/CUTS`. EMAD has been integrated into the Pin++ distribution, which can be downloaded from `github.com/SEDS/PinPP`.

REFERENCES

REFERENCES

[1] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[2] M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[3] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMp applications. *Journal of Systems Architecture*, 49(10):421–439, 2003.

[4] B. Wylie, F. Wolf, B. Mohr, and M. Geimer. Integrated runtime measurement summarization and selective event tracing for scalable parallel execution performance diagnosis. *Applied Parallel Computing: State of the Art in Scientific Computing*, pages 460–469, 2007.

[5] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200, ACM, 2005.

[6] Reima Piilil Erkki Salonen. Find the bug, Fix the bug, Do it fewer times (TimeToPic). http://www.timetopic.net/Pages/default.aspx, 2012.

[7] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Symposium on Networked Systems Design and Implementation. USENIX Association*, pages 26–26, 2012.

[8] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 145–155, IEEE Press, 2012.

[9] James H. Hill, Hamilton A. Turner, James R. Edmondson, and Douglas C. Schmidt. Unit Testing Non-functional Concerns of Component-based Distributed Systems. In *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*, pages 406–415, Denver, Colorado, apr 2009.

[10] James H. Hill. Context-based Analysis of System Execution Traces for Validating Distributed Real-time and Embedded System Quality-of-Service Properties. In *Proceedings of 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 92–101, Macau, P.R.C., August 2010.

[11] A. Mos, J. Murphy, et al. Performance monitoring of Java component-oriented distributed applications. In *Proceedings of 9th IEEE International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 9–12, 2001.

[12] D. Mania, J. Murphy, and J. McManis. Developing performance models from non-intrusive monitoring traces. In *Proceedings of Information Technology and Telecommunications (IT&T)*, 2002.

[13] T. Parsons, A. Mos, and J. Murphy. Non-intrusive end-to-end runtime path tracing for J2EE systems. In *Proceedings of IEEE Software*, 153(4):149, 2006.

[14] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Mining Console Logs for Large-scale System Problem Detection. In *Proceedings of the 3rd conference on Tackling Computer Systems Problems with Machine Learning Techniques*, SysML'08, pages 4–4, Berkeley, CA, USA, 2008.

[15] Meiyappan Nagappan, Kesheng Wu, and Mladen A. Vouk. Efficiently extracting operational profiles from execution logs using suffix arrays. In *Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*, ISSRE'09, pages 41–50, 2009.

[16] Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. Analysis of a Very Large Web Search Engine Query Log. *Special Interest Group on Information Retrieval (SIGIR) Forum*, 33:6–12, September 1999.

[17] Y. Yin, S. Byna, H. Song, X.H. Sun, and R. Thakur. Boosting Application-Specific Parallel I/O Optimization Using IOSIG. In *Symposium on 12th IEEE/ACM Cluster, Cloud and Grid Computing (CCGrid)*, pages 196–203, 2012.

[18] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

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

[20] Haroon Malik, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *Proceedings of the 14th IEEE European Conference on Software Maintenance and Reengineering (CSMR)*, pages 222–231, 2010.

[21] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the IEEE International Conference on Software Engineering*, pages 1012–1021. IEEE Press, 2013.

[22] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 125–134, 2009.

[23] Manjula Peiris, James H. Hill, Jorgen Thelin, Sergey Bykov, Gabriel Kliot, and Christian Konig. PAD: Performance anomaly detection in multi-server distributed systems. In *Proceedings of the 7th IEEE International Conference on Cloud Computing*, Alaska, USA, June 2014.

[24] Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani. Performance antipatterns as logical predicates. In *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 146–156, IEEE, 2010.

[25] Jing Xu. Rule-based automatic software performance diagnosis and improvement. *Performance Evaluation*, 67(8):585–611, 2010.

[26] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software: Practice and Experience*, 24(6):527–542, 1994.

[27] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. In *Workshop on Software and Performance*, pages 127–136, 2000.

[28] Connie U. Smith and Lloyd G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, pages 717–725, 2003.

[29] Apache Software Foundation. The Apache HTTP Server Project. `http://httpd.apache.org/`.

[30] Free Software Foundation. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`.

[31] PHP Group. PHP. `http://php.net/`.

[32] Harvey W Gunther. Websphere application server development best practices for performance and scalability. *IBM WebSphere Application Server Standard and Advanced Editions–White paper*, 2000.

[33] Jafri Din, Anas Bassam Al-Badareen, and Yusmadi Yah Jusoh. Antipatterns detection approaches in object-oriented design: A literature review. In *7th International Conference on Computing and Convergence Technology (ICCCT)*, pages 926–931, IEEE, 2012.

[34] Catia Trubiani and Anne Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 19–30, 2011.

[35] Catia Trubiani and Anne Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 19–30, 2011.

[36] M. Cinque, D. Cotroneo, and A. Pecchia. A logging approach for effective dependability evaluation of complex systems. In *Proceedings of the 2nd International Conference on Dependability (DEPEND)*, pages 105–110, Washington, DC, USA, 2009. IEEE Computer Society.

[37] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software Engineering (ICSE)*, pages 282–291, New York, NY, USA, 2006. ACM.

[38] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *IEEE International Conference on Data Mining*, pages 149–158, 2009.

[39] Meiyappan Nagappan and Mladen A. Vouk. Abstracting log lines to log event types for mining software system logs. In *Proceedings of the 7th International Working Conference on Mining Software Repositories (MSR 2010 Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010*, pages 114–117, 2010.

[40] Hossein Safyallah and Kamran Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 84–88, 2006.

[41] J.G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. *ACM Special Interest Group on Operating Systems (SIGOPS) Operating Systems Review*, 44(1):91–96, 2010.

[42] M. Fischer, J. Oberleitner, H. Gall, and T Gschwind. System evolution tracking through execution trace analysis. In *13th International Workshop on Program Comprehension (IWPC)*, pages 237–246, May 2005.

[43] S. Voigt, J. Bohnet, and J. Dollner. Object aware execution trace exploration. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 201–210, September 2009.

[44] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[45] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer, 2004.

[46] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 269–280. IEEE Computer Society, 2004.

[47] Gene Novark, Emery D Berger, and Benjamin G Zorn. Efficiently and precisely locating memory leaks and bloat. In *ACM Sigplan Notices*, volume 44, pages 397–407. ACM, 2009.

[48] Xi Chen, Asia Slowinska, and Herbert Bos. Who allocated my memory? detecting custom memory allocators in C binaries. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 22–31. IEEE, 2013.

[49] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 307–320. USENIX Association, 2012.

[50] Aravind Menon, Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International conference on Virtual execution environments*, pages 13–23. ACM, 2005.

[51] Matthew Hertz, Stephen M Blackburn, J Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. Generating object lifetime traces with merlin. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):476–516, 2006.

[52] Guoqing Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *ACM SIGPLAN Notices*, volume 48, pages 111–130. ACM, 2013.

[53] Nathan P Ricci, Samuel Z Guyer, and J Eliot B. Moss. Elephant tracks: Portable production of complete and precise GC traces. *ACM SIGPLAN Notices*, 48(11):109–118, 2013.

[54] Trevor Parsons. A framework for detecting performance design and deployment antipatterns in component based enterprise systems. In *Proceedings of the 2nd International Doctoral Symposium on Middleware*, pages 1–5. ACM, 2005.

[55] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19:137–, March 1976.

[56] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[57] Sun Micro Systems. *Java Servlet Specification* . SUN, 3.0 edition, December 2009.

[58] Sun Micro Systems. *Java Server Pages Specification* . SUN, Version 2.1 edition, May 2006.

[59] SUN Micro Systems. Java Messaging Service Specification. `java.sun.com/products/jms/`, 2002.

[60] David Chappell. *Enterprise Service Bus*. OReilly, 2004.

[61] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, pages 67–82, Grenoble, France, November 2005.

[62] Object Management Group. *Deployment and Configuration Adopted Submission*. Object Management Group, OMG Document mars/03-05-08 edition, July 2003.

[63] Douglas C. Schmidt. The Adaptive Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[64] Douglas C. Schmidt. The Adaptive Communication Environment: An object-oriented network programming toolkit for developing communication software. In *CiteSeer*, pages 214–225, 1993.

[65] Manjula Peiris and James H. Hill. Auto-constructing dataflow models from system execution traces. *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, June 2013.

[66] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, pages 3–17, 1996.

[67] Mohammed J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning Journal*, 42(1/2):31–60, Jan/Feb 2001.

[68] P. Tzvetkov, X. Yan, and J. Han. TSP: Mining top-k closed sequential patterns. *Knowledge and Information Systems*, 7(4):438–457, 2005.

[69] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:558–565, July 1978.

[70] L.J. Mazlack. Causality recognition for data mining in an inherently ill defined world. In *International Joint Workshop on Soft Computing for Internet and Bioinformatics*, 2003.

[71] G. Shafer. *A mathematical theory of evidence*, volume 76. Princeton university press Princeton, 1976.

[72] T. Manjula Peiris and James H. Hill. Adapting System Execution Traces for Validation of Distributed System QoS Properties. In *Proceedings of 15th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, pages 162–171, Shenzhen, China, April 2012.

[73] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Oopsla 2002: Reconsidering custom memory allocation. *ACM SIGPLAN Notices*, 48(4):46–57, 2013.

[74] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[75] Naouel Moha. Detection and correction of design defects in object-oriented designs. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 949–950. ACM, 2007.

[76] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proceedings of the Winter 1992 USENIX Conference*. Citeseer, 1991.

[77] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *SIGPLAN Not.*, pages 89–100, June 2007.

[78] James H Hill and Dennis C Feiock. Pin++: An object-oriented framework for writing pintools. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, pages 133–141. ACM, 2014.

[79] ISO/IEC. International Standard: Programming Languages - C++. Number 14882:1998(E) in ASC X3, September 1998.

[80] ANSI/ISO. ANSI/ISO Standard C. Number ISO/IEC 9899:2011, April 2011.

[81] Apache Software Foundation. Xerces-C++ XML Parser. `http://xerces.apache.org/xerces-c/`.

[82] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.

[83] Frank Eichinger, Klemens Böhm, and Matthias Huber. Mining edge-weighted call graphs to localise software bugs. In *Machine Learning and Knowledge Discovery in Databases*, pages 333–348. Springer, 2008.

[84] Intel Corporation. Pin 2.14 User Guide. `https://software.intel.com/sites/landingpage/pintool/docs/67254/Pin/html/`.

[85] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *ACM SIGPLAN Notices*, volume 36, pages 114–124. ACM, 2001.

[86] John A. Hartigan and Manchek A. Wong. Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.

[87] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[88] Chris Fraley and Adrian E. Raftery. How many clusters? which clustering method? answers via model-based cluster analysis. *The computer journal*, 41(8):578–588, 1998.

[89] Catherine A. Sugar and Gareth M. James. Finding the number of clusters in a dataset. *Journal of the American Statistical Association*, 98(463), 2003.

[90] Victoria J. Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[91] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.

[92] Apache Software Foundation. Apache Axis2/C. `http://axis.apache.org/axis2/c/core/index.html`.

[93] Srinath Perera, Chathura Herath, Jaliya Ekanayake, Eran Chinthaka, Ajith Ranabahu, Deepal Jayasinghe, Sanjiva Weerawarana, and Glen Daniels. Axis2, middleware for next generation web services. In *International Conference on Web Services, 2006 (ICWS'06).*, pages 833–840. IEEE, 2006.

[94] Muhammad Imran and Helmut Hlavacs. Provenance in the cloud: Why and how. In *3rd International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 106–112, 2012.

[95] Douglas C. Schmidt, Bala Natarajan, Aniruddha Gokhale, Nanbor Wang, and Christopher Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[96] Microsoft Cooperation. Northwind database. `https://northwinddatabase.codeplex.com/`.

[97] SQLite. Release History. `http://www.sqlite.org/changes.html`.

[98] Stan Salvador and Philip Chan. Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In *16th IEEE International Conference on Tools with Artificial Intelligence, 2004 (ICTAI 2004)*, pages 576–584. IEEE, 2004.

[99] Jiawei Han, Jianyong Wang, Ying Lu, and Petre Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proceedings of the IEEE International Conference on Data Mining 2002 (ICDM 2002)*, pages 211–218. IEEE, 2002.

[100] Maged El-Sayed, Carolina Ruiz, and Elke A. Rundensteiner. Fs-miner: Efficient and incremental mining of frequent sequence patterns in web logs. In *Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 128–135. ACM, 2004.

[101] Michael Steinbach, George Karypis, Vipin Kumar, et al. A comparison of document clustering techniques. In *Knowledge Discovery and Data mining (KDD) workshop on text mining*, volume 400, pages 525–526. Boston, 2000.

[102] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[103] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[104] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124. ACM, 2010.

[105] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 241–252. ACM, 2010.

[106] Microsoft. Available Chart Types. `http://office.microsoft.com/en-us/excel-help/available-chart-types-HA010342187.aspx`.

[107] Richard C. Sprinthall and Stephen T. Fisk. *Basic statistical analysis*. Prentice Hall Englewood Cliffs, NJ, 1990.

[108] David C. Hoaglin, Frederick Mosteller, and John Wilder Tukey. *Understanding robust and exploratory data analysis*, volume 3. Wiley New York, 1983.

[109] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 100(8):933–940, 1987.

[110] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, 1992.

[111] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 149–160. ACM, 2001.

[112] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1012–1021. IEEE Press, 2013.

[113] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[114] C.U. Smith and L.G. Williams. New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. In *(Computer Measurement Group (CMG)*, volume 2, pages 667–674, Dallas, TX, 2003.

[115] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[116] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301. ACM, 2002.

APPENDICES

APPENDIX A PAD: PERFORMANCE ANOMALY DETECTION IN MULTI-SERVER DISTRIBUTED SYSTEMS

This appendix presents our work on analyzing performance counter data in system execution logs to detect performance anomalies in multi-server distributed systems. Performance anomalies in multi-server distributed systems can be in the form of under-achievements of performance goals such as low throughput or high latency. In such situations, system execution logs might not contain direct clues (*e.g.*, error messages or exceptions) that can be used as a starting point for analysis. Instead, they usually include performance counters that track different aspects of system performance.

Developers and testers typically analyze the performance counters to find system performance anomalies and reason about performance characteristics. Multi-server distributed systems, however, contain hundreds of servers each constantly generating performance data—making manual analysis error prone and time consuming. It is therefore essential to develop techniques and build automatic tools for performance analysis and diagnostics of large multi-server distributed systems using the performance data generated during execution.

Therefore, to diagnosis performance anomalies in distributed multi-server systems, we have developed a tool called *Performance Anomaly Detector (PAD)*. The objectives of PAD are: (1) give distributed system developers insights about distributed system performance from collected performance data; (2) minimize developer time required to analyze large amounts of performance data generated across hundreds to thousands of servers; and (3) assist system developers and administrators in troubleshooting performance related issues and finding root causes. To achieve the above goals PAD provides:

1. Summary of distributed system performance data using *visualizations* and *summary statistics*;
2. *Threshold analysis* for performance counters;
3. *Correlation analysis* for automatic detection of relationships between performance counters; and
4. *Comparative analysis* for automatic detection of anomalous performance counters.

The capabilities listed above enable a powerful combination of user-driven navigation analysis and automatic analysis. In user-driven navigation analysis, the person troubleshooting the system applies expert knowledge in a semi-manual process assisted by the tool. When this process does not lead to successful problem resolution, automatic correlation and comparative analysis techniques are used to automatically try to find clues for performance problems.

A.1 Motivation : The Orleans Cloud Computing System

The motivation for developing PAD started with our experience diagnosing the performance of the *Orleans system* [102]. Orleans is a programming model and runtime for large-scale distributed cloud computing services. Orleans is based on an *actor programming model*. Actors in Orleans are virtual and isolated computation entities that use asynchronous message passing to communicate. The actor model is suitable for interactive request-reply applications (as opposite to MapReduce [103] style models that are suitable for offline batch processing) and is highly scalable due to the independent nature of actors and their interactions.

Although the main design goal of Orleans is to simplify the programming model for cloud applications while providing scalability and reliability, providing efficiency is not less important for cloud applications that pay for consuming cloud resources. It is thus critical for Orleans to provide good performance. To ensure Orleans and its applications are executing within their performance requirements, it is necessary to continually track its performance. When performance requirements are not met, it is necessary to identify and resolve performance bottlenecks in a timely manner.

The need for PAD emerged since the early stages of the Orleans project when we occasionally faced non-trivial bugs that required manually looking through large sets of execution logs—literally searching for a needle in a haystack. We describe two such examples and specific data exploration techniques we used.

A.1.1 Stuck Random Number Generator

On one occasion, our regression performance test running on a cluster of servers failed after running fine for several hours. The external symptoms were lower than expected throughput and a large number of failed requests. We started by scanning and grepping through the logs with scripts to find a point in time where the number of failed requests suddenly started to grow. We then continued searching for the root cause.

After a laborious process of comparing performance counters across different servers, we discovered that some performance counters started to significantly diverge starting roughly at the time when the requests began timing out. In particular, there was one server that received a much larger number of requests than the other servers. Looking at the logs of this server, we consequently discovered that a disproportionally large number of actors were placed on it compared to other servers. This imbalance kept growing as the time advanced.

In this specific test, the actors were randomly placed on servers and the expectation was the number of actors (and as a result also the number of requests) should be roughly equal across all servers. We now had the evidence that from a certain point on, disproportionally more and more actors were placed on one server only. That lead us to look closer into the placement logic. After a thorough code analysis we discovered that we were using the random number generator (RNG) in a thread-unsafe manner. C# RNG is not thread-safe and, if accessed simultaneously by multiple threads, can get “stuck” returning zero forever. This caused actors to be placed on one server (with index zero) from the moment the RNG got stuck. The fix was to protect access to RNG with a lock.

A.1.2 Leaking Buffer Pool

In this occasion, we observed decreasing throughput and growing requests latency. By using similar manual techniques like in the RNG case above, we were able to correlate the performance degradation with increasing memory pressure. At approximately the same time as the performance started to degrade, the amount of available memory in the system started to shrink and the overhead of garbage collection activities started to increase. This lead us to suspect a memory leak. Although Orleans is written in a managed language, it uses a custom buffer pool for messages aimed at minimizing the rate of memory allocations and reducing the pressure on the memory subsystem and the garbage collector. Consequently, we found a bug in our buffer pool implementation that caused code acquiring the buffer from the pool to occasionally not release it back to the pool (*i.e.*, leaking memory).

The above two bugs helped us define a number of requirements for PAD: (1) ability to visualize performance counters across time and easily find points in time when values start diverging from the norm; (2) automatically find counters that exhibit large variance across different servers (Comparative Analysis within a dataset); (3) automatically correlate one counter that we knew to be a symptom of a problem to other counters that could potentially lead to the root cause of the anomaly (Correlation Analysis).

We have used Orleans performance tests data to validate the applicability of our tool. Current applications of PAD show that it is capable of supporting root cause analysis of performance problems in Orleans. It is important to note that the applicability of the techniques we have developed as well as the PAD tool itself are not limited to Orleans and can be applied to any multi-server distributed system which generates performance data.

A.2 Performance Counter Data and Challenges

This section describes about performance counter data and challenges that need to be addressed when analyzing them for performance anomaly detection.

A.2.1 Performance Counters

When run in production, multi-server distributed system performance is closely monitored. The collected performance data is stored in execution logs in the form of performance counters [20]. *Performance counters* track specific system states or resources during execution, such as CPU, memory, network, and application/framework specific information. Typical large production multi-server distributed systems run on clusters consisting of hundreds to thousands of servers. Each server periodically (typically every couple of minutes) tracks a large number of counters (hundreds in [20]) and stores them in the log.

In Orleans, a typical deployment consists of tens to hundreds of servers each tracking about 200 counters every five minute. The log is either stored separately for each server in its local file system or in a shared cloud storage, such as Azure Table storage[1]. Table A.1 provides examples of different classes of performance counters in Orleans.

Table A.1.: Examples of different classes of performance counters in Orleans

Type	**Examples**
Orleans Runtime	CPU usage, Percentage of time in garbage collection
Message Queues	Lengths of the send and receive message queues
Messaging	Number of total messages sent and received
Actors	Number of actors on a server

[1]`http://www.windowsazure.com/en-us/develop/net/how-to-guides/table-services/`

A.2.2 Challenges in Analyzing Performance Counter Data

The approaches used in PAD are based on the above mentioned performance diagnosis techniques. In addition, we have tackled a number of unique challenges in our setting that we detail below.

1. **Large data volumes.** As already mentioned, multi-server distributed systems generate a large amount of performance data, which is impossible to analyze manually. Navigating the vast amount of data is also *hard* as it is not easy to decide how to slide-and-dice it: (1) what set of performance counters to consider and (2) whether to look at the performance counters across different servers at a particular time, particular server across different times, or both.

   Developers sometimes have an idea, or clue, about the source of the problem. In such cases, they can manually inspect the relevant counters. For most performance issues, however, it is *hard* to decide what counters are relevant. Incorrect selection can cost valuable developer time at best and/or lead to wrong conclusions at worse. It is therefore important to inspect the performance counters that are more closely related to the problem under investigation.

2. **Insufficient training data.** One approach for performance diagnosis is to classify the counters into performance crisis situations, as done in [104, 105]. This kind of classification requires many different datasets and known labels (performance crisis situations) in order to apply machine learning based classification techniques. Such labeled datasets, however, are not always available. For example, although Orleans has been used in several projects within Microsoft, we did not have access to any labeled historical data. Because the labeled performance crisis data was unavailable, we could not apply machine learning classification techniques.

3. **Time correlation.** A distributed nature of the systems we consider poses a major challenge when correlating data collected from different servers across time. Servers are located on different physical machines, each having a different physi-

cal clock that may not be always synchronized. Unfortunately, some performance counters are sensitive to time and therefore even a 1 second approximation may give incorrect results.

A.3 PAD-Assisted Investigation

In this section, we describe how PAD assists developers in finding performance problems and anomalies. Developers start the troubleshooting process when they suspect that a performance related problem has occurred. This can be either deviation from explicit performance requirements like Service Level Agreements (SLA), implicit internal implementation requirements (*e.g.*, CPU time) or deviation from normal performance learned from previous executions.

The developer troubleshooting the system is engaged in a *PAD-assisted investigation process*, which combines their expert knowledge, manual steps, and automatic anomaly detection techniques to find performance problems and its root cause(s). PAD helps developers in every step of this process, which typically involves the following five steps: (1) collecting the performance counters data; (2) visualizing the data; (3) threshold analysis; (4) correlation analysis; and (5) comparative analysis. Steps 1, 4, and 5 are completely automated by PAD, while steps 2 and 3 are manual steps assisted by PAD. We now describe each step in a typical troubleshooting workflow session in detail.

Step 1 - Performance Data Collection. The developer starts by directing PAD to gather relevant performance counter data. The developer only needs to provide the location of the log files, or the Azure storage account that holds the logs, and the PAD automatically downloads the data, parses it, and stores it in an in-memory compact data structure.

Step 2 - Data Visualization. After data is gathered, the developer typically wants to visualize it. Visualizing the data can sometimes reveal the problem quickly without requiring further complicated analysis. System developers typically suspect certain performance counters, which they prefer to analyze first. The selection of the performance counters is based on developer's knowledge about the system, and the performance diagnosis issue of interest. For example, if the developer suspects that the system is experiencing memory pressure, the developer can use PAD to visualize and summarize performance counters related to garbage collection or memory usage. PAD provides three different visualizations for different data views:

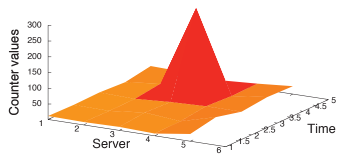

Figure A.1.: Detailed view

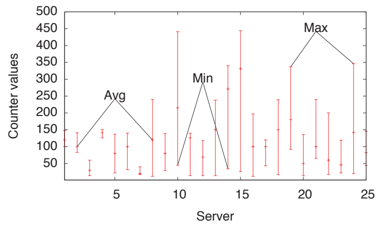

Figure A.2.: Server view

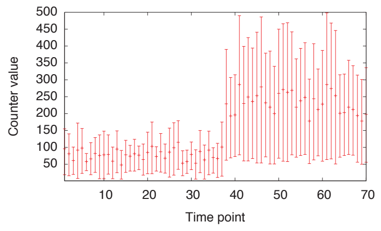

Figure A.3.: Time view

1. **Detailed view.** In the detailed view, PAD provides a 3D data plot. A 3D plot allows the developer to visualize and compare values both spatially (across all

servers) and temporally (across time). Figure A.1 illustrates an example plot produced by PAD for one performance counter. The X-axis represents server name, the Y-axis represents time, and the Z-axis represents the performance counter value. Any point in the plot captures the value of a performance counter at a particular time and in a particular server. The detailed view provides developers with overall trend information based on time and location. It can also prompt developers to perform further analysis when there are spikes (or anomalies) in the plot.

2. **Server view.** In the server view PAD visualizes summary statistics (*i.e.,* average, median, standard deviation, minimum, maximum, and quantiles) across time for a selected performance counter in each server. Figure A.2 illustrates an example of a server view graph, which is called a *stock type chart* [106], because it shows the average, maximum and minimum values of the performance counter in each server. This view allows developers to quickly compare performance counter summary statistics across all servers.

3. **Time view.** In the time view, PAD visualizes summary statistics (*i.e.,* average, median, standard deviation, minimum, maximum, and quantiles) across all servers for a selected performance counter at each time point. Figure A.3 illustrates an example of a time view graph. The time points are calculated with respect to the start time of system execution. This view allows developers to quickly compare performance counter summary statistics across all times regardless of the server.

By providing visualizations in three different views developers are able to gain more insight about system performance. For example, spike in the server view might be an indication of a hot server that performs more work than others. This allows developers to reduce the problem space into one particular server and concentrate further investigations at this server (like in the case in Section A.1.1)—eventually saving time. The visualization may not reveal any insights, or may trigger further analysis, including the need to look for other counters or compare certain counter values to predefined thresholds.

Step 3 - Threshold Analysis In threshold analysis, developers define threshold values for a given counter. PAD compares counter values (or their statistical properties, *i.e.*, means, medians, and quantiles) against the predefined threshold values. Performance counters that violate their threshold are reported back. Developers define thresholds using an XML configuration file. Listing A.1 illustrates an example for configuring thresholds for different performance counters.

```xml
<ThresholdConfig>
 <PerformanceCounter Name="Runtime.GC.PercentOfTimeInGC">
   <Rule AppliesTo="DetailView">
    <Statistic>Any</Statistic>
    <ExpectedValue>15</ExpectedValue>
    <ComparisonOperator>GreaterThan</ComparisonOperator>
   </Rule>
 </PerformanceCounter>
 <PerformanceCounter Name="MessageQueue.NumQueuedMsgs">
   <Rule AppliesTo="TimeView">
    <Statistic>Average</Statistic>
    <ExpectedValue>5</ExpectedValue>
    <ComparisonOperator>GreaterThan</ComparisonOperator>
   </Rule>
 </PerformanceCounter>
</ThresholdConfig>
```

Listing A.1: Example threshold analysis configuration file.

Developers can configure thresholds that apply to the detailed, server, or time view of each performance counter. Developers can specify what statistical property (*e.g.*, mean, median) to apply the rule to, or that the threshold should be compared with all values of the distribution. Likewise, developers can specify an expected percentage with respect to a statistical property. For example, the developer can ask to find all the occurrences of a particular performance counter exceeding more than X% from the median. Finally, threshold analysis in PAD supports Z-score [107] comparisons for each value in the distribution. This helps developers detect outliers when the values are distributed according to a normal distribution.

Threshold analysis is usually used in combination with expert knowledge related to acceptable range of values for certain counters. For example, the developer may want to check if the time spent in garbage collection (GC) has exceed 15% of the CPU time at any point in time. Listing A.1 illustrates an example configuration file with two rules: (1) find any time and server that the value of the *PercentOfTimeInGC* counter was greater than 15% ("any in the detailed view") and (2) find any time that the average value of the *NumQueuedMsgs* counter across all servers was greater than 5 ("average in the time view").

Step 4 - Correlation Analysis Using the first three steps above, the developers may be able to find what counters behave abnormally. This, however, may not facilitate root cause analysis. For example, imagine the developer has established that a certain server spends more than 15% in GC. The question now is why? What has happened in the system to cause this undesired behavior? The developer may not have a direct answer to this question and may not know the exact counter to look for. In such a situation, the developer can use correlation analysis to find the counters responsible for the root of the problem.

In correlation analysis PAD detects a set of counters that can explain the distribution of another performance counter. PAD supports two correlation analysis techniques:

1. **1) Pearson Coefficient**. The Pearson coefficient is used to check whether any two performance counters have a linear correlation [107]. Pearson coefficient calculates a value in the range [-1, 1]. The closer this value to either endpoint, the greater the correlation between the two performance counters.

2. **2) Spearman Coefficient**. The Spearman coefficient is a measure of how well two counters can be described using a monotonic function [107]. Spearman coefficient also provides a value in the range [-1, 1]. When the value is close to either endpoint, the two performance counters can be explained as a monotonically increasing function of the other.

PAD finds all explanatory counters that have a correlation value greater than some X (usually 0.9 in our usage) with the performance counter of interest using Pearson and Spearman correlations. This enables system developers to narrow down reasons for abnormal values in certain performance counters.

An example of using correlation analysis is when the time in GC exceeded the threshold of 15%, the tool found that this spike in GC activity correlated to a spike in a number of queued requests in this server. The server in question was receiving more load than the other servers and failed to keep up. This provided developers with enough information to look into the reason for load imbalance, and helped the developers identify the root cause.

Step 5 - Comparative Analysis Sometimes the developer may not know what counters to start with. In such situations, using visualization or threshold analysis might be too time consuming, provide too much data that is hard to analyze, and have a low chance of finding the root cause. In such cases PAD can help automatically detect anomalous counters based on statistical properties, such as average, median and quantiles, that deviate from other "normal" behavior of this counter.

PAD finds abnormal performance counters using *comparative analysis* [108]. Comparative analysis is a form of exploratory data analysis technique where statistical properties of different viewpoints of a performance counter dataset are compared against each other. More specifically, PAD implements the following comparative analysis methods:

1. **Comparative analysis within a dataset.** In this analysis, PAD uses a given dataset to find performance counters that have abnormal statistical properties either in specific servers, or at different time points.

PAD uses Equation A.1 for comparative analysis within a dataset.

$$X = \frac{\left|GlobalMedian - LocalMedian\right|}{GlobalStandardDeviation} \tag{A.1}$$

In this equation, we use a global median as a reference point to compare with a local median. A local median is a statistical property of the distribution of the performance counter in a server, or at a time point. PAD uses medians instead of averages because they are more robust to noisy data [104] (high and low fluctuations of a performance counter will have little impact on the median). Both metrics will therefore remain stable enough to use as a reference for comparison. By taking the different between the global and local medians, PAD calculates a local counter's deviation with respect to its global value. Finally, PAD normalizes the calculated deviation by the standard deviation of the global distribution to account for the fact that different performance counters can have different ranges of values. This provides PAD with a normalized method for comparing different counters that would be hard to compare using raw values.

2. **Comparative analysis between datasets.** PAD can also be used to compare different datasets, such as different regression test runs of the same application or different system releases. In this analysis, the developer specifies the reference ("correct") dataset, and PAD attempts to detect suspicious performance counters in the anomalous dataset. PAD uses Equation A.2 for comparative analysis between datasets.

$$X = \frac{\left|RefDatasetMedian - DatasetMedian\right|}{RefDatasetStdDev} \tag{A.2}$$

As illustrated in Equation A.2, PAD uses global medians and standard deviations of each performance counter in each dataset to calculate the deviation value X. PAD automatically performs the comparative analysis for all counters (the developer does not need to specify specific counters as in the threshold or comparative analysis). Once the suspicions counters are found, the developer can use PAD for visualization, threshold, and correlation analysis of the performance counters PAD has identified.

A.4 Implementation of PAD

Figure A.4 shows the overall workflow of PAD and its internal modules. PAD can

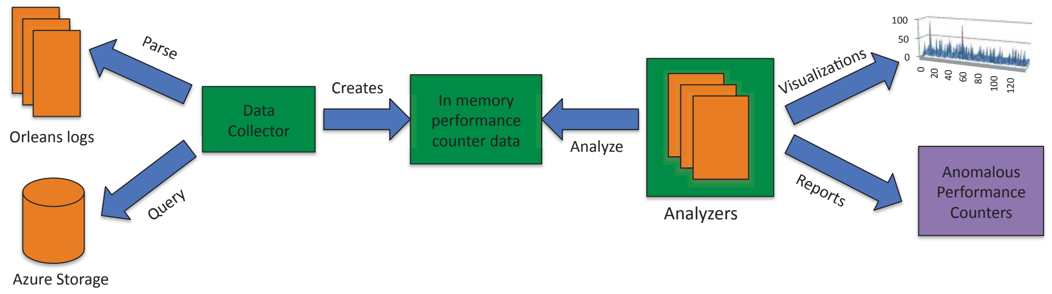

Figure A.4.: Design of PAD.

either collect data from Azure storage or log files. The data collection component is decoupled from the analysis components so that new data sources can be integrated without changing the analysis modules. After collecting the data, PAD builds an in memory model that is used by all analysis modules.

PAD (by default) implements the analysis techniques described in the previous section. Developers can also extend it with their own analysis techniques through an extensible analysis framework. The different features of PAD (*e.g.*, how to collect data, what type of analysis to use, and how to use visualization) are easily configurable via XML. Last, the visualization component of PAD is based on automatically generating Excel charts for selected set of performance counters and uses C# COM interface of Microsoft Excel.

Time correlation. As specified in Section A.2.2, another challenge in analyzing the performance of Orleans is correlating the counters across different servers at different points in time. This is a common problem in distributed systems, where there is no global clock shared by all servers and where per-server clocks may not be fully synchronized [109].

Prior research efforts have proposed several techniques, such as vector clocks [110], to address this problem. These approaches, however, are intrusive as they requires system instrumentation, and send messages between servers to perform the correlation. PAD uses a different approach. PAD's goal is to find the distribution of values of a particular performance counter at time point t_i from all servers. Since servers do not start at exactly the same time, we first find two time points $[t_s, t_f]$ where t_s is the latest starting time point of performance counters recording across all servers and t_f is the earliest finish time point of counters recording across all servers. We consider counter values in all servers during $[t_s, t_f]$ only.

Assuming the configured periodic logging interval of performance counters is d, PAD calculates the maximum number of time points N that can be contained inside the time interval $[t_s, t_f]$ using Equation A.3.

$$N = \left\lfloor \frac{t_f - t_s}{d} \right\rfloor \tag{A.3}$$

Since clocks of different servers may not be synchronized, all servers may not have exactly N time points. PAD therefore takes the maximal N' time points that are common to each server such that $N' \leq N$. PAD then indexes each time point from 1 to N' starting from t_s in each server. Because we have taken the same number of points from each server, it allows PAD to correlate performance counter values at similar indices in each server. The distribution of performance counter values at each time point are the correlated values at each index.

A.5 Applying PAD to Orleans

This section discusses two applications of PAD to analyze the performance counters in Orleans. In these particular scenarios, we used automated Orleans performance tests running on 25 servers machines and a set of client machines as load generators. Each client is configured to send 1 million requests.

A.5.1 Unbalanced DHT

In Orleans, actor instances are hosted on all servers. A distributed directory maps actor identities to their locations so incoming requests are brokered to their correct locations. The actor registry is implemented as a *Distributed Hash Table (DHT)* [111]. Each server is responsible for hosting a portion of the DHT. It is important to keep the DHT balanced so each server handles roughly the same amount of requests related to resolving actors locations.

During one test, Orleans was experiencing lower than expected throughput. We first analyzed Orleans performance counters using PAD by performing a comparative analysis within a dataset (Equation A.1 in Section A.3) on a problematic performance test dataset. PAD found two anomalous performance counters in one particular server: *Registrations.Local* counter and *Registrations.Remote.Received* counter.

This means that the number of local registrations of the DHT in this server was high compared to other servers *(Registrations.Local)* and that it also received more remote registration requests than other servers *(Registrations.Remote.Received)*. This was caused by the fact that this server was responsible for a much larger portion of the DHT.

PAD therefore was able to correctly identify the anomalous performance counters related to this issue. More importantly, PAD helped us pinpoint the root cause of performance degradation.

A.5.2 Performance Bottleneck and Tuning Analysis

We also used PAD to assist us in evaluating various performance optimization techniques in Orleans. As part of this work, we analyzed the impact of the different optimization techniques on performance (*e.g.*, end-to-end throughput and latency) and specific performance counters. For example, we implemented a certain batching algorithm and inspected its impact on end-to-end throughput, number of messages, message size distribution, buffer pool, and number of socket system calls. PAD therefore allowed us to quickly assess the effectiveness of various optimization techniques on low-level system components and greatly shortened our trial cycle. Without PAD, detailed investigation of a large number of performance counters would be very difficult.

A.6 Related Work

A.6.1 Approaches that Rely on Historical Performance Data

There are several related works [19,20,22,112] on analyzing performance of large-scale distributed systems similar to Orleans. Similar to PAD, these approaches rely on performance counters to detect performance anomalies. Foo et al. [19] calculate performance signatures from previous executions and use them as a baseline to compare against performance signatures of new executions. They assume that older executions do not suffer from performance anomalies. This approach is close to regression testing as it validates if anomalies are introduced into newer software versions. They, however, only do comparative analysis, which only provides a Yes/No answer on performance anomalies. In contrast, PAD facilitates different types of analysis beyond regression testing.

Nagaraj et al. [7] propose a method to compare two system logs, one with good and one with bad performance. After categorizing log messages as events and states, they calculate summary statistics for event timings, event counts, and state variable values used to compare the logs. Their approach is similar to the comparative analysis in PAD, but they do not provide other non-comparative techniques.

Bodik et al. [104] also propose a signature-based performance anomaly detection. Their method calculates signatures called *fingerprints* from historical performance data collected during a performance crisis. The goal is to quickly identify whether a similar performance crisis has occurred in the past so that known solutions can be applied. This approach is hard to apply when there are no previously known crises.

A.6.2 Approaches that Do Not Require Historical Data

Malik et al. [20] applied *Principal Component Analysis (PCA)* [113] to reduce the number of counters used to analyze performance anomalies. The main assumption is that counters that have high variance are the ones that represent the performance anomalies. This assumption, however, does not hold in all cases. For example, a system that experiences varying workloads may result in high variances in most performance counters without any actual performance problem.

Attariyan et al. [49] proposed a *performance summarization* approach for identifying root causes of performance anomalies based on human errors, such as misconfigurations. They used dynamic binary instrumentation [5] to monitor application as it executes instead of execution logs or performance counters.

However, their techniques only focus on misconfigurations and do not help to find root causes for other performance problems, such as bugs in implementation or design, like PAD. Finally, there are other approaches [24, 25, 34] that use annotated software models to detect performance anti-patterns [114]. These approaches, however, use software model simulations and not real production software. Moreover, these approaches do not rely on statistical analysis, but instead use rules and logical-predicate analysis to detect performance problems.

A.7 Lessons Learned and Conclusions

In this paper, we presented PAD, a tool to analyze performance counters in multi-server distributed systems. PAD combines user-driven navigation analysis with automatic correlation and comparative analysis techniques. Based on our experience in applying PAD to the Orleans framework, we discovered that PAD was able to reduce developers' time and efforts in detecting anomalous performance cases and improve developers' ability to perform deeper analysis of such behaviors. Below we detail the lessons learned based on our experience with PAD.

1. **Visualization and summary statistic is a key part in performance anomaly detection.** Visualization provides a quick overview of performance and triggers deeper analysis when needed. We believe that visualization should be the first step in human-based performance anomaly detection. Multiple view points (server or time) as well as summary statistics (*e.g.*, average, median, standard deviation, minimum, maximum, and quantiles) are very helpful in navigating the large amounts of data, and selecting a view for further analysis.

2. **Reducing the data size.** It is important to reduce the number of performance counters before visualizing data and performing root cause analysis. This saves developers time and effort by focusing their attention on data most relevant to the anomaly. For example, although Orleans has nearly 200 performance counters, we discovered certain performance issues can be summarized using only few counters.

   We also tried to apply Principal Component Analysis to reduce the number of performance counters used in the analysis. This approach transformed the original performance counters into a different, smaller dataset with different dimensions. The new counters, however, bared no semantic meaning, could not be correlated back to the actual system, and did not help us with root causes analysis.

3. **Fully automated root cause analysis for performance anomalies is hard.** Existing research on automating root cause analysis is based on functional failures [115, 116]. As explained in Section A.6, expert knowledge is required to analyze the root causes of performance anomalies. This knowledge differs from system to system, which makes it hard to generalize and automate. PAD addresses this challenge by combining automatic correlation and compara-

tive analysis with manual user-driven navigation analysis. We still believe that commonalities between different automated root cause analysis processes must be identified and reused. Finally, techniques to formalize the required expert knowledge from different domains are required so developers can begin developing domain-specific automated techniques for root cause analysis.

APPENDIX B DETAILED CALL TREES OF EXCESSIVE DYNAMIC MEMORY ALLOCATIONS DETECTED BY EMAD

This appendix presents the detailed call trees of excessive dynamic memory allocations in SQLite, TAO and Axis2-C. The maximum call frequency of each routine is shown inside the parenthesis after routine name.

B.1 Excessive Dynamic Memory Allocations in SQLite

```
sqlite3PagerWrite (16417)
insertCell (9222)
sqlite3BtreeInsert (8479)
sqlite3VdbeExec (4342)
sqlite3_step (14098)
sqlite3_column_name (10659)
sqlite3_prepare (49)
sqlite3_exec (3439)
__libc_free (3385)
sqlite3_free (175158)
sqlite3VdbeMemRelease (788186)
sqlite3BtreeNext (378600)
```

Listing B.1: Call-tree for the routine *pager_write* in SQLite-3.5.9

```
sqlite3DbMallocZero (45858)
sqlite3Expr (26765)
sqlite3Parser (26436)
sqlite3RunParser (102322)
sqlite3Prepare (3450)
```

```
sqlite3LockAndPrepare (3450)
sqlite3_prepare (3450)
sqlite3_exec (3439)
__libc_free (3385)
sqlite3_free (175158)
sqlite3VdbeMemRelease (788186)
sqlite3BtreeNext (378600)
sqlite3VdbeExec (60800)
sqlite3_step (14098)
sqlite3_column_name (10659)
```

Listing B.2: Call-tree for the routine *sqlite3DbMallocRaw* in SQLite-3.5.9

```
write32bits (14848)
sqlite3BitvecTest (14848)
sqlite3PcacheMakeDirty (8101)
pager_write (14318)
sqlite3PagerWrite (14318)
insertCell (8727)
sqlite3BtreeInsert (7798)
applyAffinity (7473)
btreeParseCell (14430)
sqlite3BtreeKeySize.part.112 (13948)
```

Listing B.3: Call-tree for the routine *memjrnlWrite* in SQLite-3.8.5

B.2 Excessive Dynamic Memory Allocations in TAO

```
operator>> (10000)
TAO_GIOP_Message_Gen_Parser_12::parse_header (10000)
TAO_GIOP_Message_Base::process_request (10000)
TAO_GIOP_Message_Base::process_request_message (10000)
```

```
TAO_Transport::process_parsed_messages (10000)
TAO_Transport::handle_input_parse_data (10000)
TAO_Transport::handle_input (10001)
TAO_Connection_Handler::handle_input_internal (10001)
TAO_Connection_Handler::handle_input_eh (10001)
TAO_IIOP_Connection_Handler::handle_input (10001)
ACE_TP_Reactor::dispatch_socket_event (10001)
ACE_TP_Reactor::handle_socket_events (10002)
ACE_TP_Reactor::dispatch_i (10002)
ACE_TP_Reactor::handle_events (10003)
TAO_ORB_Core::run (10003)
CORBA::ORB::run (1)
CORBA::ORB::run (1)
TAO_ORB_Manager::run (1)
main (1)
__libc_start_main (1)
_start (1)
```

Listing B.4: Complete call tree for the excessive dynamic memory allocation in TAO.

B.3 Excessive Dynamic Memory Allocations in Axis2-C

This appendix presents the excessive dynamic memory allocations of Axis2-C that were detected by EMAD.

```
  guththila_xml_reader_wrapper_get_prefix (16100)
  axiom_xml_reader_get_prefix (16100)
  axiom_stax_builder_process_namespaces (16100)
  axiom_stax_builder_create_om_element (16100)
  axiom_stax_builder_next_with_token (16000)
  axiom_soap_builder_next (38000)
  axiom_soap_builder_create (20000)
  axis2_http_transport_utils_process_http_post_request (2000)
  axis2_apache2_worker_process_request (2000)
  axis2_handler (2000)
  ap_run_handler (2000)
```

Listing B.5: Call-tree for the routine *guththila_get_prefix*.

```
axiom_element_free (16000)
axiom_node_free_detached_subtree (36000)
axiom_node_free_tree (4000)
axiom_document_free (2000)
axiom_stax_builder_free (2000)
axiom_soap_builder_free (2000)
axiom_soap_envelope_free (2000)
axis2_msg_ctx_free (4000)
axis2_apache2_worker_process_request (4000)
axis2_handler (2000)
ap_run_handler (2000)
```

Listing B.6: Call-tree for the routine *axutil_hash_first*.

```
axutil_qname_create (58901)
axiom_element_get_qname (6089)
axiom_children_qname_iterator_has_next (6662)
axiom_element_get_first_child_with_qname (2080)
axis2_addr_in_extract_svc_grp_ctx_id (2000)
axis2_addr_in_handler_invoke (2000)
axis2_handler_invoke (2000)
axis2_phase_invoke (18000)
axis2_engine_invoke_phases (10000)
axis2_engine_receive (4000)
axis2_http_transport_utils_process_http_post_request (2000)
```

Listing B.7: Call-tree for the routine *axutil_strdup*.

```
axutil_qname_to_string (40000)
axiom_element_get_attribute (18130)
axis2_addr_in_extract_ref_params (6000)
axis2_addr_in_handler_invoke (2000)
axis2_handler_invoke (2000)
axis2_phase_invoke (18000)
axis2_engine_invoke_phases (10000)
axis2_engine_receive (4000)
axis2_http_transport_utils_process_http_post_request (2000)
axis2_apache2_worker_process_request (2000)
axis2_handler (2000)
```

Listing B.8: Call-tree for the routine *axutil_stracat*.

```
axiom_namespace_create Id (36000)
axis2_addr_out_handler_process_string_info (6000)
axis2_addr_out_handler_invoke (6000)
axis2_handler_invoke (2000)
axis2_phase_invoke (18000)
axis2_engine_invoke_phases (10000)
axis2_engine_receive (4000)
axis2_http_transport_utils_process_http_post_request (2000)
axis2_apache2_worker_process_request (2000)
axis2_handler (2000)
ap_run_handler (2000)
```

Listing B.9: Call-tree for the routine *axutil_string_create*.

```
axiom_element_create (20000)
axiom_soap_header_block_create_with_parent (8000)
axiom_soap_header_add_header_block (8000)
axis2_addr_out_handler_process_string_info (6000)
axis2_addr_out_handler_invoke (6000)
```

```
axis2_handler_invoke (2000)
axis2_phase_invoke (18000)
axis2_engine_invoke_phases (10000)
axis2_engine_receive (4000)
axis2_http_transport_utils_process_http_post_request (2000)
axis2_apache2_worker_process_request (2000)
```

Listing B.10: Call-tree for the routine *axiom_node_create*.

```
 axutil_hash_get (80578)
axutil_param_container_get_param (24172)
_init (8029)
axis2_svc_grp_get_param (6027)
axis2_svc_get_param (4027)
axis2_op_get_param (2016)
axis2_msg_ctx_get_parameter (2000)
axis2_ctx_handler_invoke (2000)
axis2_handler_invoke (2000)
axis2_phase_invoke (18000)
axis2_engine_invoke_phases (10000)
```

Listing B.11: Call-tree for the routine *axutil_hash_find_entry*.

```
axiom_stax_builder_create_om_element (16100)
axiom_stax_builder_next_with_token (16000)
axiom_soap_builder_next (38000)
axiom_soap_builder_create (20000)
axis2_http_transport_utils_process_http_post_request (2000)
axis2_apache2_worker_process_request (2000)
axis2_handler (2000)
ap_run_handler (2000)
ap_invoke_handler (2000)
ap_process_request_internal (2000)
```

```
ap_process_request (2000)
```

Listing B.12: Call-tree for the routine *axutil_string_create_assume_ownership*.

VITA

VITA

Thelge Manjula Peiris received his bachelor's degree from University of Moratuwa, Sri Lanka in 2006 and his masters's degree from Purdue University Indianapolis in 2013. Before joining Purdue University he worked as a software engineer at WSO2. Upon receiving his Ph.D. from Purdue University he joined Amazon Web Services as a software engineer.