Workload-sensitive Timing Behavior Analysis for Fault Localization in Software Systems

Matthias Rohr

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2014

1. Examiner: Prof. Dr. Wilhelm Hasselbring
Kiel University

2. Examiner: Prof. Dr. Lars Grunske
University of Stuttgart

Date of the oral examination (disputation): October 2nd, 2014

Please cite as:

> Matthias Rohr. *Workload-sensitive Timing Behavior Analysis for Fault Localization in Software Systems*. Dissertation (PhD thesis), Faculty of Engineering, Kiel University, Kiel, Germany, Jan. 2015.

```
@phdthesis{rohr2015workloadSensitiveTimingBehaviorAnalysis,
  author    = {Matthias Rohr},
  title     = {Workload-sensitive Timing Behavior Analysis
    for Fault Localization in Software Systems},
  type      = {Dissertation ({PhD} thesis)},
  address   = {Kiel, Germany},
  school    = {Faculty of Engineering, Kiel University},
  month     = jan,
  year      = {2015},
  isbn      = {978-3-7347-4516-4},
}
```

© 2015 by Matthias Rohr
Herstellung und Verlag: BoD – Books on Demand, Norderstedt

Abstract

Software timing behavior measurements, such as response times, often show high statistical variance. This variance can make the analysis difficult or even threaten the applicability of statistical techniques. This thesis introduces a method for improving the analysis of software response time measurements that show high variance.

Our approach can find relations between timing behavior variance and both trace shape information and workload intensity information. This relation is used to provide timing behavior measurements with virtually less variance. This can make timing behavior analysis more robust (e.g., improved confidence and precision) and faster (e.g., less simulation runs and shorter monitoring period). The thesis contributes TracSTA (Trace-Context-Sensitive Timing Behavior Analysis) and WiSTA (Workload-Intensity-Sensitive Timing Behavior Analysis). TracSTA uses trace shape information (i.e., the shape of the control flow corresponding to a software operation execution) and WiSTA uses workload intensity metrics (e.g., the number of concurrent software executions) to create context-specific timing behavior profiles.

Both the applicability and effectiveness are evaluated in several case studies and field studies. The evaluation shows a strong relation between timing behavior and the metrics considered by TracSTA and WiSTA. Additionally, a fault localization approach for enterprise software systems is presented as application scenario. It uses the timing behavior data provided by TracSTA and WiSTA for anomaly detection.

Zusammenfassung

Die Analyse von Zeitverhalten wie z.B. Antwortzeiten von Software-Operationen ist oft schwierig wegen der hohen statistischen Varianz. Diese Varianz gefährdet sogar die Anwendbarkeit von statistischen Verfahren. In dieser Arbeit wird eine Methode zur Verbesserung der Analyse von Antwortzeiten mit hoher statistischer Varianz vorgestellt.

Der vorgestellte Ansatz ist in der Lage, einen Teil der Varianz aus dem gemessenen Zeitverhalten anhand von Aufrufsequenzen und Schwankungen in der Nutzungsintensität zu erklären. Dadurch kann praktisch Varianz aus den Messdaten entfernt werden, was die Anwendbarkeit von statistischen Analysen in Bezug auf Verlässlichkeit, Präzision und Geschwindigkeit (z.B. kürzere Messperiode und Simulationsdauer) verbessern kann. Der Hauptbeitrag dieser Arbeit liegt in den zwei Verfahren TracSTA (Trace-Context-Sensitive Timing Behavior Analysis) und WiSTA (Workload-Intensity-Sensitive Timing Behavior Analysis). TracSTA verwendet die Form des Aufrufflusses (d.h. die Form der Aufrufsequenz, in die ein Methodenaufruf eingebettet ist). WiSTA wertet die Nutzungsintensität aus (z.B. Anzahl gleichzeitig ausgeführter Methoden). Dies resultiert in kontextspezifischen Antwortzeitprofilen.

In mehreren Fall- und Feldstudien wird die Anwendbarkeit und die Wirksamkeit evaluiert. Es zeigt sich ein deutlicher Zusammenhang zwischen dem Zeitverhalten und den von TracSTA und WiSTA betrachteten Einflussfaktoren. Zusätzlich wird als Anwendungsszenario ein Ansatz zur Fehlerlokalisierung vorgestellt, welcher von TracSTA und WiSTA bereitgestellte Antwortzeiten zur Anomalieerkennung verwendet.

Table of Contents

Abstract iii

Zusammenfassung v

1. Introduction 1
1.1. Motivation and Problem . . . 1
1.2. Contributions and Evaluation . . . 3
1.3. Thesis Structure . . . 6
1.4. Bibliographical Notes . . . 6

2. Foundations 9
2.1. Software Timing Behavior . . . 9
2.2. Software Faults and Dependability of Software Systems . . 23
2.3. Automatic Fault Localization for Software Systems 27
2.4. Anomaly Detection . . . 32
2.5. Software Application Monitoring . . . 35

3. Fault Localization Approach 39
3.1. Approach Overview and Fault Localization Assumptions . 39
3.2. Software System Model and Monitoring Model . . . 43
3.3. Instrumentation and Trace Synthesis . . . 46
3.4. Trace-Context-Sensitive Timing Behavior Analysis . . . 48
3.5. Workload-Intensity-Sensitive Timing Behavior Analysis . . 49
3.6. Anomaly Detection . . . 52
3.7. Anomaly Correlation and Visualization . . . 58

4. TracSTA: Trace-Context-Sensitive Timing Behavior Analysis 63
4.1. Correlation between Timing Behavior and Trace Context . 64
4.2. Trace-Context-Sensitive Timing Behavior Analysis . . . 67
4.3. Empirical Evaluation . . . 78
4.4. Summary . . . 89

5. WiSTA: Workload-Intensity-Sensitive Timing Beh. Analysis 91
5.1. Correlation btw. Timing Behavior and Workload Intensity 92
5.2. Workload-Intensity-Sensitive Timing Behavior Analysis . 96
5.3. Empirical Evaluation 103
5.4. Summary 111

6. Related Work 115
6.1. Context-sensitive Timing Behavior Analysis 115
6.2. Fault Localization & Failure Diagnosis for Software Systems 132

7. Conclusions 145
7.1. Summary 145
7.2. Discussion 148
7.3. Threats to validity 152
7.4. Future Work 154

Appendices

Appendix A. Timing Behavior Distribution Examples 159

Appendix B. Standard Deviation Reduction 167

Appendix C. Listing Example Chapter 5 169

Appendix D. Call Graph Profiling Tools 171
D.1. Gprof 172
D.2. Google's Perftools CPU Profiler 173
D.3. Valgrind's Call Graph Generator Callgrind 174
D.4. Java's HPROF Profiler 174
D.5. NetBeans 6.9 Java Profiler 176

Appendix E. Garbage Collection Analysis 179

List of Figures 183

List of Tables 187

Bibliography 189

Index 223

1. Introduction

This thesis introduces a timing behavior analysis method for distributed enterprise software systems and an online fault localization approach that uses it. This chapter starts by motivating the thesis. Section 1.2 states its contribution and presents how it is evaluated. The structure of the thesis and bibliographical notes follow in Sections 1.3 and 1.4.

1.1. Motivation and Problem

Many companies depend on enterprise software systems, especially if these systems provide services and products, such as online banking, online stores, and online auction sites. This thesis is in the domain of automatic management of availability and performance of such systems. Both are essential requirements: Low availability and outages can lead to significant loss of revenue. Insufficient performance can also motivate customers to choose a competitor and it can be a waste of computational resources and IT costs.

One strategy for improving the availability and performance of enterprise software systems is based on the monitoring and analysis of software timing behavior. For optimizing performance, there are automatic approaches that adapt the systems during runtime (e.g., Arlitt et al., 2001, Garlan et al., 2003, Diaconescu et al., 2004, and van Hoorn, 2014), and methods to optimize performance, such as profiling [Graham et al., 1982], localization of bottlenecks [Smith and Williams, 2001b; Hoffman, 2005], workload characterization [Menascé et al., 1999], and regression benchmarking [Kalibera, 2006]. For improving availability, several automatic failure diagnosis approaches have been proposed, such as provided by Chen et al. [2002], Aguilera et al. [2003], Agarwal et al. [2004], and Yilmaz et al. [2008]. This thesis primarily aims to support approaches for automatic failure diagnosis based on software timing behavior monitoring. The automation of failure diagnosis can potentially reduce repair times and therefore improve availability (see Page 25), as manual failure diagnosis of software faults is time-consuming and error-prone [Rohr,

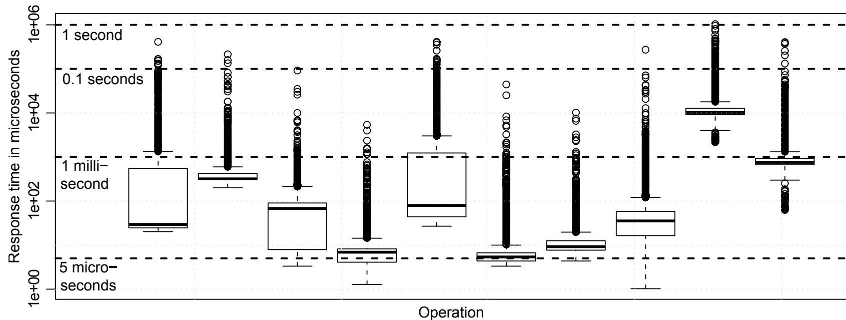


Figure 1.1.: Response times of ten internal software operations of an online store.

2007; Marwede et al., 2009]. The ongoing trend of increasing system complexity [Lyu, 2007] additionally motivates the automation of failure diagnosis.

This thesis addresses the problem that software timing behavior has statistical properties that makes it difficult to analyze for the purposes described above. For instance, software timing behavior tends to show very high variance [Mielke, 2006; van Hoorn, 2007], which challenges the applicability and robustness of several statistical methods. Possible explanations are for instance varying workload, as well as deterministic and non-deterministic mechanisms, such as scheduling and caching. An example for the high variance is shown in the boxplot of some internal software response times of an online shopping platform (Case Study 3, in Section 5.3.3) in Figure 1.1: The operation response times spread over several levels of magnitude; for instance the response times of the 8th operation from the left (an operation that provides price information) spreads between 62 microseconds and 400 milliseconds (factor 6451). Note, these are response times of internal software operations and are not end-to-end response times. In addition to high variance, the distribution of software timing behavior often shows heavy tails and multimodality. These and other characteristics are closer described in Section 2.1.3 and an example for a multimodal distribution can be found in Figure 2.5 on Page 19.

The relevance of the problem described above arises from the general statistical consequences and consequences to concrete application scenarios. A general consequence of high variance is that it makes it more difficult to draw statistical conclusions from single measurements [Menascé

and Almeida, 2001, pp. 168]. This reduces the quality of results (e.g., in terms of lower confidence and accuracy) or more measurement values are needed [Mitrani, 1982]. Multimodality and heavy-tailed distributions can question the applicability of timing behavior analysis approaches, such as failure diagnosis, online performance management, and regression benchmarking. Additionally, many approaches internally rely on foundational statistical methods that assume simple distributions or use the mean value as representative value for the complete distribution. Approaches that use simulation would require more simulation runs for scenarios with high variance for achieving the same confidence [Jain, 1991; Mitrani, 1982]. Especially for anomaly detection, high variance, multimodality, and heavy-tailed distributions could lead to high false alarm rates and bad anomaly detection quality. This is valid for modern research approaches and also for classical monitoring and control systems that allow the administrators to define thresholds. For instance, a classical way to determine thresholds goes back to Shewhart [1931]'s foundational work on quality control: an upper and lower control limit are defined by a range of three standard deviations around the mean value of historical observations.

We see two applicability requirements for practical solutions to the general problem of this thesis (high variance and multimodality in software timing behavior):

Rq.1 A solution should be able to continuously operate in production systems with a suitably low overhead.

Rq.2 It should be easily applicable to large software systems. This includes that only small changes have to be made to the system (e.g., non-intrusive monitoring) and that the configuration effort is relatively low (e.g., not many parameters, architectural model automatically learned).

1.2. Contributions and Evaluation

This thesis has primary and secondary contributions as illustrated in Figure 1.2. The two highlighted elements are the primary contributions, consisting of the two novel timing behavior analysis methods TracSTA (Trace-Context-Sensitive Timing Behavior Analysis) and WiSTA (Workload-Intensity-Sensitive Timing Behavior Analysis). Both TracSTA and WiSTA are quantitatively evaluated in industry- and lab-studies. The

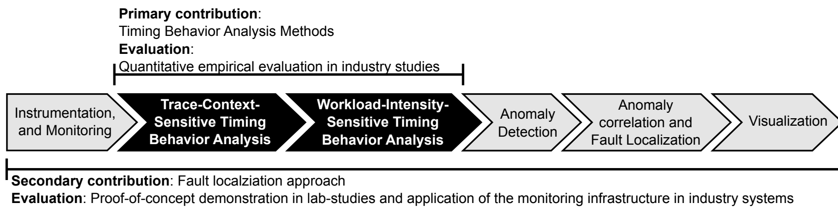


Figure 1.2.: Overview of the fault localization approach and the two central contributions of this thesis.

secondary contribution consists of a new approach to fault localization. This approach embeds both WiSTA and TracSTA, and extended our monitoring instrumentation Kieker [Rohr et al., 2008c; van Hoorn et al., 2009]. The applicability of the fault localization approach is evaluated in industry systems, while the fault localization capabilities are demonstrated in a lab study.

The two primary contributions TracSTA and WiSTA both analyze timing behavior in the context of workload information. In both cases, the workload information is extracted from control flow information of application-layer monitoring data. Both TracSTA and WiSTA add annotations to the monitoring data. These annotations categorize each response time according to their workload contexts. These annotations can be used as additional information in an analysis of the response times to reduce the problem with high variance and multimodality described above. This follows Menascé and Almeida [2001]'s general suggestion to "reduce the variability of measurements" by dividing timing behavior measurements into classes that correspond to similar requests [Menascé and Almeida, 2001, p. 168].

The combined hypothesis of TracSTA and WiSTA has two parts:

$\mathbf{H}_{\mathrm{TW1}}$ A significant part of the variance in software operation response times of multi-user enterprise software systems is correlated to the full trace shape and workload intensity.

$\mathbf{H}_{\mathrm{TW2}}$ This correlation can be used in practice to "reduce" the variance from the perspective of subsequent timing behavior analysis steps (see Figure 1.2), such as anomaly detection.

More detailed hypotheses for TracSTA ($\mathbf{H}_{\mathrm{T1a}}$, $\mathbf{H}_{\mathrm{T1b}}$, $\mathbf{H}_{\mathrm{T2}}$) are in Chapter 4 and for WiSTA ($\mathbf{H}_{\mathrm{W1}}$, $\mathbf{H}_{\mathrm{W2}}$) in Chapter 5. The wording "in practice" of $\mathbf{H}_{\mathrm{TW2}}$ corresponds to the requirements **Rq.1** and **Rq.2** of Page 3.

TracSTA categorizes software operation response times based on the trace shape. The timing behavior of software operations can be quite different depending on the type of a call, as demonstrated later in this thesis. The type of a call may be specific for instance in the identity of the caller, or in whether subcalls are made or not. Our TracSTA method extends the profiling research presented by Ammons et al. [1997] and Graham et al. [1982]. The empirical evaluations of TracSTA demonstrate in industry systems that our trace contexts relate to significantly more variance than the concepts provided by related work. Furthermore, our evaluation shows that TracSTA in some cases splits a multimodal distributions into multiple unimodal distributions.

WiSTA is based on the observation that operation response times in enterprise software systems depend on the *amount* of workload. Typically, software operation response times increase with increasing workload [Jain, 1991], as parallel requests compete for shared resources (e.g., CPU, or memory access). WiSTA distinguishes timing behavior by workload intensity. As with TracSTA, the empirical evaluation of WiSTA was also performed in industry systems. This demonstrates that WiSTA's metrics corresponds to a significant part of variance in the timing behavior of real systems. Workload intensity analysis is a common part in the domain of queueing network analysis and other analytical performance analysis methods (e.g., Menascé and Almeida, 2001; Smith and Williams, 2001b; and Jain, 1991) and some authors use such analytical models also for online analysis during system operation (e.g., Nou et al., 2008). WiSTA does not use analytical performance analysis methods, and therefore, it has relatively low computational resource requirements even in large systems. Furthermore, WiSTA introduces a new metric for workload intensity that can be automatically derived by typical monitoring frameworks, without requiring hardware resource monitoring. WiSTA automatically addresses that in a distributed system, software operations usually consume directly only local resources and that parallel software operation executions may have different operation-individual resource sharing issues.

The secondary contribution of this thesis is a new fault localization approach based on timing behavior monitoring. This contribution is secondary in the sense that no empirical evaluation was performed, it is less focused than the primary contribution, and it partly assembles existing methods. The fault localization approach uses TracSTA and WiSTA to isolate variance and multimodality before anomaly detection. In case of a failure detection, all anomalies are correlated in the context

of their architectural correspondence to compute for each component instance a probability for containing a fault for a failure. The approach visualizes the fault localization result with colors according to fault probability in a 3-layer architectural visualization. WiSTA and TracSTA can both significantly improve the anomaly detection quality and the fault localization quality through the reduction of variance and multimodality. An empirical evaluation of the complete fault localization approach is out of scope of this thesis, because of the large number of controlled variables for each conceptual step of the fault localization approach (Figure 1.2) and the challenge to specify realistic fault load and workload for several systems. Nonetheless, the fault localization approach is demostrated in a distributed system in the lab that is exposed to artificial fault load.

1.3. Thesis Structure

Chapter 2 provides the required foundational background for the remainder.

Chapter 3 presents our approach for fault localization, which provides the application scenario for the two primary contributions. Furthermore, it defines basic terminology and a software execution model for the later chapters.

Chapters 4 and 5 introduce the two main contributions TracSTA and WiSTA. Both chapters contain the empirical evaluation for each timing behavior analysis method.

Chapter 6 presents the related work for all contributions of the thesis.

Chapter 7 concludes the thesis with a summary, a discussion, and potential future work.

1.4. Bibliographical Notes

Parts of this thesis have been published before in publications that have been authored or co-authored by the writer of this thesis:

- The research on trace-context-sensitive timing behavior analysis Chapter 4 was published in Rohr et al. [2008b] and Rohr et al. [2008a].

- The research on workload-intensity-sensitive timing behavior analysis of Chapter 5 was published in Rohr et al. [2010]. The research on workload intensity and on workload generation was supported by the master thesis of van Hoorn [2007], which was supervised by the author of this thesis.

- Our concept for timing behavior anomaly detection with a focus on considering workload of Chapter 3 was initially published in Rohr et al. [2007]. Some failure diagnosis concepts of Chapter 3 are based on our joint work in Marwede et al. [2009] and were addressed by the diploma thesis of van Hoorn [2007], which was supervised by the author of this thesis. Research on the visualization of failure diagnosis results and on fault localization have been conducted in the context of the master thesis of Marwede [2008], which was also supervised by the author of this thesis.

- The concepts on timing behavior monitoring in distributed software systems were published in Focke et al. [2007b] and Rohr et al. [2008c]. Some of this research was conducted and implemented in the context of the master thesis by Focke [2006], which was supervised by the author of this thesis.

- Parts of the foundations on software reliability and software faults have been coauthored and published in Eusgeld et al. [2008] and Ploski et al. [2007].

The following additional student theses and master theses have been supervised and conducted in the context of this thesis:

- Stransky [2006] explored failure diagnosis using neural networks.

- Sommer [2007] applied timing behavior anomaly detection for intrusion detection.

- Schwenkenberg [2007] contributed fault injection techniques.

2. Foundations

2.1. Software Timing Behavior

The ISO/IEC 9126-1 [2001] standard's taxonomy of quality characteristics provides the term efficiency for what is called performance in the software performance engineering community. Efficiency is further divided into *time behavior* and *resource utilization* [Jung et al., 2004; Sabetta and Koziolek, 2008], as illustrated in Figure 2.1. A common definition in the software performance evaluation community defines *performance* as "the degree to which a software system or component meets its objectives for timeliness" [Smith and Williams, 2001b]. This document uses the term *timing behavior* instead of time behavior, with a focus on timeliness, and without considering non-timing aspects such as memory consumption. The collection of software timing behavior is relatively easy, but its analysis can be very complex, as it results from processes on all the system layers from the hardware layer to the application layer [Yilmaz et al., 2008; Lashari and Srinivas, 2003; Stewart and Shen, 2005].

In the remainder of this section, major concepts of timing behavior metrics, timing behavior modeling and analysis methods are outlined. Additionally, statistical characteristics of monitored software timing behavior and its major influences (e.g., workload) are presented.

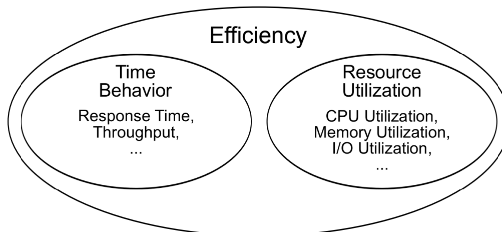


Figure 2.1.: Efficiency in the ISO/IEC 9126-1 [Sabetta and Koziolek, 2008].

2.1.1. Timing Behavior Metrics

In the following, definitions for several timing behavior metrics are provided, before timing behavior modeling and analysis methods are presented in Section 2.1.2.

2.1.1.1. Response Time and Execution Time

In general, *response time* is defined as the duration between a request and the corresponding response [Jain, 1991, p.37, and Smith and Williams, 2001b, p.3]. A request and response can be between a system user and a system, or between system components. This definition can be refined by considering the time to start computing the result, and to deliver the response, as illustrated in Figure 2.2. Response time (1) only focuses on the time required to create the response from the request, without the initial reaction time. Response time (2) and (3) both include the reaction time, but differ in including the time to deliver the response to the caller. Response time (3) is also called *end-to-end response time* [Jain, 1991].

This thesis focuses on operation response times in conformance to Figure 2.2's response time (1). The term *operation* refers to the software programming language concept, also called method (e.g., in Java and C#), routine, and procedure. The term *operation response time* is defined as the duration between the start and the end of an operation execution, including the time spent in other operation executions that are invoked by the execution, and without the time required to start the operation's execution and with the time to deliver its response. For simplicity, this thesis uses the term *response time* for operation response time (1). Not all operations have a response time, as an infinite loop can be part of operation.

We define the *execution time* of an operation as the fraction of the response time that is spent for the operation execution itself, *without* both the time spent for interacting with subcalls and without the response time of subcalls [van Hoorn, 2007; Rohr et al., 2010]. Therefore, the response time of an operation execution is the sum of its execution time, and the response times of all nested operation executions. Note, this definition is similar but not identical to definitions for instance by Musa et al., 1987 and Patterson and Hennessy, 2008 that define the execution time as the time actually spent by the CPU for executing instructions of a program, which is also termed *CPU time* [Smith and Williams, 2001b]. In the following, only the first definition is used in this thesis, as it uses the

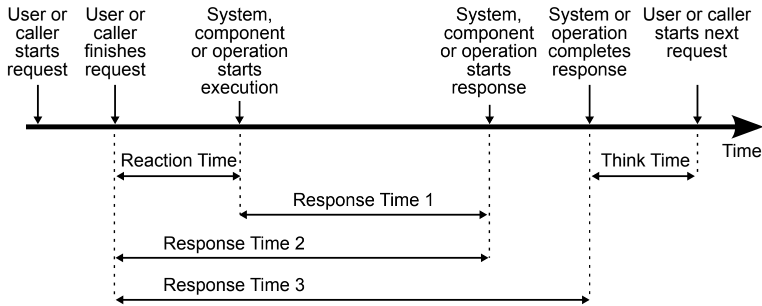


Figure 2.2.: Response time metrics. Illustration based on Jain [1991].

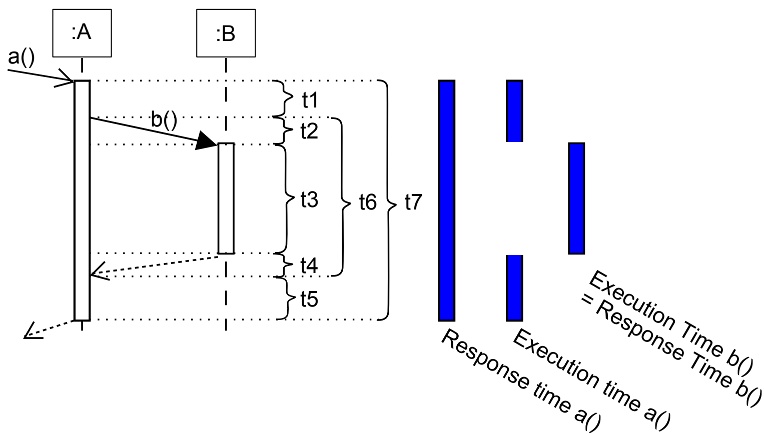


Figure 2.3.: Response times and execution times.

perspective of software application layer monitoring, in contrast to the second definition's viewpoint of lower system layers or resources.

An example for the above definitions of response time and execution time are provided in the sequence diagram in Figure 2.3: The response time of operation *a()* is the sum of *a()*'s execution time before and after the execution of *b()* (t1 and t5), the time to call *b()* (t2) and return from *b*'s execution (t4), and the execution time of *b()* itself (t3). The execution time of *b()* is equal to its response time, since *b()* has no subcalls, from the point of view of this monitoring instrumentation. The sloped operation call notations in Figure 2.3 indicate time durations needed by the execution environment to start or to end executions.

As mentioned above, some timing metrics are from the perspective of resources, such as CPU, I/O devices, or processes [Smith and Williams, 2001b]. These metrics explicitly consider details of scheduling and software-hardware interaction, such as transmission time, service time, waiting time, queueing time, and residence time. For instance, the literature on queueing network analysis (e.g., Menascé and Almeida, 2001; Jain, 1991) uses these metrics. Hardware resource monitoring requires deep instrumentation [Woodside, 2008], which has typically too much overhead for continuous application in large software systems.

Resource-oriented metrics are not presented here in detail, as this thesis focuses on analyzing measurements on software application layer and abstracts from resource consumption details. This abstraction has the advantages that no hardware resource monitoring is required, and that the timing behavior models are smaller and simpler.

2.1.1.2. Throughput and Utilization

Throughput can be defined as the number of requests processed within some time interval [Smith and Williams, 2001b, p.3, and Jain, 1991, p.38]. While a system's user might be primarily interested in a short response time, the system owner might aim for high throughput to minimize costs per usage [Cheng, 2008].

The *utilization* of a resource is the fraction of time in which the resource is busy providing service [Jain, 1991, p.39 and Menascé and Almeida, 2001, p. 109]. An example is CPU utilization. Resource utilization is, for instance, used in sizing, i.e., selecting a suitable amount of hardware for a particular system and workload. SAP typically targets an average CPU utilization of 65 % [Cheng, 2008].

2.1.1.3. Workload, Workload Intensity, and Scalability

The *workload* of a system is the set of all inputs received from the environment [Menascé and Almeida, 2001, p. 205]. Workload can be structured into workload intensity and individual request characteristics (compare [Sabetta and Koziolek, 2008]).

Workload intensity is the *amount* of usage during a time period. In this context, usage may be requests [Smith and Williams, 2001a, p. 139], transactions, processes, or customers [Menascé and Almeida, 2001, p. 502]. Examples for workload intensity metrics are arrival rate, request arrival burstiness, inter-arrival time [Jain, 1991], the percentage of saturation

throughput [Stewart and Shen, 2005], and the number of user requests within a system.

Individual request characteristics are timing behavior relevant characteristics that are specific to a class of requests, such as particular request types, the shape of the corresponding trace, the size and values of operation call parameters (e.g., Koziolek et al. [2008]), and its service demands. Such characteristics can be valuable information for a timing behavior analysis: for instance, software that computes a digital signature for files would have typically larger response times for videos than for images, as images are typically smaller than videos. The *service demand* is defined as the total amount (of time or other resource usage metric) that a request or operation execution requires from a particular resource [Menascé and Almeida, 2001, p.69, and Smith and Williams, 2001a, p. 139]. Note, the above definition of individual request characteristic is more general than the service demand (as for instance defined by [Sabetta and Koziolek, 2008]), because it includes general characteristics of a request type and not only its resource requirements.

Scalability is the ability of a system to meet its response time or throughput objectives as the workload intensity increases [Smith and Williams, 2001b, p. 5]. Alternatively, scalability can be considered as a system's ability to be successfully sized for different workloads [Cheng, 2008; Marquard and Götz, 2008].

2.1.2. Timing Behavior Modeling and Analysis Methods

Herzog [2000] divides software systems into *real-time systems* and *resource-sharing systems*: the timing behavior of real-time systems is usually analyzed using deterministic models (e.g., worst case models) with fixed time intervals, while resource-sharing systems require stochastic timing models due to contention and non-deterministic scheduling. The emphasis in this thesis is on resource-sharing systems (e.g., enterprise software systems, and Web applications).

Furthermore, performance analysis techniques can be distinguished based on whether they are measurement-based techniques, analytical approaches, or use simulation [Woodside, 2008]. These classes overlap and are combined in practice, as for instance the parameters of an analytical model may be determined by measurement. Measurement techniques are typically data-centric, while the analytical and simulation approaches are model-centric, i.e., abstraction is used to capture the essence of a systems performance [Woodside, 2008].

Major measurement-based techniques are workload characterization, timing behavior monitoring, and monitoring data analysis. Classes of monitoring data analysis approaches are, for instance, bottleneck identification, anomaly detection, and performance tuning. It is a classical measurement-based analysis to break down end-to-end response times into component response times [Cheng, 2008].

Analytical performance analysis approaches focus on typical performance models (for non-real-time systems), such as queueing networks, layered queueing networks [Rolia and Sevcik, 1995; Franks et al., 2009], stochastic Petri nets, and Stochastic Process Algebra [Herzog, 2000]. These models can be solved analytically for many purposes such as predicting the mean response time for different design alternatives, and to determine bottleneck resources. Queueing networks and layered queueing networks are formalisms to model performance relevant (physical and logical) resources and their usage within a system. Analytical performance analysis models can be difficulty to apply (e.g., Thereska and Ganger [2008]) because of high resource consumption. Many analytical approaches make strong assumptions and strong simplifications, such as the exponential distribution of response times, to be mathematically tractable. Many recent model-based performance prediction methods are based on queueing networks [Balsamo et al., 2004].

Simulation techniques provide an alternative to solving performance models analytically, as analytical models can easily become mathematically intractable if common model simplifications cannot be made. Simulation methods can be more adequate than analytical methods to evaluate design alternatives [Bause et al., 2008], as simulation models can be extended to include specific details.

Performance analysis techniques can be further distinguished based on the phase of the software life cycle in which they are primarily applied, although there are some approaches that span all phases. Some approaches address the early design time of a system, while other techniques, such as the approach presented in this thesis, are performance analysis approaches, which are used after the software is released in its production environment.

A typical example of a technique used during early design is performance prediction. *Performance prediction* estimates performance metrics, such as average response time, throughput, and resource utilization for an expected workload (e.g., by Balsamo et al., 2004, and Becker et al., 2007). Liu et al. [2005] reported a low performance prediction error of often less than 10 % for a component-based (Java Enterprise Edition)

software application.

Much later in the software life cycle than prediction is profiling. *Profiling* refers to approaches that record execution profiles for quantifying the fraction of time and resources used in parts of a software program (e.g., Smith and Williams, 2001b, p. 312 and Spivey, 2004). Profiling methods are for instance presented by Viswanathan and Liang [2000], Graham et al. [1982], Xie and Notkin [2002], and Hauswirth et al. [2004].

Examples for profiling with several tools such as gprof, Google's perftools, Valgrind, and Java's HPROF are in Appendix D on Page 171. The performance evaluation technique software *regression benchmarking* [Bulej et al., 2005] aims at detecting regressions in software performance between different versions of a software system.

More comprehensive reviews of performance analysis topics, such as timing behavior modeling, measurement, and analysis are provided by Jain [1991]; Herzog [2000]; Smith and Williams [2001b]; Balsamo et al. [2004]; van Hoorn et al. [2009]; and Sabetta and Koziolek [2008].

2.1.3. Characteristics of Software Timing Behavior

Modern software systems have complex timing behavior. It is challenging to characterize the timing behavior of typical enterprise Java software applications and multi-user Web applications [Lashari and Srinivas, 2003; Stewart and Shen, 2005]. Technologies such as garbage collection and just-in-time compilation make it difficult to determine and understand performance down to hardware resources [Hauswirth et al., 2004]. Enterprise software applications are usually deployed in middleware environments that do not provide real-time properties and show non-trivial scheduling and queueing behavior. Such software applications are multi-user systems with concurrent and heterogeneous user requests competing for computational resources (see Herzog, 2000). The resulting response time distributions often show high variance and do not follow simple distribution families, such as exponential or normal distributions (e.g., Mielke, 2006; van Hoorn, 2007; Sambasivan et al., 2011).

Statistical properties of the timing behavior of real world software systems are not well-researched: For instance, Mielke [2006] states that there is a lack of detailed knowledge of relevant statistical timing behavior properties for Enterprise Resource Planning systems, and Harchol-Balter [2008] explains that relatively little is known about the timing behavior of server farms.

Response time measurements often are well-described by log-normal distributions or Pareto distributions (e.g., Mielke, 2006). Timing behavior distributions consisting of right-skewed (i.e., mode $<$ median $<$ mean), long and heavy-tailed distribution elements have been reported for Java-based Web applications [van Hoorn, 2007]. TELNET connection durations showed response times (data transfer times included) that well-fit log-normal distributions [Paxson, 1994]. Mielke [2006] demonstrated that log-normal and Generalized Lambda distributions fit well to the response times for Enterprise Resource Planning systems. Our measurements [van Hoorn, 2007] suggested that operation response times are well described with log-normal distributions. Response time distributions can also be described using Generalized Lambda Distributions, which actually can be parameterized to follow the shape of log-normal distributions [Au-Yeung et al., 2004]. An example for the fitting of a log-normal distribution (2 and 3 parameters) to measurement data, here from a simple Java Servlet, is displayed in Figure 2.4. In this example, the 3-parameter log-normal distribution fits better to this right-skewed data than the 2-parameter log-normal distribution, especially since it provides the possibility of a right-shift to address that the minimum response time seems to be at 1 second.

Furthermore, empirical results show that workload is non-stationary (e.g., mobile network traffic [D'Alconzo et al., 2009]) for instance due to time-of-day variations with strong 24-hour seasonality. This suggests that it can be more suitable to compare in anomaly detection measurements with measurements of another day with the same time, instead of comparing a measurement to its previous measurement of the same day.

2.1.3.1. Heavy tails and high variance

Workload and timing behavior distributions can have so-called heavy tails [Menascé and Almeida, 2001; van Hoorn, 2007, p. 168]. *Heavy-tailed distributions* have many observations in the so-called tails (i.e., the left or right end of a distribution) [Field et al., 2012], and sample data is "characterized by the presence of observations with very large magnitudes, a phenomenon often referred to as high variability" [Kogon and Williams, 1998]. Additional mathematical definitions on heavy-tailed distributions can be found in Harchol-Balter [2002], and in Crovella et al. [1998]. The simplest heavy-tailed distribution is the Pareto distribution [Crovella and Bestavros, 1997]. Strict definitions do not consider the log-normal distributions as being heavy-tailed; however, log-normal distributions show

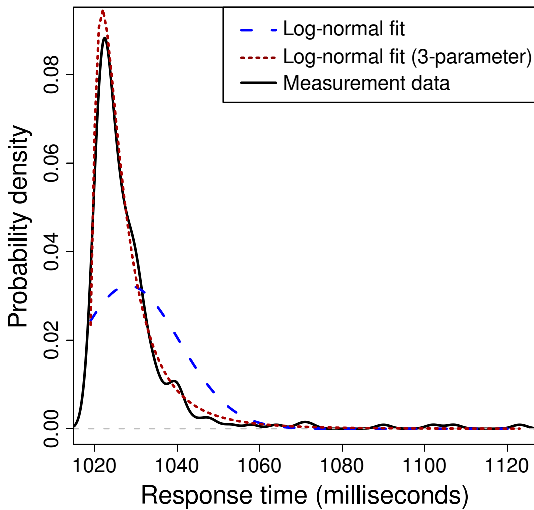


Figure 2.4.: Probability density estimate for response time measurements and two fitted distributions.

similar characteristics in sample data that are typical for heavy-tailed distributions [Crovella and Bestavros, 1997]. Typical characteristics are for instance:

- Very high variance, especially compared with exponential distributions; Sample values can span over many orders of magnitude [Harchol-Balter, 2002]
- “Extreme” variability in workload of Web-based systems [Menascé and Almeida, 2001]
- Many small and some very large values, but far less medium sized values than for exponential distributions [Crovella et al., 1998]

Other examples for heavy-tailed distributions in computer systems are provided by empirical studies on FTP data transfer size and duration [Paxson, 1994], Internet file transfer times [Downey, 2001], computer system tasks sizes [Harchol-Balter, 2002], and HTTP response size [Menascé and Almeida, 2001, p. 167].

Workload with heavy-tailed distributions can results in high variability in timing behavior or heavy-tailed response times: For instance, Crovella

and Bestavros [1997] and Vallamsetty et al. [2003] provid empirical evidence for heavy-tailed response time distributions of Web applications and e-commerce systems.

Heavy-tailed service request sizes lead to very long service times in a typical single server system [Harchol-Balter, 2002]. Furthermore, right-skewed heavy-tailed service size distributions can lead to a non-negligible probability for the occurrence of very long jobs and if a system's scheduler cannot prevent that such long jobs block many small jobs, then the average response time will be quite high (see Psounis et al., 2005). Our measurements in a Java Web application showed right-skewed software operation response time distributions with heavy tails [van Hoorn, 2007].

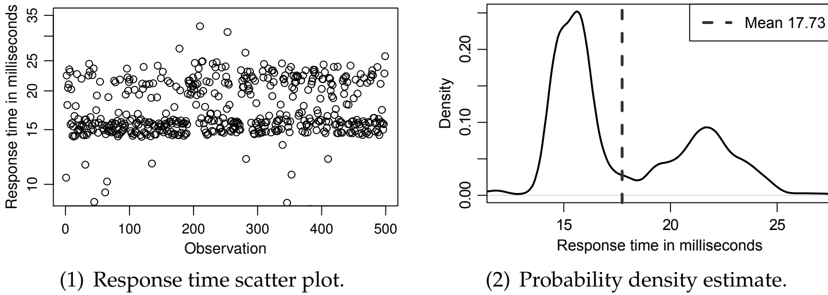
2.1.3.2. Multimodal distributions

Response time distributions can be *multimodal*, i.e., the probability density function has more than one local maximum [Simon, 2006, p. 473]. An example for this is provided in Figure 2.5, which shows a response time distribution monitored in the DaCapo Eclipse Benchmark [Blackburn et al., 2006] as described in Rohr et al. [2008a]. Multimodal distributions are problematic for many timing behavior analysis approaches, as these cannot accurately be described by a single simple distribution [Rohr et al., 2008a]. Furthermore, multimodal distributions may have mean values that are a bad representative for a typical sample value. For instance, the mean in Figure 2.5(2) is located at a point of relatively low probability density. An anomaly detection approach that assumes the mean value as good representative for "normal" behavior is unsuitable in this case.

Other empirical examples for multimodality can be found in the work of Bulej et al. [2005], Mielke [2006], and van Hoorn [2007]. Bulej et al. [2005] observed multimodal response time distributions in different versions of CORBA middleware and used the term "cluster" for each group of similar response times. These authors illustrate that clusters in timing behavior measurements reduce the potential to detect changes in the timing behavior of software.

2.1.4. Influences to Software Timing Behavior

In the following, influences to timing behavior are presented. The timing behavior depends on many variables such as a system's design, code, and execution environment [Woodside et al., 2007]. Potential influences to



(1) Response time scatter plot. (2) Probability density estimate.

Figure 2.5.: Multimodality of response times. (See Rohr et al. [2008a] for details.)

timing behavior can be on all system layers, such as hardware-, network-, operating system-, middleware-, and software application layer. Timing behavior propagates through the system by control flow [Kapova et al., 2010]. Some of the influences to timing behavior are dynamic, i.e., vary often during runtime, while others, such as a system's general design and hardware, can be considered static as they do not change frequently and not during runtime. In the following, major categories of influences are summarized. Especially the influence of workload intensity to timing behavior is relevant for this thesis.

2.1.4.1. Middleware, Operating System, and Execution Environment

The performance of component-based software systems can be strongly determined by the implementation of the middleware that hosts the components and its configuration [Gorton and Liu, 2003; Liu et al., 2005]. For instance, enterprise software systems are middleware-intensive, as the structure and behavior of these systems are significantly demerited by its middleware [Giesecke, 2008]. A significant part of the total execution time is spent in the middleware, depending on the particular amount of middleware service usage; for instance, 15–30 % of the execution time was spent in the middleware for the benchmark scenarios used by Lashari and Srinivas [2003]. The operating system itself is also a timing behavior influence: the memory management of operating systems (physical page allocation) is known to cause a non-deterministic varying influence on software timing behavior [Hocko and Kalibera, 2010].

Many software applications are executed in so-called managed execution environments, such as Java or C# software. Lashari and Srinivas [2003] reported that it is very challenging to characterize the runtime behavior of Java software applications, as the managed execution environment spends much time for other activities than bytecode execution, such as dynamic "just-in-time" compilation, native code execution, and execution of services of the operating system and execution environment. Dynamic compilation typically uses non-deterministic optimization techniques resulting in non-deterministic timing behavior [Georges et al., 2008; Hocko and Kalibera, 2010]. Performance issues of automatic memory management (garbage collection) are summarized in Section 2.1.4.6, on Page 22.

The middleware, the execution environment (e.g., Java virtual machine), and the operating system all contain scheduling algorithms. Depending on the scheduling strategy, a scheduler might optimize the mean response times at the cost of variance [Sambasivan et al., 2011].

2.1.4.2. Software Architecture and Software Implementation

The architecture of a software system, i.e., its fundamental organization and design principles [ISO/IEC 42010, 2006], and its implementation are key performance factors of a software system. Examples for performance-related design decisions in the software architecture are the organization of data access and how the system is structured into interacting components. On implementation level, scheduling algorithms, sort algorithms, and principles and data structure of concurrent programming are examples for well-studied performance related topics. Typical software performance patterns and anti-patterns can be found in Smith and Williams [2001b,a, 2002].

2.1.4.3. State

Performance relevant state elements are for example cache content, data state, software application state, and the state of the operating system. A categorization of state and its relation to performance is presented in detail by Kapova et al. [2010].

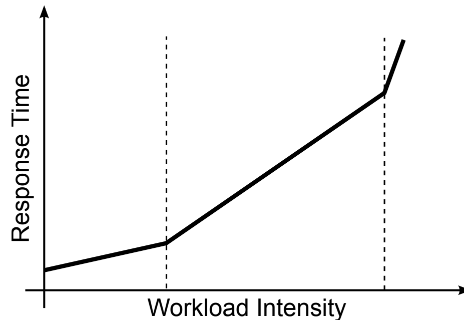


Figure 2.6.: Typical relation between response times and workload intensity. Illustration based on [Jain, 1991].

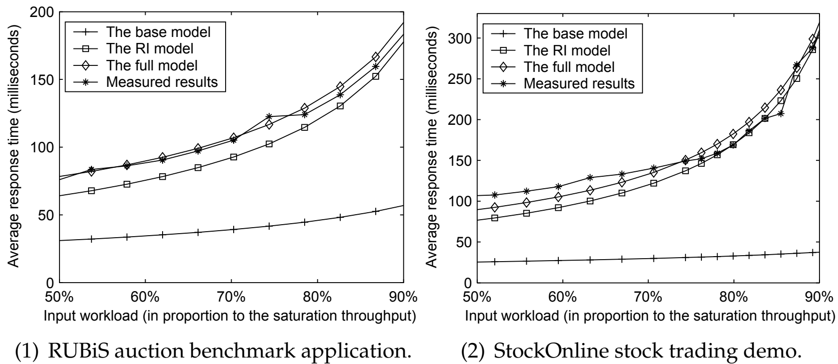
2.1.4.4. Individual Request Characteristics

Individual request characteristics (definition on Page 13) can have a strong influence on software timing behavior. Therefore, some performance analysis approaches explicitly consider request characteristics. Examples for request characteristics that are explicitly addressed are parameter values and value sizes [Koziolek et al., 2008], caller identity of a request [Graham et al., 1982], and resource consumption metrics, such as the number of accesses to persistence layers, CPU consumption, memory consumption and the number of network accesses [Cheng, 2008; Menascé and Almeida, 2001].

2.1.4.5. Workload Intensity and Timing Behavior

Response times often increase by increasing workload intensity (e.g., in terms of the number of requests per minute; see definition on Page 12). The typical relation between workload intensity and response times is illustrated in Figure 2.6 [Jain, 1991]: Up to some first workload intensity (left dotted line) the response time does not increase significantly while the workload increases. After a second workload intensity level (right dotted line) the response time increases rapidly. Already Scherr [1965] reported a similar relation between response times and the number of online users in multi-user systems [Herzog, 2000].

The line “measured results” in Figure 2.7 shows the relation between workload intensity and average response times of two distributed JavaEE applications [Stewart and Shen, 2005]. The average response times are



(1) RUBiS auction benchmark application. (2) StockOnline stock trading demo.

Figure 2.7.: Measured and predicted average response times in relation to workload intensity. Images from Stewart and Shen [2005].

from measurements and three different prediction models, and follow approximately the shape described by Figure 2.6.

Examples for research approaches that explicitly include workload intensity in the evaluation of timing behavior measurements are by Maxion [1990], Smith and Williams [2001a], and Zhang et al. [2007].

2.1.4.6. Automatic memory management (Garbage Collection) and Timing Behavior

A major influence to performance and responsiveness for software systems implemented in programming languages, such as Java and C# is automatic memory management, also known as garbage collection [Printezis, 2004; Meier, 2007; Cheng, 2008; Blackburn et al., 2004].

Many garbage collectors (e.g., collectors released together with Java versions 1.4, 5, and 6) are organized in garbage collection runs. Some of the runs (minor collections) have shorter execution times than other less frequent runs (major collections) [Printezis, 2004]. The collection runs are not completely concurrent to normal program execution (e.g., Java SE 5's Concurrent Mark-Sweep Collector, Serial Collector, and Parallel Collector), so that the application is paused [Sun Microsystems, 2006]. Worst-case pause times of several seconds can occur [Printezis, 2004]. The Garbage-First (G1) collector of Java 7 and Java 8 aims to limit pause times to below 0.5 seconds [Oracle, 2013] and allows setting a not-guaranteed

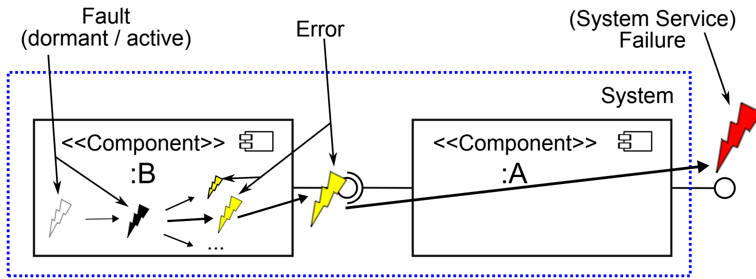


Figure 2.8.: Error propagation example in a component-based software system.

maximum pause time target [Oracle, 2014].

Own measurements on garbage collection activity during the case studies of this thesis are presented in Appendix E.

2.2. Software Faults and Dependability of Software Systems

The foundations on software faults and fault localization that are relevant for this thesis are part of the research fields on dependability and on fault tolerance. In particular the foundational concepts and terminology related to faults, errors and failures are presented in the following. Additionally, empirical research results on software faults are summarized.

2.2.1. Faults, Errors, Failures, and Availability

Central terms in the field of dependable and fault tolerant computing are fault, error, and failure. These terms are sometimes used synonymously in the literature, but are carefully distinguished here to not confuse causes and symptoms of failures. The most common definitions are provided by Avižienis et al. [2004]: System *failures* are violations of the corresponding system's specification or of what the system is intended to do. *Faults* are root causes of failures. Faults create errors upon activation. *Errors* are invalid system states that can cause a system failure by propagating to the outside of the system, e.g., by becoming visible to the system users. Figure 2.8 shows an example for the so-called error propagation between a fault and a corresponding failure.

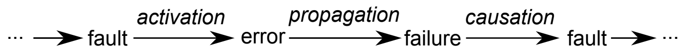


Figure 2.9.: Fundamental chain of dependability threats [Avižienis et al., 2004].

These definitions of fault, error, and failure always refer to a system boundary, which can be considered a frontier between the system and its environment. This boundary depends on a viewpoint. For the viewpoint of a system interacting with its users, a failure is an error that becomes visible to the user, as illustrated in Figure 2.8. In a component-based system, the terms fault, error, and failure can also refer to the context of a component. From the viewpoint of a component, an error becomes a failure if it propagates to the component's outside, and violates the component's specification or what it is intended to do. For instance, the symbol between the components A and B in Figure 2.8 can be considered a failure from the perspective of component B.

The causality relationship between faults, errors, and failures is illustrated in Figure 2.9 [Avižienis et al., 2004]. The causality relation is open ended at both sides, as other causal fault-error-failure relationships can exist.

Software is pure design [Littlewood and Strigini, 2000] and therefore, *software faults* are design faults [Musa et al., 1987, p. 7, and Musa, 2004, p.36]. Software faults are introduced into software during any phase of software development, such as specification, programming, or installation [Laprie et al., 1995, p. 48]. In practice, it is not feasible to develop complex software systems that are free of faults [Eusgeld et al., 2008].

Many software dependability approaches are adopted from hardware reliability approaches [Lyu, 2007]. Littlewood and Strigini [2000] state that hardware systems failures tend to be dominated by random physical failures of components. A central principle of (non-software) reliability engineering is copy-redundancy [Eusgeld et al., 2008]: critical system components are identically replicated such that the system does not fail if single components fail. For instance, cooling facilities of nuclear power plants have to be replicated several times to reduce the probability of a complete system failure. While copy-redundancy is effective against physical deterioration, it is ineffective against design faults [Eusgeld et al., 2008]. Hence, many such hardware reliability methods cannot be used to address software design faults (or hardware design faults). Another particular characteristic of (software) design faults is that a successful repair fixes a fault for all time [Musa et al., 1987, p. 7].

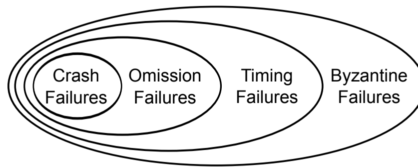


Figure 2.10.: Failure class hierarchy based on Cristian et al. [1995].

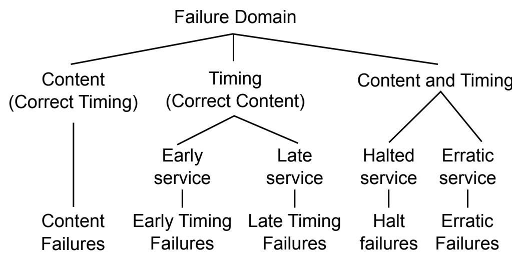


Figure 2.11.: Failure modes with respect to the failure domain viewpoint [Avižienis et al., 2004].

Failures can be categorized, such as displayed in Figures 2.10 and 2.11: Cristian et al. [1995] (Figure 2.10) define failure classes according to how efficient they can be compensated by fault tolerance mechanisms. A mechanism that is able to tolerate failures of a higher class is able to tolerate failures of a lower class as well. The most restricted class (i.e., "weakest") in this model are *Crash Failures*, which are those failures that occur when a component or system prematurely halts. Crash Failures is a proper subclass of *Timing Failures* consisting of cases in which a component answers a request too late, never or too early. Another categorization is provided by Avižienis et al. [2004], illustrated in Figure 2.11. It distinguishes failures based on whether the content or timing behavior of a system and its output deviates from the expected behavior.

Availability is commonly defined as "the average (over time) probability that a system or a capability of a system is currently functional in a specified environment" [Musa et al., 1987; Musa, 2004]. Alternatively, it can be expressed as relation between the mean time to failure (MTTF) (also called mean time between failure) and the mean time to repair (MTTR) [Musa et al., 1987; Gray, 1986]:

$$\text{Availability} = \frac{\mathrm{MTTF}}{\mathrm{MTTF} + \mathrm{MTTR}} \tag{2.1}$$

Therefore, availability can be improved through the reduction of the mean time to repair and by increasing the time to a failure. The MTTR can be further divided into separate times for failure detection, fault localization, and fault removal [Giesecke et al., 2006]. Reducing any of these times also improves availability.

2.2.2. Empirical Research on Failures and Faults in Software Systems

In the following, empirical research on software faults is summarized. In the context of this thesis, a more detailed overview was published in Ploski et al. [2007].

Software faults are a significant cause of system and platform failures. Küng and Krause [2007] and Gray [1986] reported empirical results that more software faults caused system failures than hardware faults. Both studies also show that administration faults cause many failures; Many administration faults can be considered software faults, as the faults are present as configuration files. Schroeder and Gibson [2006] evaluated failure data from several high-performance computing platforms, which were used mainly for scientific computing. In this study, node crash failures (special subclass of system failures) were caused in about 20 % of the cases by software. Another study reported that more than 70 % of the outages in several network and communication systems in the telecommunication domain were software related [Network Reliability Council, 1993; Lai and Wang, 1995].

Empirical research on software faults is often connected with research on fault categorization, as there is no generally accepted categorization schema. This lack of a generally accepted categorization schema was reported by Marick [1990]. Knuth [1989] provided one of the first studies on software faults. It was created over a period of 10 years during the development of the typesetting system TEX. Knuth [1989] categorized faults into classes such as “data structure debacle”, “forgotten function”, and “trivial typo”. A later repeated categorization of these faults resulted in 53 % missing entities and 41 % incorrect entities [DeMillo and Mathur, 1995]. Dallmeier [2010] analyzed a large bug report database for the software tool Eclipse. A fraction of the faults was manually categorized. The most common fault categories were related to missing or wrong null checks, and to incorrect boolean and arithmetical expressions.

Some software faults are difficult to localize. So called Heisenbugs

are known to be hard to find and remain for a long period of time even in extensively tested production systems [Gray, 1986; Eisenstadt, 1997]. Heisenbugs seem to temporally disappear during the search for the bug (e.g., when debugging tools are active) [Bourne, 2004]. A collection of faults related to software vulnerabilities can be found in [CWE/SANS 2009, 2009]; Three categories for what were considered the most 25 significant security-related faults were identified: insecure component interaction (e.g., improper input validation), risky resource management (e.g., unchecked string length in C), and porous defenses (misuse of defensive techniques, e.g., hard-coded passwords). Grottke et al. [2010] categorized the software faults from 18 space missions into Bohrbugs (61 %) and Mandelbugs (37 %). Bohrbugs are easy to diagnose and easy to reactivate, in contrast to Mandelbugs, which cannot be systematically reactivated as they only cause under complex conditions a failure (e.g., time)[Grottke et al., 2010].

The empirical studies of Ostrand and Weyuker [2002], Fenton and Ohlsson [2000], and Adams [1984], and Nagappan et al. [2006] addressed how faults are distributed among the software's source files and modules. The results suggested that most faults are within a small part of a software's source code and that fault densities are not increasing by source code file size and module size.

2.3. Automatic Fault Localization for Software Systems

In practice, it is not feasible to develop complex software systems that are free of faults [Eusgeld et al., 2008]. After a fault causes a failure, it is often not considered sufficient to only address the failure, for instance by automatically rebooting parts of the system [Candea et al., 2004]. Moreover, it is desirable to prevent that the same fault is being activated again or a problem is persistent and repair is needed to return to normal operation. One part of fault removal is *failure diagnosis*, which identifies a failure's or error's corresponding fault in terms of type and localization [Avižienis et al., 2004, p. 25].

Fault localization and the synonym *fault isolation* can be defined as identification of the location (e.g., a component) of a failure's root cause (compare with Bocaniala and Palade, 2006, p. 6). Others define these terms less narrow and include the detection (Isermann and Ballé, 1997,

and [Isermann, 2006, p. 413]) or the identification of the type of the fault [Steinder and Sethi, 2004]. The term fault localization can be used to refer to the localization of faulty statements in a software program (also denoted debugging), and it can refer to the localization of faults in running software system (that contains soft- and hardware). The focus of this thesis is on the second meaning.

Fault localization approaches differ in result granularity: some approaches point to single statements within the software's source code (e.g., Tarantula Jones et al. [2002]; Jones and Harrold [2005]) while other refer to larger blocks such as modules or components (e.g., Yilmaz et al., 2008 and Kiciman, 2005).

A large number of approaches for the localization of faults in software programs exists, which are surveyed for instance by Wong and Debroy [2009]. Program-spectrum-based approaches (e.g., Reps et al. [1997], Chen et al. [2002], and Tarantula [Jones et al., 2002; Jones and Harrold, 2005]) compare which parts of a program are active during passed runs with which are active during failed runs [Abreu et al., 2007]. For instance, a statement is a fault candidate, if it is executed in every failed run and not in any passed run. Program-state-based approaches, such as Zeller [2002]'s Delta Debugging analyze and modify variable values in test runs to localize bugs [Wong and Debroy, 2009].

Automatic failure diagnosis can be motivated by pointing out that manual failure diagnosis can be very time-consuming and error-prone [Marwede et al., 2009]. The definition of availability on Page 25 shows that a reduction of failure diagnosis time can improve availability. Katzela and Schwartz [1995] argue that human operators are not efficient for failure diagnosis in communication networks, because of the "amount and complexity of status information generated by a fault, the increasing size of and complexity of the network and the limited processing capacity of a human operator". Manual fault localization in software may result in a search in space across program state to find infected variables, and a search in time over millions of program states [Cleve and Zeller, 2005]. Debugging is especially a challenge if there is a large temporal and spatial distance between causes and symptoms and for failures that are hard to repeat during debugging (e.g., Heisenbugs) [Eisenstadt, 1997; Bourne, 2004].

Fault localization can be challenging because of the propagation process between faults, errors, and failures (see Section 2.2.1 and Figure 2.9 on 24). Some faults lead to correlated failure manifestations in the entire system [Pertet and Narasimhan, 2005]. Furthermore, symptoms can be

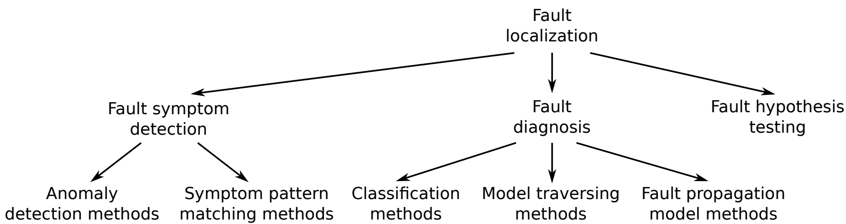


Figure 2.12.: Fault localization in software systems: concepts and techniques.

ambiguous, inconsistent, redundant, and incomplete and may result from unrelated faults that happen at the same time [Steinder and Sethi, 2004; Katzela and Schwartz, 1995]. Additionally, a combination of undesired conditions may lead to a failure, and this combination is considered as the fault of the failure [Musa et al., 1987].

A basic and relatively old general method to localize faults is *limit checking*. It dates back to the application in technical systems at the end of the 19th century [Isermann, 2006, p. 8]. In limit checking, *thresholds* are define as upper and lower limits on monitorable attributes. A violation of the thresholds provides a signal corresponding to a particular location. A classical way to determine thresholds goes back to Shewhart [1931]'s foundational work on quality control: an upper and lower control limit are defined by a range of three standard deviations around the mean value of historical observations. There are new approaches based on control charts: for instance the work of Amin et al. [2012] uses modern control chart techniques for the early detection of quality of service violations. This can be used as trigger to automatically adapt the system to achieve a high quality of service. Another basic approach is *diagnostic checking*, which consists of regular status testing of components: A component is tested with input parameters for which the correct output is known to detect the presence of an error [Lee and Anderson, 1990, p.191].

We structure fault localization in software systems as illustrated in Figure 2.12. This structure is combined from the three frequently-cited sources listed in Table 2.1. The terminology has been adapted to the standard terminology from the fault tolerance domain of Avižienis et al. [2004] (see Section 2.2), which more clearly distinguishes between faults and failures. The first layer of Figure 2.12 distinguished three steps in fault localization:

- *Fault symptom detection* identifies symptoms and indicators caused

Table 2.1.: Categorization schemes related to fault localization.

Domain	Survey	Categories	Research Focus
Technical systems	[Isermann, 2006]	Fault detection and fault diagnosis	Detection and diagnosis
Computer networks	[Steinder and Sethi, 2004]	AI Techniques (incl. expert systems); Model Traversing; Fault Propagation Models	Diagnosis
Intrusion detection	[Axelsson, 2000]	Anomaly detection; Misuse pattern detection	Detection

by a fault. Sometimes, these symptoms can clearly be separated from normal system behavior. However, especially fault tolerance mechanisms can mask symptoms as a side effect [Maxion, 1990]. A failure message is an easily detectable fault symptom. Nevertheless, more information may be required for a particular fault localization than just the externally visible failures [Steinder and Sethi, 2004; Isermann, 2006].

- *Fault diagnosis* determines a fault based on the symptoms determined in fault diagnosis [Isermann, 2006]. For instance, event correlation techniques (e.g., Gruschke, 1998a; Tiffany, 2002) are used to determine faults from large amounts symptoms, status information, events, and general monitoring data.

- *Fault hypothesis testing* evaluates whether a fault hypothesis is true or false [Steinder and Sethi, 2004].

Fault symptom detection can be categorized into anomaly detection methods and symptom pattern matching methods. This is analogous to how intrusion detection literature, such as Kumar [1995], LaPadula [1999], Axelsson [2000], and Patcha and Park [2007] structure the detection task.

- *Anomaly detection* methods aim to detect anomalies in observations [Axelsson, 2000]. For instance, a set of historic observations is considered to be of normal behavior; these observations are used to create a normal behavior profile, which than is used for assessing whether new observations are normal or not. A weaknesses of anomaly detection is that it can lead to high false alarm rates, as

anomalies do not necessarily correspond to the desired detection target (e.g., intrusions, or faults) [Axelsson, 2000]. High false alarm rates can result in reduced attention to detections [Ghosh et al., 1998]. More foundations on anomaly detection are presented in Section 2.4, on Page 32.

- *Symptom pattern matching* methods, also denoted signature-based detection or misuse pattern detection in the intrusion detection domain, use specifications of correct and/or incorrect system state or behavior to identify the state transition that turns correct into incorrect system state (cp. Axelsson, 2000). The component that initialized this transition is considered to be responsible for the failure (e.g., Kumar, 1995). Signature-based methods can only detect known faults or intrusion [Kruegel and Vigna, 2003]. This is problematic for dealing with intrusions such as the Internet worm W32.SQLExp.Worm. Already ten minutes after its release, it had compromised 90 % of all vulnerable systems [Farshchi, 2003].

Fault diagnosis can be further structured into techniques analog to a categorization for diagnosis from the domain of computer networks [Steinder and Sethi, 2004; Tiffany, 2002]:

- *Fault classification* methods are dominated by expert system approaches that use typical classification techniques from artificial intelligence research. A typical approach would use a knowledge base created from human experience or understanding represented as if-then rules [Steinder and Sethi, 2004].

- *Model traversing* analyzes a formal system model of behavioral or structural relationships to determine faults for observed symptoms [Steinder and Sethi, 2004]. Examples for these models are software architecture models, such as call dependency graphs. A summary of model traversing techniques is provided by Steinder and Sethi [2004].

- *Fault propagation model* methods, also denoted inference methods in the technical systems domain, explicitly represent relations between fault and symptoms. An example for such models are graph-based models, such as fault trees [Steinder and Sethi, 2004; Isermann, 2006].

2.4. Anomaly Detection

This section presents some basic anomaly detection concepts that are relevant for Chapter 3. Comprehensive surveys can be found in the work of Chandola et al. [2009], Patcha and Park [2007], and Hodge and Austin [2004].

2.4.1. What are Anomalies?

Anomalies can be defined as objects or observations that are unusual or in some way inconsistent with most other objects of a data set [Tan et al., 2006, p. 651]. The term *outlier* is sometimes used synonymously [Chandola et al., 2009]. A common definition for the term outlier is by Hawkins [1980], defining it an observation which deviates so much from other observations that it suggests that is was "generated by a different mechanism". Grubbs [1956] defines outliers within a sample as observations appearing to "deviate markedly from other members of the sample in which it occurs". In other words, anomalies can be considered deviations from the normal. What is considered normal may be manually specified, or learned from training data. In some cases it is the task to "clean" data from outliers as preprocessing (e.g., Jain [1991, p. 19]), while in other cases, it is the target to study especially the outliers (e.g., for intrusion detection or failure diagnosis).

Three types of anomalies can be distinguished [Munoz-Garcia et al., 1990; Tan et al., 2006]: Atypical observations, erroneous observations, and observations from a different class. Atypical observations are the consequence of inherent variability, i.e., such observations are inconsistent with most other observations of a data set as natural result of the characteristics of the observed population. Erroneous observations can result from measurement errors, experiment design errors, and data processing errors. Observations that correspond to another class of objects than the class of the "normal" objects may also differ in their characteristics. For instance, immune system data (e.g., number of white blood particles) of a sick patient might be atypical and inconsistent to those measurements of patients that belong to the class of healthy people [Munoz-Garcia et al., 1990; Tan et al., 2006].

Timing behavior anomalies could be defined as deviations from normal timing behavior. Timing behavior anomalies can have many possible reasons. It has been observed that also faults often have an influence on timing behavior [Kao et al., 1993]. Other examples for fault localiza-

tion based on timing behavior are provided by Yilmaz et al. [2008] and Agarwal et al. [2004].

2.4.2. Training Data Labels

Many anomaly detection approaches learn what can be considered normal or anomalous from training data, instead of requiring a formal specification or thresholds. Training-data-based approaches can be distinguished by the existence of training data labels [Hodge and Austin, 2004; Chandola et al., 2009]:

- *Supervised anomaly detection* requires training data with instances labeled as anomalous and normal. A basic method is to compare new observations to both the models obtained from training data, to decide which model fits better [Chandola et al., 2009]. In many domains, it is difficult to get accurate and representative anomaly data for training, especially if new types of anomalies can occur [Chandola et al., 2009].
- *Semi-supervised anomaly detection* only requires training data for either anomalies or normal observations. Most approaches of this category use training data with instances labeled for being normal, as accurate and representative training data for normal observations can typically easier be obtained than for anomalies [Chandola et al., 2009; Hodge and Austin, 2004].
- *Unsupervised anomaly detection* does not require training data for neither anomalies nor normal observations [Chandola et al., 2009; Hodge and Austin, 2004]. Instead, those observations are considered anomalies that differ from most observations within a data set. It is implicitly assumed that most observations within this data set are normal [Chandola et al., 2009]. As mentioned above, the classical approach of Shewhart [1931] assumes that normal observations are within the range of three (or more) standard deviations of the data around the data's mean value.

2.4.3. Anomaly Detection Result

Typically, anomaly detection approaches produce either a binary result or a score [Chandola et al., 2009]. A binary result is "positive" (observation is considered anomalous) or "negative" (observation is considered

normal). Non-binary anomaly detection provides scores, for instance from $[0, 1]$, can express the confidence that an observation is considered an anomaly. Advantages of scores are for instance that scores can be ordered [Chandola et al., 2009] and scores provide more possibilities for aggregation.

Binary results can be correct or incorrect. There are two types of correct and two types of incorrect results for binary anomaly detectors based on whether a true anomaly was presented to the approach (e.g., [Fawcett, 2006]):

- Correct: a positive output in presence of an anomaly ("true-positive") and a negative output when no anomaly is present ("true-negative").
- Incorrect: a positive output in absence of anomalies ("false-positive", *false alarm*) and a negative result in presence of an anomaly ("false-negative").

2.4.4. Anomaly Detection Categories

Anomaly detection techniques can be grouped into the following four partly-overlapping categories [Chandola et al., 2009]:

- *Classification-based techniques* identify one or more classes in training data and test observations against these classes. There are many different methods within this group, such as approaches based on Neural Networks and Support Vector Machines (SVMs) [Chandola et al., 2009]. Classification-based techniques are in particular suitable, if the normal data consists of multiple classes. An example for multi-class anomaly detection is illustrated in Figure 2.13.

- *Nearest-neighbor-based techniques* consider objects to be normal that are close to neighbors. In contrast to classification and clustering techniques, the number of classes or clusters does not need to be known [Chandola et al., 2009]. Some approaches take more than one nearest neighbor into account, as the nearest may be an anomaly itself, as illustrated in Figure 2.13.

- *Clustering-based techniques* assume that normal objects are clustered. Similar to the nearest-neighbor-based approaches, distance measures are used, but in reference to clusters (e.g., a cluster's centroid) instead in reference to neighbors [Chandola et al., 2009]. The two anomalies in Figure 2.13 should be detected by a clustering-based approach, as both anomalies are clearly outside the clusters.

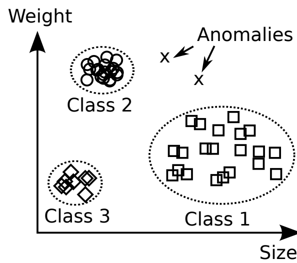


Figure 2.13.: Anomaly detection example with three classes of normal behavior and two characteristics. An anomaly detector for this scenario should consider both weight and size for reliable detection. Image inspired by [Breunig et al., 2000].

- *Statistical-based techniques* detect anomalies based on statistical models. Observations that are considered an anomaly if they do not fit to the statistical model of normal behavior [Chandola et al., 2009]. For example, box plots define anomalies based on interquartile distances [Montgomery and Runger, 2003, p. 207]. A probability density model for Figure 2.13 would have low probability density at the area of the anomalies (if those are not within the training data); therefore the anomalies would be detected.

2.5. Software Application Monitoring

Monitoring supervises, records, analyzes, or verifies the operation of a system or of a component [IEEE Standards Board, 2002]. A monitor "is a tool used to observe the activities on a system" [Waheed and Rover, 1995]. The terms logging, and profiling are often not clearly distinguished from monitoring.

The term *logging* is often used in the context of software logging frameworks. Logging frameworks provide an API to record messages from within a software program. The recorded messages are often stored into standardized log files. Logging can be used to store precise context information about application execution for debugging [Gülcü, 2003]. A focus of logging is debugging [Gülcü, 2003; Kernighan and Pike, 1999], but it can also be used for recording internal software application states or for recording data for performance analysis [Gupta, 2005]. Furthermore,

Layer	Monitoring targets
Business Process	Business process cycle time and rate, ...
Application	Throughput, response time, #users ...
Middleware	Message queue size, active jobs, ...
Execution Environment	Memory consumption, active threads, ...
Operating System	Sytem processes, CPU utilization, ...
Hardware	CPU cycles, network statistics, ...

Figure 2.14.: Software system layers and monitoring targets. Image based on Focke [2006].

logging can be used for monitoring, for instance by providing logs to a service for automatic supervision [Gupta, 2005].

As stated earlier in this Chapter on Page 15 and demonstrated in Appendix D, *profiling* refers to approaches that create execution profiles for quantifying the fraction of time and resources used in parts of a software program [Smith and Williams, 2001b, p. 312]. In contrast to monitoring, it is usually more intrusive and provides deeper insights in the execution of a system. Profiling provides a more detailed view on dynamic behavior which allows developers to identify performance bottlenecks or memory leaks. Software is usually developed, tested and operated in different environments (development, test, production) (see e.g., Dustin, 2002, p.47 and Humble and Farley, 2010). In contrast to monitoring, profiling is classically a technique to be used in the development and in the test environment and not within production environment, as the overhead is usually to large for continuous operation in production [van Hoorn et al., 2009]. It is also mainly a tool for developers, while monitoring is more used by system administrators.

Many different software monitoring frameworks, such as DTrace, InfraRed [Govindraj et al., 2006], and Kieker [van Hoorn et al., 2009] are available to monitor software runtime behavior. Several approaches, such as Dapper [Sigelman et al., 2010], Stardust [Thereska et al., 2006], and Kieker [van Hoorn et al., 2009] support monitoring request traces in distributed systems and are used for problem diagnosis.

Monitoring tools and approaches exist for each system layer. Figure 2.14 illustrates examples for typical monitoring metrics for each layer [Focke, 2006; Woodside, 2008]. Horizontal approaches focus on a particular system layer, while vertical monitoring approaches, such as

the approach of Hauswirth et al. [2004], monitor on multiple layers.

Enterprise software systems can consist of multiple execution environments. Therefore, monitoring technology for enterprise systems should be able to deal with the issue of local clocks in a distributed system. Each physical node of a distributed system has a local clock. Different local clocks usually differ, even if time synchronization mechanisms, such as NTP, are used. This makes it difficult (or even impossible) to ensure correct temporal order in monitoring data [Lamport, 1978]. Monitoring approaches such as Kieker [van Hoorn et al., 2009] ensure the correct order of monitored events within a *Trace*, which can be defined as a sequence of synchronous operation executions corresponding to a single external request. Note, this definition is more restrictive than definition of trace as sequence of time stamped runtime observations (e.g., Dallmeier, 2010, p.14).

Monitoring can be categorized into event driven or timer driven [Jain, 1991, p. 95]: *Event driven monitoring* is activated only upon events, such as an execution of a software operation, a change of a state, and an arrival of an external service request. *Timer driven monitoring* creates an observation at particular times, independently of the occurrence of events. For instance, the status of a variable or register may be observed every 50 milliseconds. Timer driven monitoring is suitable to monitor very frequent events. It is also denoted *sampling* since not all state changes are observed. For instance Dapper [Sigelman et al., 2010] and Stardust [Thereska et al., 2006] use sampling to reduce overhead [Sambasivan et al., 2011].

Event driven monitoring can be further categorized into the monitoring of control flow events, data flow events, and process-level events [Bemmerl and Bode, 1991; Mansouri-Samani, 1995]. Control flow monitoring may observe for example that a software operation is called, and that a return value is passed to a caller. Data flow event monitoring focuses on observing access and changes to variables and objects.

The integration of monitoring frameworks typically requires the *instrumentation* of the software application with so-called *monitoring probes* [van Hoorn, 2007]. Instrumentation involves several application-specific design decisions, such as regarding what to monitor and where to place probes in the architecture [Focke et al., 2007a]. A summary of software monitoring tools and a discussion of instrumentation issues for distributed enterprise software applications is provided by van Hoorn et al. [2009].

Monitoring data provides input data for further analysis, such as capacity management and QoS (quality of service) management [Diaconescu

et al., 2004], the identification of performance bottlenecks [Hoffman, 2005], failure detection and fault diagnosis [Chen et al., 2002], and work-load characterization [Menascé et al., 1999].

3. Fault Localization Approach

This chapter introduces an automatic fault localization approach for distributed multi-user software systems. It assumes that faults can be located based on software timing behavior measurements. As motivated in the Introduction and in Foundations 2.3, automatic fault localization could reduce repair times and improve availability.

The two primary contributions of this thesis, presented in detail in Chapters 4 and 5, are part of the fault localization approach. The fault localization serves as application example and the case study demonstrates its applicability of our two main contributions in a fault localization setting. The fault localization approach itself is not empirically evaluated, because this would require extensive controlled experiments.

The fault localization uses monitoring to learn a system's normal timing behavior. When a failure is detected, the timing behavior before the failure is compared with historical timing behavior to detect anomalies. These anomalies are analyzed in the context of the software's architecture to find the root cause of the failure. Finally, a graphical visualization shows the administrator the fault localization results in the context of the software architecture. The fault localization can be invoked after a failure was detected; the failure detection itself is not part of the approach.

The following section includes an overview of the steps of our fault localization approach and presents fault localization assumptions. The underlying software system model and monitoring model are described in Section 3.2; these are also the base for Chapters 4 and 5. The single steps of the fault localization are described in Sections 3.3–3.7.

3.1. Approach Overview and Fault Localization Assumptions

Figure 3.1 shows the conceptual parts of the complete fault localization approach. Instrumentation and Monitoring (Section 3.3) provide the required monitoring data. The next two steps create *context-sensitive*

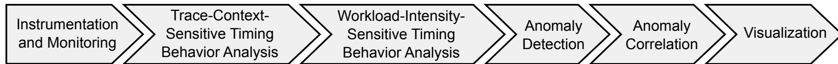
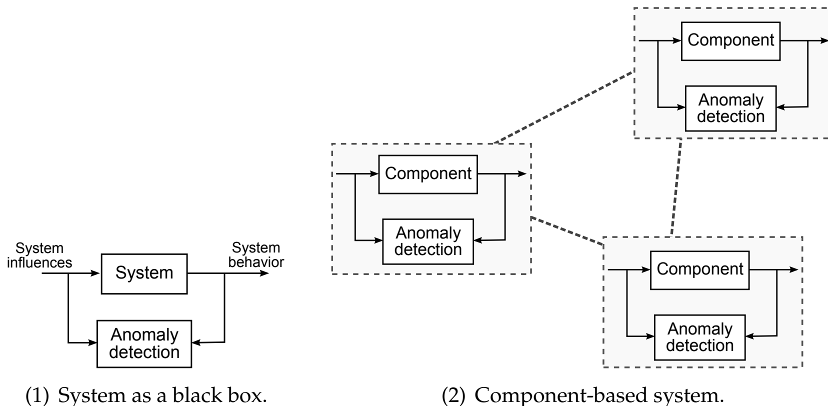


Figure 3.1.: Conceptual steps of the fault localization approach.

timing behavior models from the monitoring data (Sections 3.4 and 3.5). In Anomaly Detection (Section 3.6), the timing behavior models are used as reference models to identify deviations from normal behavior. The Anomaly Correlation (Section 3.7) evaluates all detected anomalies together in their architectural context to localize their root cause. The results are visualized in architectural diagrams for the system administrators (Section 3.7).

As illustrated in Figure 3.2, our approach localizes faults in a component-based system: First, anomaly detection computes anomaly scores for each component independently, as illustrated in Figure 3.2(1). For this, the anomaly detection algorithm compares current timing behavior with expected behavior based on historical observations. Fault localization approaches, such as Time Will Tell [Yilmaz et al., 2008], R-Capriccio [Zhang et al., 2007], and Spectroscope [Sambasivan et al., 2011] stop at this point and list the components ordered by the relative anomaly strength. Our and other approaches, such as Tiresias [Williams et al., 2007] and Pinpoint [Kiciman and Fox, 2005], additionally analyze the anomalies of all components together; the anomalies are only understood as symptoms in this case and further analysis derives the root cause from the symptoms. This step, denoted anomaly correlation, uses the structural software architecture in combination with rules that reflect knowledge on how both anomalies and errors propagate through the system. Separating anomaly correlation (or denoted event correlation) from the identification of symptoms is a common practice in general failure diagnosis systems and in communication network system (e.g., see Isermann, 2006; Gruschke, 1998b; Steinder and Sethi, 2001).

A general assumption of our fault localization approach is that active faults and errors can influence timing behavior. Support for this assumption comes from the work of Kao et al. [1993] and Mielke [2006]. As detailed in the foundations on software faults in Section 2.2, faults can cause errors, these can propagate and can create additional errors, and errors become failures by passing some system boundary, as illustrated in Figure 2.8 on Page 23. Figure 3.3 extends this propagation model to timing behavior anomalies. It can be argued that active faults and er-



(1) System as a black box.

(2) Component-based system.

Figure 3.2.: Anomaly analysis considering the system a black-box and composed out of components. (Illustrations inspired by Isermann [2006])

rors can cause changes in system behavior (otherwise there would never be failures) and these changes can also be reflected in timing behavior changes, as timing behavior "reflects everything that happens during an execution" [Yilmaz et al., 2008].

This general assumption is shared or even empirically supported by the failure diagnosis approaches in our related work that are based on timing behavior analysis (see Section 6.2.1, Page 133). For instance, Yilmaz et al. [2008] conclude from case studies that timing behavior "can be used to effectively reduce the space of potential root causes for failures". Another empirical support is provided by Williams et al. [2007]'s research, which compares several metrics in detecting changes in system behavior that observed failures. Response times performed best in their comparison against other metrics, such as CPU utilization, memory utilization, context switches, and protocol metrics.

More specific assumptions are made for parts of our approach besides our general assumption that faults and errors tend to influence timing behavior:

- The software system and its monitoring have to confirm to the model assumption described in Section 3.2. In summary, these requirements are not too strict so that it is still applicable to typical Java or C# enterprise software systems, as demonstrated in

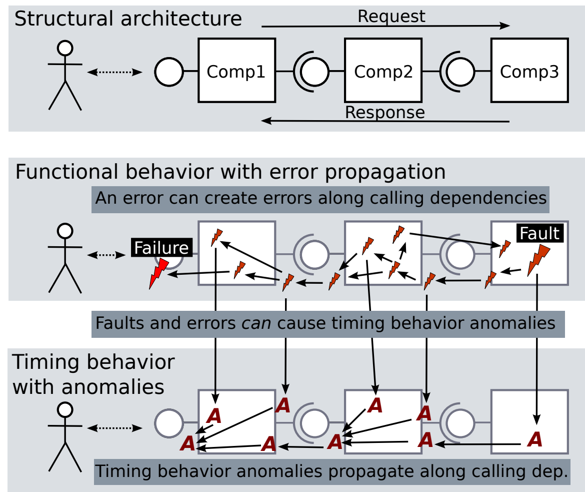


Figure 3.3.: Error propagation, errors cause anomalies, and anomaly propagation.

our case studies. However, it may not be suitable for software that is much different from enterprise software systems, such as high performance computing software, software of embedded software systems, real-time software systems, and operating system software.

- The two timing behavior analysis steps assume similar timing behavior to what is described by the corresponding foundational research presented in Foundations 2.1 on Page 9. For instance, response times of enterprise software systems tend to follow distributions such as the log-normal distribution.

- The anomaly correlation defines additional assumptions to those presented above, on how faults, errors and timing behavior propagate through the system. These are detailed in Section 3.7. The central idea behind these assumptions encoded as rules is that the propagation goes along call dependencies in the software architecture. For instance, if a particular software operation shows anomalies, then callers may also show anomalies.

3.2. Software System Model and Monitoring Model

This section presents assumptions on the software-system-under-analysis and monitoring requirements for both the fault localization approach and the two timing behavior analysis contributions of Chapters 4 and 5. The assumptions on the software system model are wide enough for application to common enterprise software systems and multi-user Web applications. The monitoring requirements presented in this section define a viewpoint on software and specify a conceptual model for the monitoring data. This model abstracts from concrete monitoring tools, so that different monitoring frameworks for the software application layer can be used, if these are able to monitor response times and request traces.

3.2.1. Software System Model

It is assumed that the software-system-under-analysis is composed out of components hosted on *execution environments*. The components provide *operations*, e.g., implemented as Web services or plain Java methods that can be requested by other components, external users, or other systems. When a request is received by the system, the execution environment assigns a free execution thread for answering the request, which eventually executes the first corresponding operation. Distributed systems, i.e., systems with more than one execution environment connected via a network, are explicitly supported by our approach.

In this thesis, the scope is limited to synchronous call actions between operation executions in the system-under-analysis. Asynchronous call actions may also occur in the system, but the corresponding caller-callee relations will be ignored. According to the UML [Object Management Group (OMG), 2007], a *synchronous call actions* temporarily stops the execution of the calling operation, denoted *caller*, and starts with executing the called operation, denoted *callee*. In other words, the caller of an operation is blocked and has to wait until the callee returns a result before it continues its own execution. *Asynchronous call actions* do not require that the caller has to wait for an answer, and hence can directly continue operation. This execution model is used by most modern programming languages such as Java, C#, and Python. However, these languages often provide additional execution concepts, besides this synchronous inter-

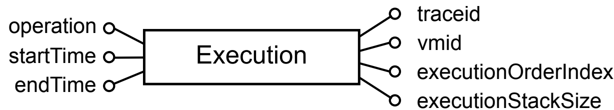


Figure 3.4.: Schema for a monitored software operation execution.

actions and asynchronous interactions. For instance, Java's exception handling mechanism can stop operation executions without letting them provide a return value.

3.2.2. Monitoring Model

Our approach requires a monitoring instrumentation that monitors executions of software operations. For simplicity, the term *execution* is used for a monitored software operation execution in the remainder of this thesis. The *monitoring data* for a time period is a multiset of such executions. We define an execution e as tuple $(o, st, nt, traceid, vmid, eoi, ess)$ for an operation o, its start time st, and end time nt. A schematic illustration of an execution is in Figure 3.4.

The $traceid$ is a unique identifier for a trace. Let a *monitored trace*, in the following just called *trace* (symbol tr), be defined as a sequence of operation executions that are connected via synchronous call actions that correspond to the same request. Traces result from user requests, and from requests of external systems or external components. As described on Page 43, the scope is limited to synchronous communication between executions: the caller of an operation is blocked and has to wait until the callee returns a result before it continues its own execution. In our model, a single user request might cause multiple such traces if asynchronous call actions are involved.

Alternative terms for concepts that are similar or identical to what we denote trace are causal path [Aguilera et al., 2003], request flow [Sambasivan et al., 2011], request path [Kiciman, 2005], path [Ball and Larus, 1996; Ammons et al., 1997], and transaction [Chanda et al., 2007].

Our approach only considers completed traces, which means that the initial operation execution has finished and that no monitoring data in between is lost. Consequently, incomplete traces and its executions are ignored. This also means that traces from infinite loops are ignored.

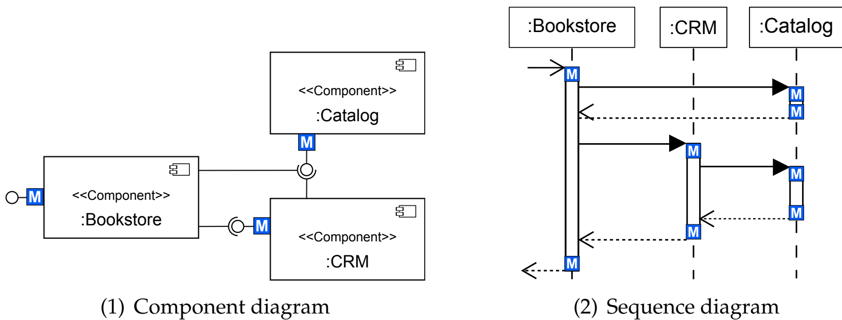
$vmid$, eoi, and ess are used for monitoring in distributed systems. The $vmid$ (virtual machine identifier) distinguishes different execution envi-

ronments, such as instances of the Java Virtual Machine, or Microsoft's Common Language Runtime. An operating system can host multiple execution environments at the same time and a physical computer can host multiple operating system instances via virtualization. The eoi (execution order index) is a counter for operation executions of a trace. The ess (execution stack size) of an operation execution is defined as the number of monitored operation executions that are on the stack during the execution's operation is called. Both eoi and ess enable the reconstruction of a trace from monitoring data. In systems with a single execution environment, these additional attributes do not need to be monitored, as clock times in a non-distributed system can reliably define the order of executions.

Conforming to the definitions of response time (1) in th Foundations on Page 10, the *response time rt* of an execution is defined as the number of time units (e.g., milliseconds) between the start time and the end time of an execution: $rt = nt - st$. The *execution time* is the fraction of the response time that is not spent in monitored subcalls. Our response time metric does not distinguish CPU time for the operation execution from other times, such as I/O processing time, resource waiting time, and response times of invoked operations (subcalls). Hence, this metric does not directly describe resource demands (e.g., CPU and I/O). The advantage of this response time metric is that it can be efficiently monitored and it does not require platform-specific monitoring functionality such as hardware performance counters.

Example In the following, an example illustrates monitoring that confirms to the description above. A bookstore software system with three components is assumed; each instrumented with a monitoring probe.

As indicated in the component diagram in Figure 3.5(1) by the blue "M" boxes, monitoring probes are integrated such that each component service request is monitored. The corresponding sequence diagram in Figure 3.5(2) shows that the monitoring logic is activated at the start and end of the activity for a service request. Table 3.1 shows example monitoring data for such a monitoring instrumentation. It lists four executions of in total three different operations within a single trace. The timestamps are provided in milliseconds reduced by some offset.



(1) Component diagram (2) Sequence diagram

Figure 3.5.: Bookstore example. “M” boxes indicate monitoring probes.

Table 3.1.: Example monitoring data for a request to the bookstore.

Operation o	$traceid$	Start time	End time	$vmid$	eoi	ess
Catalog.getBook(boolean)	1	44.86	47.00	node1	1	1
Catalog.getBook(boolean)	1	48.87	69.53	node1	3	2
CRM.getOffers()	1	48.85	69.56	node1	2	1
Bookstore.searchBook()	1	42.38	69.60	node1	0	0

3.3. Instrumentation and Trace Synthesis

In the following, it is briefly outlined how monitoring can be implemented that confirms to the monitoring model specified in the section above. Additionally, the raw monitoring data needs to be transformed into trace models by trace synthesis.

3.3.1. Instrumentation for Monitoring using Kieker

We use the monitoring framework Kieker for monitoring and instrumentation. Various other monitoring frameworks could provide suitable monitoring data for our approach with minor or no extension. Kieker is an open-source monitoring framework maintained by the software engineering group of the University of Kiel. Details on Kieker are published by van Hoorn et al. [2009]; Rohr et al. [2008c]; Focke et al. [2007a,b].

We limit the instrumentation of a software system to a subset of all operations. Our fault localization approach aims to provide failure diagnosis support; it estimates which parts of the software system contain a

failure's fault. At least each major component should be instrumented to be able to draw failure diagnosis on component level. A full instrumentation of all operations might cause a non-acceptable or non-reasonable monitoring overhead for continuous monitoring during regular operation [Focke et al., 2007a]. Full instrumentation also leads to the highest computational demands for analysis, and makes the usability of visualization more challenging.

We suggest instrumenting at least entry-points of major components, so that the resulting architectural model shows the major software structure. Furthermore, we suggest avoiding operations of little internal logic, such as "getter" and "setter" operations, and not to instrument data transport objects (see discussion in Section 7.2, Page 149).

So far, our approach does not support monitoring that changes the number of monitoring points during runtime. This would especially require an extension of our anomaly detection concept to avoid that changes in the monitoring infrastructure are flagged as anomalies.

3.3.2. Trace Synthesis

An essential step in preparing the monitoring data for our approach is the reconstruction of unambiguous traces from monitoring data, called *trace synthesis*, based on our previous work in Rohr et al. [2008c]. This produces *message traces* which are defined as finite sequences of *messages*. A message itself is defined as tuple $(e, e', a \in \{Call, Return\})$. The executions e and e' with $e \neq e'$ are connected related by a call or return action a. For the raw monitoring data from Table 3.1 the resulting single message trace is illustrated in Equation 3.1.

$$\begin{array}{lll}
((\$, & Bookstore.searchBook(), & Call), \\
(Bookstore.searchBook(), & Catalog.getBook(boolean)_1, & Call), \\
(Catalog.getBook(boolean)_1, & Bookstore.searchBook(), & Return), \\
(Bookstore.searchBook(), & CRM.getOffers(), & Call), \\
(CRM.getOffers(), & Catalog.getBook(boolean)_2, & Call), \\
(Catalog.getBook(boolean)_2, & CRM.getOffers(), & Return), \\
(CRM.getOffers(), & Bookstore.searchBook(), & Return), \\
(Bookstore.searchBook(), & \$, & Return))
\end{array} \tag{3.1}$$

The $ represents the origin of the call to the first execution within the trace. The operation names (e.g., $CRM.getOffers()$) represent executions

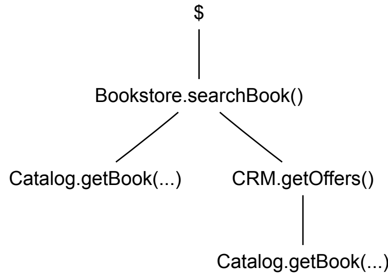


Figure 3.6.: Dynamic Call Tree for monitoring data of Table 3.1.

and $Catalog.getBook(boolean)_1$ is the first of the two executions of that operation. A message trace is both complete and well-formed if it starts with a call from \$, and ends with a return to \$. Each call $(e, e', Call)$ has a successor with an invert return $(e', e, Return)$, there are no returns without corresponding calls, and each element (e', e'', a) is directly succeeded by (e'', e''', a') except $e'' = \$$.

A *dynamic call tree* (*DCT*) is an ordered tree that represents executions of a trace as tree nodes labeled by its operation names [Ammons et al., 1997]. An edge from one node to another corresponds to a caller-callee relation within the trace. Figure 3.6 on Page 48 shows the DCT for the message trace of Equation 3.1, belonging to the data in Table 3.1 (Page 46) and to the sequence diagram of Figure 3.5(2) (Page 46).

Our trace synthesis creates message traces and dynamic call trees from monitoring data. Our approach is in particular designed for distributed systems. In practice, even proper time synchronization using NTP (network time protocol) is not precise enough to reliably identify caller/callee relationships between operation executions. Therefore, our analysis uses the two counters eoi and ess to ensure the correct order and nesting of operation executions. Our stack-machine-based implementation is part of the open Kieker sources.

3.4. Trace-Context-Sensitive Timing Behavior Analysis

After monitoring and trace synthesis, the response times are categorized according to their trace shape. For this, our approach uses the *full* trace

shape information. We denote this information the trace context. In the following, our approach Trace-Context-Sensitive timing behavior analysis (TracSTA) is summarized – a detailed description of TracSTA with an empirical evaluation can be found in Chapter 4.

TracSTA anticipates that software operation executions having the same trace shape tend to have similar timing behavior. Therefore, TracSTA splits single response time distributions into trace-shape-specific datasets to provide response time distributions with less variance. Less variance usually improves anomaly detection quality, as it better describes what is normal and what an anomaly is.

The example in Figure 3.7 illustrates this using artificial data. In this example, a catalog interacts with a database via a cache. Two types of traces exist: The first one in the upper part of the sequence diagram corresponds to a cache hit, i.e., the requested data can be found within the cache and is directly returned. The second one, shown in the lower part of the sequence diagram, corresponds to a cache miss; the data has to be fetched from the database. The response time of "Cache" will be quite different depending on whether a database access is required, as reflected in the response time distribution in the top right. TracSTA provides from the one multimodal distribution the two unimodal trace-shape-specific probability density distributions illustrated in the lower half of Figure 3.7.

3.5. Workload-Intensity-Sensitive Timing Behavior Analysis

After timing behavior is categorized into partitions based on trace contexts, WiSTA is applied, which results in a second categorization based on workload intensity.

Changes in workload intensity, such as in the number of concurrently active users, can significantly influence the timing behavior of software systems (see Foundations 2.1.1.3, on Page 12). Typically, response times tend to grow with increasing workload intensity. The workload intensity of many systems shows periodic patterns over the day and over the week, such as that there is for instance high workload intensity during working hours and low workload on weekends and on public holidays. An example is illustrated in Figure 3.8 (details in [Rohr et al., 2010]) showing the number of active users over five days of an online photo printing service that has an opposite periodic workload: High workload intensity

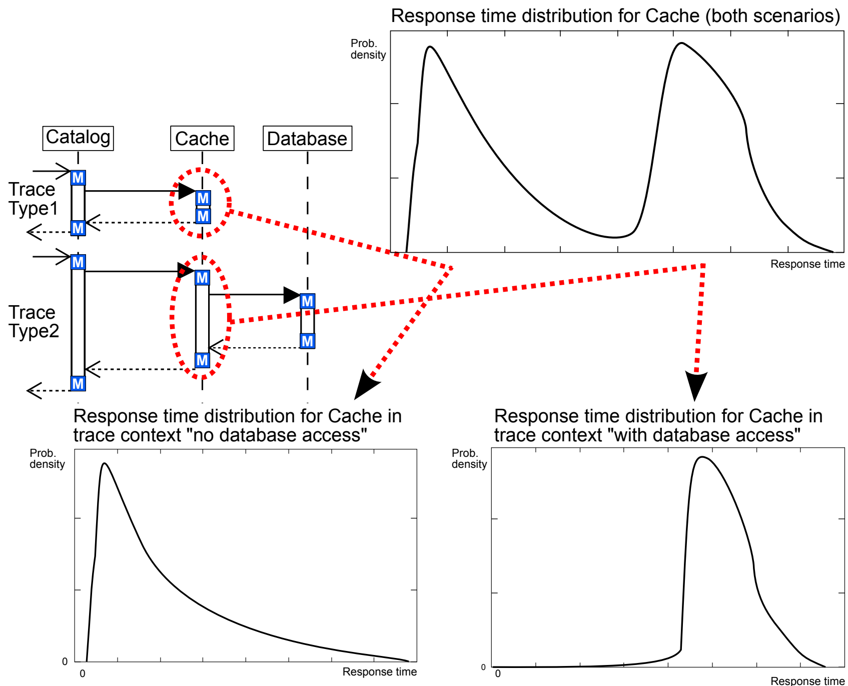


Figure 3.7.: Example: Different trace shapes can correspond to different timing behavior.

occurs in the evening with up to 200 % of the average. As indicated by workload measurements of two other photo printing services shown in Figure 3.9, a second seasonal pattern exists over the year.

As both workload intensity changes over time and it influences timing behavior, an anomaly detection approach should evaluate timing behavior in the context of the workload intensity or it will also implicitly be an anomaly detector for workload intensity changes. Ignoring workload intensity can reduce anomaly detection quality during periods of unusual high or low workload intensity.

At this point we apply WiSTA, presented in detail in Chapter 5, to distinguish software timing behavior based on workload intensity. More precisely, the response times belonging to trace contexts provided by the previous section, are further split into partitions that correspond to different levels of workload intensity. A special characteristic of WiSTA is

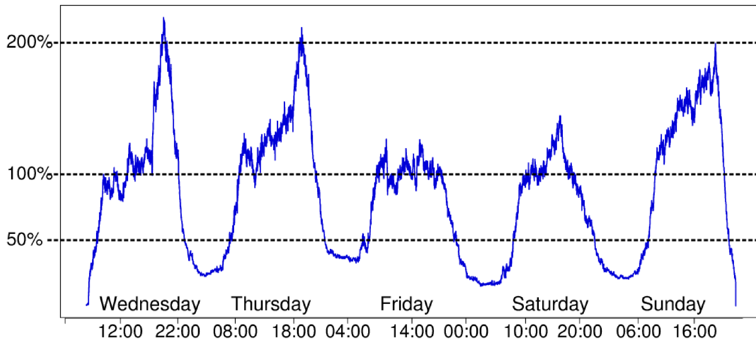


Figure 3.8.: Workload curve (active user sessions) of an online photo printing service.

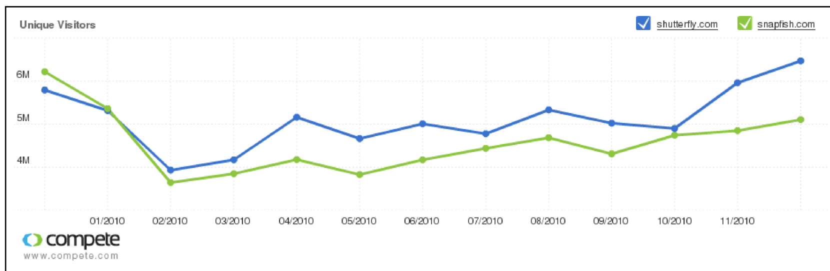
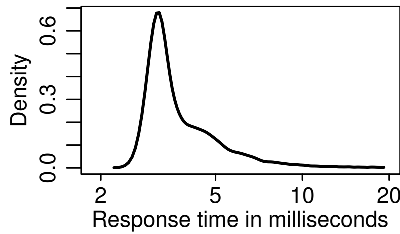


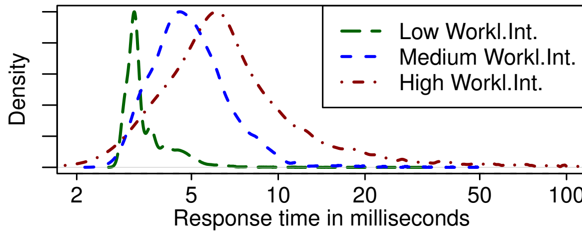
Figure 3.9.: Unique visitors of two online photo printing services. Graph by compete.com.

its workload metric that addresses also the configuration of the workload (i.e., *which* operations are executed) besides the amount of workload. The motivation behind this is that concurrent executions of some operations might compete for the same resource (e.g., CPU, network), which results in strong timing behavior interferences, while concurrent executions of other operations that do not compete for the same resources might have low timing behavior interferences.

Figure 3.10 demonstrates WiSTA with lab measurement data. Figure 3.10(1) shows the timing behavior distribution for *all* response times of one trace context. WiSTA defines three partitions of workload-intensity-specific timing behavior; the resulting distributions after splitting the data into low, medium, and high workload are displayed in Figure 3.10(2).



(1) All workload intensities.



(2) Three workload intensity levels.

Figure 3.10.: Response time distributions specific to workload intensity.

3.6. Anomaly Detection

In the previous three sections, software operation executions and their timing behavior were monitored and categorized into partitions based on trace shapes and workload intensity. This section describes how this is used for anomaly detection.

We use the straight forward anomaly detection paradigm of comparing new timing behavior measurements to historical ones. This assumes that the historical data is free of anomalies or that anomalies are at least rare (see Eskin, 2000).

Our anomaly detection is executed in three steps: preparation of training datasets, initialization of anomaly algorithms, and computation of anomaly scores.

3.6.1. Preparation of Training Datasets

As demonstrated in Table 3.2, trace shapes and workload intensity are used to define training datasets that are subsets of the full training data.

Table 3.2.: Example for training data sets.

Training dataset	Operation	Trace context	Workload intensity	Response times (ms)
1	Op1	Tc1	$[-\infty; 1.7]$	$3.3, 3.0, 2.9, 3.2, \ldots$
2	Op1	Tc1	$]1.7; 2.8]$	$9.0, 4.1, 5.0, 3.1, \ldots$
3	Op1	Tc1	$]2.8; \infty]$	$8.4, 3.5, 6.4, 26.3, \ldots$
4	Op1	Tc2	$[-\infty; 4]$	$0.3, 0.4, 0.5, 0.4, \ldots$
5	Op1	Tc2	$]4; \infty]$	$0.9, 2.3, 9, 2.0, \ldots$

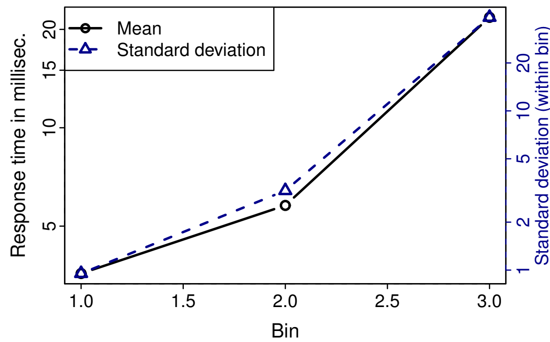
More precisely, each training dataset belongs to a single trace context and to a specific workload intensity interval. First TracSTA defines trace contexts; next, each trace context dataset is split by WiSTA into workload-intensity-specific datasets.

In the example in Table 3.2 on Page 53, there are three workload intensity intervals defined for trace context Tc1 in the first three table rows. These intervals correspond to the low-, medium-, and high workload intensity distributions of Figure 3.10(2). If the response times of a trace context show *no* correlation to workload intensity, then only the interval $[-\infty; +\infty]$ exists.

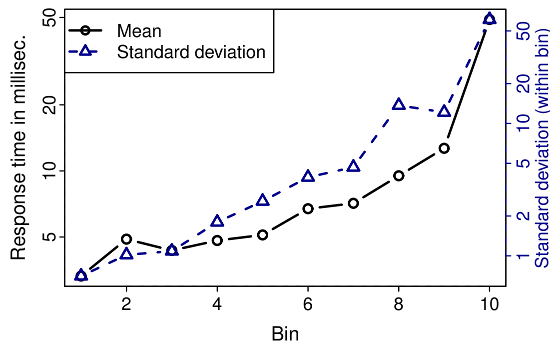
For anomaly detection each workload intensity interval should be large enough, so that the resulting training dataset contains at least several hundred response times. Our default number of different workload intensity bins is set to 5, and overlapping intervals are allowed. More bins lead to higher precision, but also to an increase in requirements for storage and computation. Figure 3.10(2) on Page 52 shows the resulting response time statistics for three bins; for the same data, Figure 3.11(2) indicates that 10 bins might even be a better choice, since the bins still show individual distribution characteristics: for each workload-intensity-based bin the mean response time and the standard deviation of the corresponding response times are different to those of other bins.

3.6.2. Initialization of the Anomaly Detectors

We use multiple anomaly detection algorithms for computing anomaly scores. Each algorithm is instantiated for each training dataset. Therefore, each instance, denoted *anomaly detector AD*, is specific to a workload intensity interval, a trace context, and an anomaly detection algorithm.



(1) 3 bins.



(2) 10 bins.

Figure 3.11.: Response time distribution characteristics for two different numbers of bins.

We define an *anomaly score* $\in [0; 1] \subset \mathbb{R}$ as the degree to what extend the observation is considered an anomaly. A score of 0 means that the execution has a quite normal timing behavior; a score of 1 means that this execution is considered to be very anomalous. In contrast to binary anomaly scores (i.e., an observation is classified an anomaly or not), non-binary scores offer more mathematical possibilities, such as expressing confidence and sorting potential causes by the strength of anomalies, as described in Foundations 2.4.3.

Each anomaly detector needs memory at runtime and CPU time for initialization. Too many anomaly detectors might impose a too large overhead or too high resource costs. To reduce the number of anomaly

detectors, trace contexts can be merged by the method described in Section 4.2.4, fewer anomaly detection algorithms can be used, fewer software operations can be instrumented, and fewer workload intensity intervals can be defined.

The anomaly detection algorithms used by our approach are not specific to timing behavior distributions. To suite the distribution characteristics of software timing behavior (see Section 2.1.3, Page 15), we apply the algorithms only on log-transformed response times. The following three anomaly detection algorithms are used:

- Normalized LOF (Local Outlier Factor) [Breunig et al., 2000]: LOF computes an anomaly score based on both distance and density. In contrast to other approaches, it can detect anomalies in data sets that have varying local density. We approximated the LOF's parameter k in several experiments with varying class size $|C|$ between 200 and 20,000 observations with $k = 1.92^{log(|C|)}$. Furthermore, the LOF values are normalized to $[0; 1]$. The software library ELKI 0.3 [Achtert et al., 2010] provides a Java implementation for both LOF and LoOP.

- LoOP (Local Outlier Probability) [Kriegel et al., 2009]: LoOP extends LOF by using probabilistic distance and it automatically provides anomaly scores in $[0; 1]$. As with LOF, we approximated in several experiments with varying class size between 200 and 20,000 observation the parameter k with $k = 1.92^{log(|C|)}$. For instance, $k = 142$ for 2,000 observations and $k = 639$ for 20,000 observations.

- Normalized inverse probability density: A probability density function produces a high output value for a common input value and a low or zero output value for an uncommon input value. With simple mathematics, its output can be normalized to $[0; 1]$ and inverted, such that 0 corresponds to very common input values and 1 for very uncommon input values. To estimate the density, we use Silverman [1986]'s kernel density estimation method in its implementation provided by GNU R[1]. Since timing behavior can be multimodal and right-skewed, we perform kernel density estimation at 5,000 equally spaced points and use kernels scaled to 25 %.

An evaluation of these different anomaly detection algorithms is out of the scope of this thesis. An exemplary comparison of the different

[1] http://www.r-project.org

anomaly detection algorithm is shown in Figure 3.12. Figure 3.12(1) shows the response times of a single software operation of a real system with real workload. The monitoring data is from a single calendar day. The scatter plot displays that two clusters of timing behavior exist. Furthermore, response times of over 5 seconds occur; this is several magnitudes larger than the average response time of about 60 milliseconds. These response times are not untypical, as multimodal log-normal distributions with right heavy-tails have been reported in the literature on software timing behavior (see Foundations 2.1.3, Page 15).

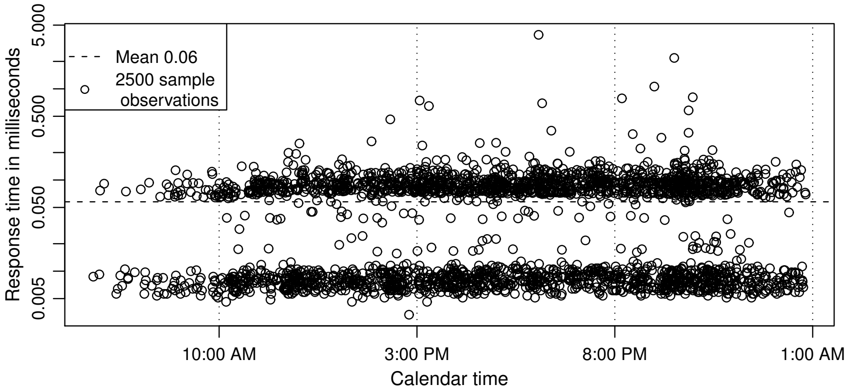
Figure 3.12(2) shows which anomaly scores would be computed for new response time observations by the three anomaly detection algorithms after trained with the data of Figure 3.12(1): for instance a response time of 0.02 ms would be scored 0.9, 0.76, and 0.98; this results in a median score of 0.9. Therefore, a response time of 0.02 ms would be considered an anomaly, which confirms to the scatter plot of Figure 3.12(1), in which response times of 0.02 ms are between the two common clusters. A response time of 0.1 ms would get low anomaly scores by all three algorithms.

3.6.3. Computation of Anomaly Scores

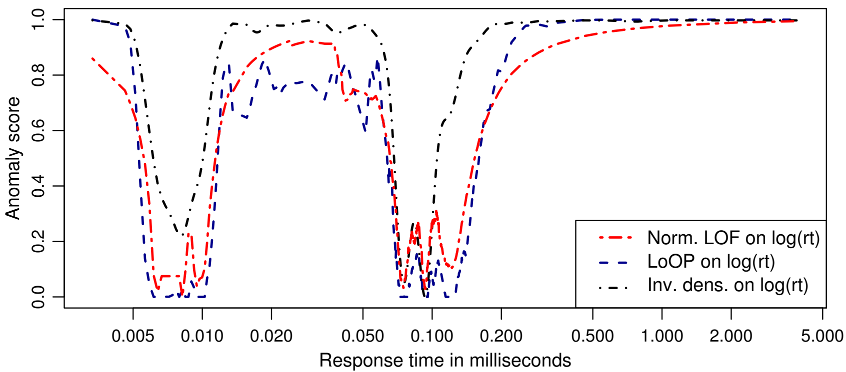
After the initialization, the anomaly detectors can be used to compute the anomaly scores. From time to time or continuously, the training datasets should be renewed and the anomaly detectors reinitialized as the normal timing behavior of a system can change over time.

The anomaly detectors can be used for failure detection and/or for fault localization. As stated in the beginning of this chapter, we address fault localization.

The anomaly detection for fault localization works as follows: After a failure has been detected, the monitoring data from the time of the failure and some time before is evaluated by the anomaly detectors. More precisely, each monitored operation execution is first preprocessed by TracSTA and WiSTA to determine the trace context and workload intensity. Next, the trace context and workload intensity are used to identify the initialized anomaly detectors that fit to the context of the execution. In our case, each execution's response time is evaluated by three anomaly detectors. Finally, the median of the three resulting anomaly scores is then annotated to its execution and the data is provided to the anomaly correlation, described in the next Section 3.7 on Page 58.



(1) Scatter plot of response times.



(2) Anomaly scores for response times using different anomaly detection functions.

Figure 3.12.: Demonstration of different anomaly detection functions.

This computation of anomaly scores for fault localization is illustrated in Table 3.3 that shows results for three evaluated executions. The first two columns are provided by monitoring. The third and fourth column are computed by TracSTA and WiSTA. The anomaly scores in the last column are the medians of three anomaly detection results.

In failure detection, the anomaly detectors and the preprocessing would regularly or immediately be applied to new monitoring data. If anomaly scores grow beyond some predefined level, an alarm would be raised. Failure detection based on timing behavior can significantly

Table 3.3.: Example for anomaly score computation.

Opera -tion	Response time (ms)	Trace context	Workload intensity	Anomaly score
Op1	3.0	Tc1	3.0	$\text{median}(0.3, 0.2, 0.0) = 0.2$
Op1	7.1	Tc1	3.0	$\text{median}(0.1, 0.1, 0.2) = 0.1$
Op1	7.1	Tc1	0.2	$\text{median}(0.8, 0.7, 0.8) = 0.8$
...	...	...	...	...

reduce the time to detect failures, as timing behavior always shows a very up-to-date view on the system. However, it is a challenge to prevent that the high level of noise in timing behavior causes many false alarms. Even in our controlled lab experiments, high single anomaly scores occurred regularly during “normal” operation. Authors such as Maxion [1990], Williams et al. [2007], Avritzer et al. [2006], Jiang et al. [2006], and Bielefeld [2012] use aggregation operations, such as mean or median filters over time windows, to reduce noise.

3.7. Anomaly Correlation and Visualization

The fault localization finishes by computing for each part of a software system an *anomaly rating* that quantifies the assumed likeliness that it contains the failure’s cause. In the previous section, anomalies where detected and represented by the anomaly *scores*; one score for each monitored software operation execution. Anomaly scores can be understood as *symptoms* of a failure. These symptoms may be at the same location as their origin, but they can also spread by propagation of timing behavior anomalies, faults, and errors, as described in Foundations 2.2.1 on Page 23 and illustrated in Figure 3.3 on Page 42. A software component might just show anomalous behavior because one of the components that it uses returns faulty results and shows anomalous behavior.

The remainder of this section summarizes our anomaly correlation concept and its visualization. The concrete algorithms together with a case study can be found in our separate publication Marwede et al. [2009]. Anomaly correlation is itself an extensive topic. Its detailed presentation and especially its evaluation highly depend on faults assumptions and system settings; therefore, it is not within the scope of this thesis.

3.7.1. Computation of Anomaly Ratings

We define an *anomaly rating* $\in [0;1]$ as the estimate that a software entity contains a failure's cause. We compute anomaly rates for several levels of software granularity - more precisely for instances of software operations, software components, and for execution environments. A high rating expresses the estimate that this entity contains a failure's fault.

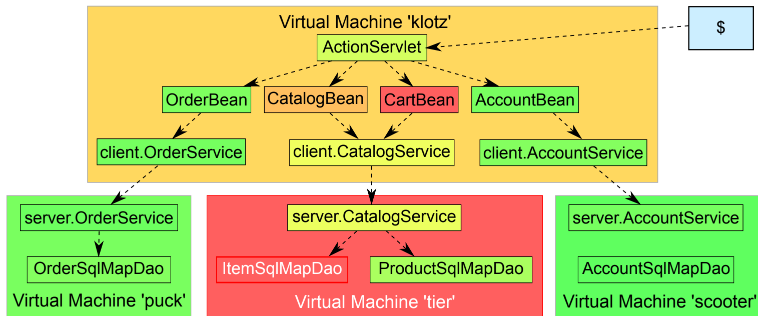
The computation of anomaly ratings is performed in several steps. The basic idea is to apply rules that compensate propagation effects of anomalies along connections of the call dependency graph. Examples for other approaches that localize faults using dependency graphs are by Gruschke [1998b], Choi et al. [1999], and Agarwal et al. [2004]. The steps to compute the anomaly ratings are:

1. Architecture model construction: Both the call dependency graph and the information on how operations, components, and execution environments map to each other are retrieved from the monitoring data. This results in a call dependency graph that uses nodes for software operations and directed edges to represent call actions between operations. Additionally, the graph is hierarchical: operations are mapped to its components and components to their execution environments.

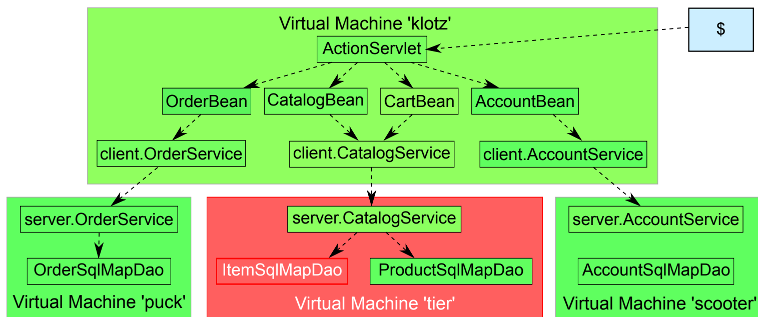
2. For each software operation, all corresponding anomaly scores are aggregated into a single anomaly rating. In our case studies, the power mean provided better results as aggregation operator than the arithmetic mean.

3. Rules are applied to the software operation's ratings that are based on knowledge on how the timing behavior anomalies typically propagate within the architectural model. Examples for these rules are:
   - If callers of an operation have in average a high anomaly rating, then the software operation's rating is increased.
   - If any callee of a software operation has a higher anomaly rating than the software operation itself, then the software operation's anomaly rating is reduced.

4. The anomaly ratings for the components and execution environments are determined by aggregating the software operation's ratings. Again, we experienced better results with the power mean than with the arithmetic mean in aggregation.



(1) A fault in ItemSqlMapDao caused high anomaly scores (symptoms) indicated by red and orange colors in depending components.



(2) Anomaly correlation removed propagation effects.

Figure 3.13.: Anomaly correlation compensates propagation effects to identify root causes instead of symptoms.

3.7.2. Visualization

The results of the automatic fault localization are provided to administrators as list of anomaly rating for the architectural elements and as graphical architecture visualization. Similar to the architectural model in correlation, the visualization is hierarchical with execution environments, components, and software operations. The hierarchy levels can be used to provide visualizations with different levels of granularity. In our visualization case studies, we did not distinguish between components and classes; this might be additionally required for larger systems to provide a good overview.

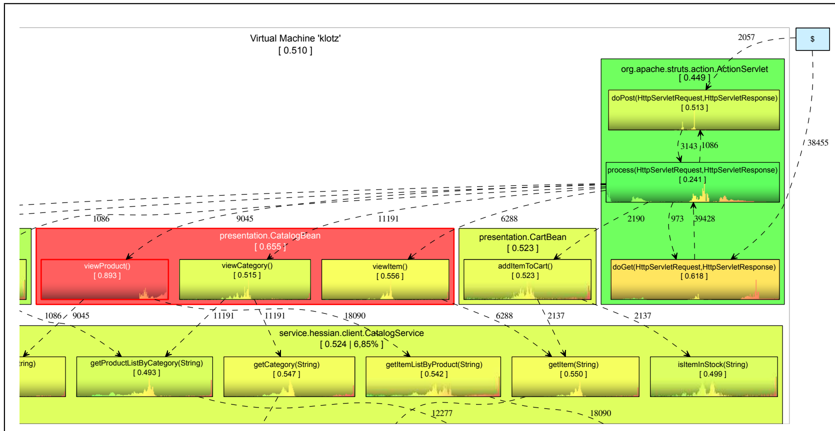


Figure 3.14.: Failure diagnosis visualization example with operation-level granularity. The failure diagnosis suspects the fault in the operation *viewProduct()* in component *presentation.CatalogBean*.

Figure 3.13 on Page 60 shows an example where only the execution environments, components, and call dependencies are visualized. Colors are used to indicate the estimated likeliness that an architectural element contains the fault. Green colors represent that an architectural element has a relatively low anomaly rating compared with the other architectural elements of the same granularity; red colors indicate high anomaly ratings. The box with the “$” represents the source of external calls. The differences between Figures 3.13(1) and 3.13(2) are discussed on Page 62.

Figure 3.13 hides details to provide a quick overview. For more details, an administrator can switch to a more detailed view as illustrated (for a different fault scenario) in Figure 3.14. Figure 3.14 shows in addition to the elements of Figure 3.13 software operations, the number of calls along an edge, the final anomaly rating, and for each software operation a histogram of the anomaly scores.

3.7.3. Anomaly Correlation Case Study

In a distributed software system, we injected faults, applied probabilistic multi-user workload, instrumented the software with monitoring probes, detected anomalies and correlated these with the algorithms described

above. The software system for this case study is the iBATIS JPetStore[2] divided into components and deployed to five servers. We generated probabilistic multi-user workload using Apache JMeter[3] extended by Markov4JMeter [van Hoorn et al., 2008]. Two classes of faults are injected based on previous work by Schwenkenberg [2007] in our research group: programming faults and database slowdowns. Each experiment run consists of 5 minutes warm-up, 15 minutes monitoring, and 5 minutes pre- and post-processing steps, such as restarting the servers. After several experiment runs, this resulted in 1.7 GB of monitoring data with 7 million executions in total and about 370,000 executions for each experiment run.

Two metrics were selected for evaluation: accuracy in localizing the fault and clearness in visualization. An accurate result is provided if the software entity in which the fault was injected has the highest anomaly rating. The clearness quantifies how focused the localization is on the part of the software that contains the fault, since faulty and non-faulty elements should be easily distinguishable based on their color by human administrators.

All five fault scenarios were accurately localized by the correlation. In one scenario, a correct localization is provided only after anomaly correlation. In the other cases, omitting correlation provides an accurate result, but the visualization does not clearly point to the faulty component.

The two failure diagnosis visualizations shown in Figure 3.13 on Page 60 are an example for differences in clearness of results from the different algorithms. It displays the results for a "database connection slowdown" scenario, with and without anomaly correlation. A fault had been injected into ItemSqlMapDao. This caused high anomaly scores in the faulty component itself, but also in components that use the faulty component. In both Figures 3.13(1) and 3.13(2) the faulty elements are highlighted correctly with red color. Figure 3.13(1) shows more yellow and orange color. Figure 3.13(2) provides a better result using correlation – it successfully compensated a timing behavior anomaly propagation to the presentation layer (Virtual Machine "klotz").

[2] http://ibatis.apache.org/
[3] `http://jakarta.apache.org/jmeter/`

4. TracSTA: Trace-Context-Sensitive Timing Behavior Analysis

This chapter introduces the first of the two primary contributions of this thesis for dealing with high variance and multimodality in timing behavior analysis.

As motivated in Chapter 1, high variance and multimodality can make it difficult to analyze software operation response times for purposes, such as anomaly detection and performance regression benchmarking. We observed that the variance and the multimodality of response times often correlate to the shape of the corresponding request traces. This variance in response times can be of several magnitudes. Therefore, trace shape is relevant information in response time analysis. An anomaly detection question, such as whether a particular response time is normal or not, should be answered "It depends on the corresponding trace shape". Similarly, one would evaluate an apartment's price by considering relevant information, such as the location (e.g., country, city, street) and the apartment's size.

Classical profiling tools (e.g., gprof [Graham et al., 1982]) and other research, for instance by Ball and Larus [1996] and Ammons et al. [1997] analyzed response times in the context of parts of the trace shape. We contribute a novel method, called *TracSTA* (trace-context-sensitive timing behavior analysis) that goes beyond current profiling practice and other research. TracSTA analyzes software operation response times in the context of their *full* trace shape. This also includes trace elements such as subcalls and operation calls that are much later within the trace than the software operation execution that is to be analyzed. Our approach helps to control more variance than earlier approaches.

The empirical evaluation shows in lab studies and industry studies that TracSTA returns response time distributions with significantly lower standard deviation compared with using less or no trace shape infor-

mation. Additionally, it is demonstrated in an industry system that multimodal timing behavior distributions can be replaced by multiple unimodal distribution using TracSTA.

Section 4.1 describes the correlation between timing behavior and trace shape information. Section 4.2 presents our approach to analyzing timing behavior in the context of its trace and the hypotheses behind the approach. The empirical evaluation follows in Section 4.3. The chapter is summarized in Section 4.4. The related work and the discussion can be found in Chapter 6 and Section 7.2.

4.1. Correlation between Timing Behavior and Trace Context

In the following, the observed correlation between timing behavior and trace shapes will be described in more detail and illustrated by an example. Furthermore, several possible underlying causes for this correlation are outlined.

Similar to other authors, listed in the Foundations (Pages 16), we observed high variance and multimodality in software operation response times of enterprise software. We identified in several systems a strong correlation of this variance to the corresponding trace shape. Additionally, several connections between multimodality in response times and trace shapes were in our monitoring data.

As defined on Page 44, a trace is a sequence of operation executions connected via synchronous call actions (i.e., calls or returns) that correspond to the same request. We define *trace shape* as the sequence of operations and the call and return actions between successive operations. Therefore, a trace shape is an abstraction of a trace – it excludes specific details of the internal operation executions of the trace (e.g., each execution's start time, end time, and response time) and excludes the trace identifier. This definition of trace shape matches to what is visualized in a dynamic call tree (see Section 3.3.2 on Page 48).

Figure 4.1 shows the correlation between response times and trace shapes. This example uses monitoring data of the iBATIS JPetStore[1], which is part of case study CS-4.1 in Section 4.3.1. Both the blue crosses and the red circles in Figure 4.1(1), which will be explained later, are response times of a single software operation (newOrder()) during two

[1]http://ibatis.apache.org/

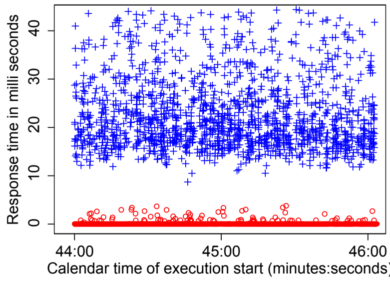
minutes of monitoring. This scatter plot and the probability density distribution in Figure 4.1(2) both show multimodal response times with clusters near 0 milliseconds and around 20 milliseconds.

A study of the traces revealed that the operation newOrder() occurs in two trace types, visualized in the sequence diagrams 4.1(3) and 4.1(4). Only the second trace type contains a subcall to insertOrder(). The source code of newOrder() includes a call to insertOrder() within nested if-statements that check for a shipping address, an order confirmation, and whether the order was already created before. This explains why a subcall to insertOrder() exists only in some traces. A distinction of the two response times based on the trace shapes of the sequence diagrams resulted in the response time distributions shown in Figures 4.1(5) and 4.1(6); the response time of newOrder() is in average about three magnitudes larger (from about 0.01 milliseconds up to 10–40 milliseconds) in presence of a subcall to insertOrder(). The source code explains this with a database interaction within insertOrder(), which is relatively time-consuming action compared with the other in-memory operations. Both distributions in 4.1(5) and 4.1(6) match to the two clusters in Figure 4.1(1): the red circles correspond to the first sequence diagram, and the blue crosses to the second one. Especially the variance of the distribution in Figure 4.1(5) is much lower than the variance of the one in Figure 4.1(2); this demonstrates that distinguishing based on trace shape can effectively isolate variance.

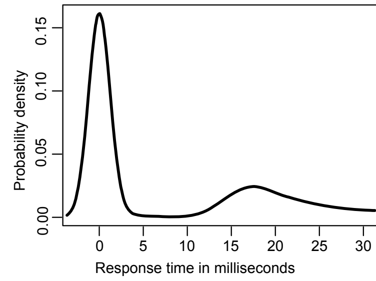
To avoid the negative effects of high variance and multimodality, it can be beneficial to first reduce the variance by considering trace shape before performing subsequent statistical analysis. This would for instance allow an anomaly detection algorithm to compare a new observation with one of the two trace-shape-specific distributions (Figures 4.1(5) and 4.1(6)) instead comparing a new observation with the combined distribution (Figure 4.1(2)) that does not distinguish both cases.

Many constellations in software architecture and implementation can cause correlations between trace shapes and response times. In the following, some causes are outlined.

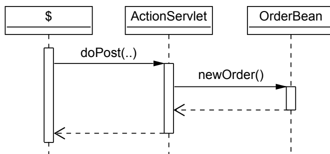
- As in the example above, software operations can contain subcalls that are only sometimes executed, if these are nested in if-statements. The monitoring traces will show which paths are taken.
- A cache can significantly reduce response times, but leads to variance. In case of a cache miss, data has to be fetched from some storage or external source. Typically, a cache hit is magnitudes



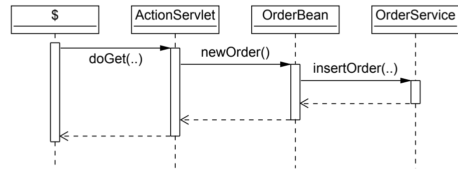
(1) Response time scatter plot newOrder().



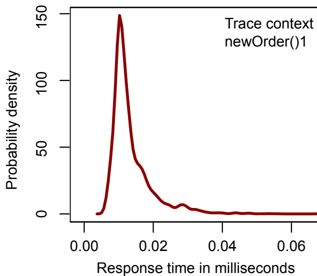
(2) Probability density distribution newOrder().



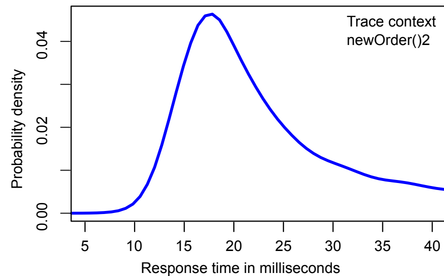
(3) Sequence diagram newOrder()1.



(4) Sequence diagram newOrder()2.



(5) Probability density distribution newOrder()1.



(6) Probability density distribution newOrder()2.

Figure 4.1.: Trace-shape-specific timing behavior: operation newOrder().

faster than a cache miss. A hit and a miss could be distinguished if the implementation uses a subcall during a cache miss.

- Good programming practice suggests to avoid code clones and to reuse software code. For this reason, a particular software operation may be called from multiple other software operations and be

reused in different use cases. The parameter types, parameter values, and parameter size can strongly vary between use cases. For instance, an online store for music, e-books, and videos can have shared operations (e.g., downloading, CRC-checking, and digital rights management) for all media types. The response times of shared operations can strongly correlate to the media type, which itself correlates to other properties such as file size.

- A service might be implemented such that the answer to a request varies depending on the current workload and depending on the user type (e.g., normal or premium). An example for the first is described by Arlitt et al. [2001] – this system changes the personalization depending on server utilization. Similarly, search engines might only provide personalized results if a user is logged in. Active personalization could be visible in traces by the occurrence of particular methods (e.g., "getPersonalizedNews(...)" or "getUserRegion(...)"). Personalized results can computationally be more expensive, which typically increases response times.

Some of the scenarios above may also correlate to other runtime characteristics, such as parameter values, system state, and parameter size besides correlating to trace shape. However, in these examples, trace monitoring is sufficient and no monitoring probes for those other characteristics are required.

4.2. Trace-Context-Sensitive Timing Behavior Analysis

This section explains our approach TracSTA for reducing variance in monitored operation response times by using trace shape information in detail. Three types of trace shape information are defined in this Section: caller context, stack context, and trace context. The concepts of caller contexts and stack contexts were used in performance analysis and profiling by Graham et al. [1982] and Ammons et al. [1997] before. TracSTA contributes the concept of trace contexts, which uses the *full* trace shape, while caller- and stack contexts use less trace shape information.

TracSTA creates trace-context-specific partitions of the response times of a software operation. We assume a system model as specified in Section 3.2.1 on Page 43. In short, this assumes that the software is

Table 4.1.: Simplified monitoring data for the ongoing TracSTA example.

Entry	Operation	Trace ID	Experiment time (μs)	Resp. time (μs)
1	d()	1	3576447911	12941
2	a()	1	3576453978	6821
3	f()	1	3576460283	334
4	f()	1	3576460675	65
5	f()	2	17259672261	1551
6	d()	3	21035692614	13534
7	a()	3	21035698342	7753
8	f()	3	21035705542	365
9	f()	3	21035705966	67
10	f()	4	26439447513	304
11	f()	5	34265513905	300
12	f()	6	36016539674	363

composed out of components that contain software operations. These operations can be called by other operations, external users, or systems.

The refined hypotheses of TracSTA of the combined hypothesis (Section 1.2) are:

$\mathbf{H}_{\mathrm{T1a}}$ A significant part of the variance in software operation response times of enterprise software systems is correlated to the full trace shape (i.e., the trace contexts).

$\mathbf{H}_{\mathrm{T1b}}$ This correlation is significantly stronger than the corresponding correlation to caller contexts and stack contexts.

$\mathbf{H}_{\mathrm{T2}}$ This correlation can be used in practice to “reduce” the variance from the perspective of subsequent timing behavior analysis steps (see Figure 1.2 in Section 1.2), such as anomaly detection.

The three essential steps of TracSTA are trace monitoring, trace shape abstraction, and partitioning. An optional fourth step, described in Section 4.2.4, further optimizes the result by reducing the number of partitions. The fourth step is not part of the empirical evaluation.

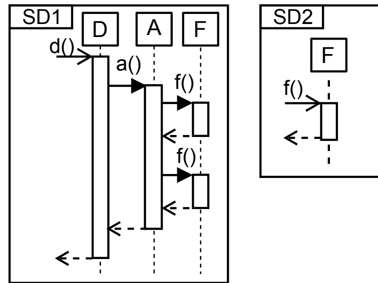


Figure 4.2.: UML sequence diagrams for the monitoring data of Table 4.1.

4.2.1. Step 1 - Trace Monitoring

The software is monitored according to Section 3.3 on Page 46. This monitoring concept defines operations, operation executions, response times, and traces.

Table 4.1 shows monitoring data from a partially instrumented telecommunication signaling system, which is also used in the second case study of this chapter (Section 4.3.2 on Page 85). The monitoring attributes vmid, eoi, ess (see Section 3.3) are omitted, because it is a single node system, which makes the timing data sufficient to correctly reconstruct the traces. Figure 4.2 visualizes the sequence diagrams corresponding to the six traces of Table 4.1. The traces 1 and 3 correspond to the sequence diagram SD1; traces 2, 4, 5, and 6 correspond to the sequence diagram SD2.

4.2.2. Step 2 - Trace Shape Abstraction

Next, for each trace, the trace shape is extracted. As defined on Page 64, a trace shape is the sequence of operations and the call and return actions between successive operations. In other words, it is only the shape of the trace – individual attributes, such as response times, execution ID, trace ID start times, and end times are omitted.

A trace shape is well-illustrated by a dynamic call tree. As described in Section 3.3.2, a dynamic call tree (DCT) [Ammons et al., 1997] is an ordered tree. Its nodes represent operation executions by its operation names and the directed tree edges correspond to the caller/callee relation within the trace. Different trace shapes imply different dynamic call trees and vice versa. In van Hoorn et al. [2009], we describe in detail how to

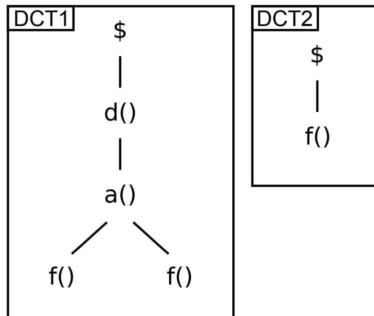


Figure 4.3.: Dynamic call trees for the sequence diagrams of Figure 4.2.

construct dynamic call trees from monitoring data and how to implement this.

Figure 4.3 shows the two DCTs for the ongoing example: DCT1 visualizes the trace shape of traces 1 and 3, which correspond to sequence diagram SD1. DCT2 shows the trace shapes of traces 2, 4, 5, and 6, which correspond to sequence diagram SD2.

4.2.3. Step 3 - Partitioning using equivalence relations

In this step, we split the response times of each operation into trace-shape-specific partitions. This partitioning is defined by one of the three following equivalence relations. TracSTA uses the third one. The two others use less trace shape information and are used for a comparison to related work and for optimizing the model size in step 4 (Section 4.2.4).

- *Caller context equivalence:* Two executions of the same operation are caller context equivalent if they have the same caller operation.
- *Stack context equivalence:* Two executions of the same operation are stack context equivalent if the paths from the corresponding nodes to their roots are equal.
- *Trace context equivalence:* Two executions of the same operation are trace context equivalent if the corresponding dynamic call trees are equal and the corresponding positions of the executions within the trees are identical.

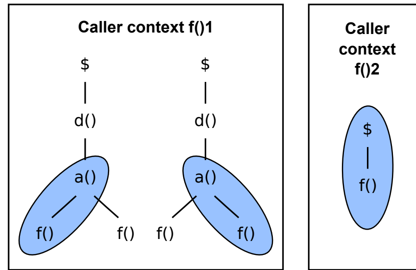
Table 4.2.: Trace shape contexts for the ongoing example.

Entry	Operation	Trace ID	Caller context	Stack context	Trace context	Response time (μs)
1	d()	1	d()1	d()1	d()1	12941
2	a()	1	a()1	a()1	a()1	6821
3	f()	1	f()1	f()1	f()1	334
4	f()	1	**f()1**	**f()1**	**f()2**	65
5	f()	2	f()2	f()2	f()3	1551
6	d()	3	d()1	d()1	d()1	13534
7	a()	3	a()1	a()1	a()1	7753
8	f()	3	f()1	f()1	f()1	365
9	f()	3	**f()1**	**f()1**	**f()2**	67
10	f()	4	f()2	f()2	f()3	304
11	f()	5	f()2	f()2	f()3	300
12	f()	6	f()2	f()2	f()3	363

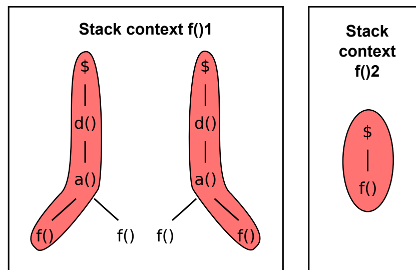
Trace context equivalence implies stack context equivalence and stack context equivalence implies caller context equivalence. Each of the three equivalence relations specifies response time partitions. In the following, we use the terms *caller-*, *stack-*, and *trace context* to refer the equivalence class on executions or their response times.

The trace shape contexts for each monitored execution of the ongoing example are listed in Table 4.2. For instance, caller context "f()1" in column "caller context" represents one of the two caller context partitions for operation $f()$. The columns for caller contexts and stack contexts are identical in this example. There is one more trace context than the other two context types: Entry 4 and entry 9 correspond both to the second subcall of operation $f()$ from operation $a()$, as shown in the sequence diagram SD1 (Figure 4.2, Page 69) and in the dynamic call tree DCT1 (Figure 4.3, Page 70). Both calls from $a()$ to $f()$ are caller context equivalent, because both are called from $a()$, and both are stack context equivalent, because the stack contains in both cases $ and $a()$. However, both calls from $a()$ to $f()$ are *not* trace context equivalent – they have the same dynamic call tree, but the $f()$ has different positions within the dynamic call tree.

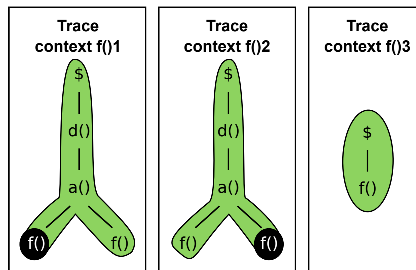
A visualization of all contexts for operation $f()$ are in Figure 4.4 on Page 72. In 4.4(1), the two colored areas in the left box are caller context



(1) The two caller contexts of $f()$.



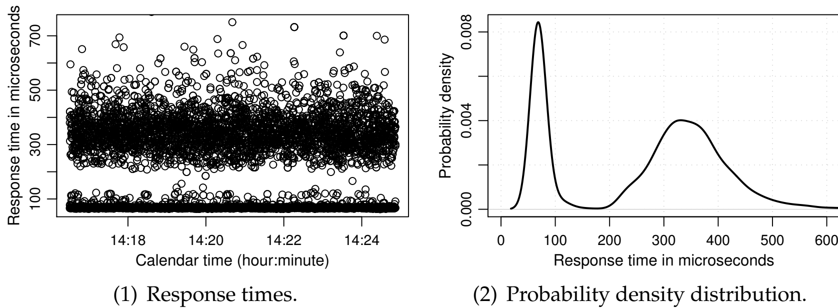
(2) The two stack contexts of $f()$.



(3) The three trace contexts of $f()$.

Figure 4.4.: All trace shape contexts for operation $f()$. The highlighted areas indicate the scope of the equivalence relations.

equivalent. This is also the case for the stack contexts in 4.4(2): both in the left box are equivalent. Only the three trace contexts of $f()$ (Figure 4.4(3)) can be distinguished, as the two left contexts have different relative position of $f()$ within the tree.



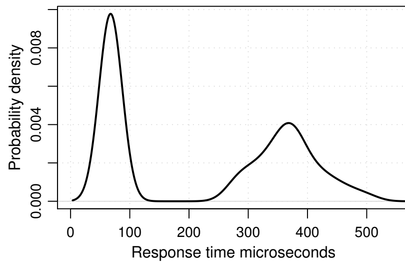
(1) Response times. (2) Probability density distribution.

Figure 4.5.: All monitored response times of operation $f()$.

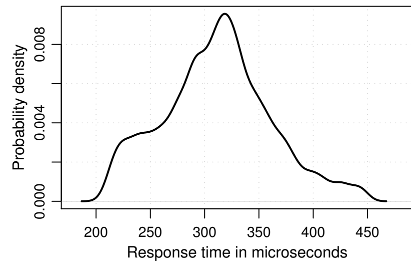
The benefit of partitioning in the context of trace shape can be demonstrated by a look at the response times of operation $f()$. A scatter plot and probability distribution of response times of operation $f()$ are shown in Figure 4.5 on Page 73. This multimodal distribution has a standard deviation of 136.47.

Figure 4.6 shows the probability density distributions for the partitions defined by stack context equivalence. As mentioned above, in this example caller context equivalence produces the same result. The stack context $f()1$ still shows a multimodal distribution (Figure 4.6(1)) like all response times of $f()$ (Figure 4.5(2)). The standard deviation for stack contexts $f()1$ and $f()2$ are 155.54 and 49.74. The average standard deviation for both stack contexts, weighted by the observed calling frequency, is 120.13. This is 11.97 % less standard deviation than without partitioning.

The distributions for the three trace contexts for operation $f()$ are shown in Figure 4.7. The multimodal distribution is replaced by three unimodal distributions. The standard deviation corresponding to trace contexts $f()1$–$f()3$ are 53.83, 2.20, and 49.74. Weighted by the calling frequency, the average standard deviation is 35.94. This means that 73.66 % of the standard deviation is connected to trace context information. In other words, most of the variance in the response time distribution of this particular operation can be removed by making trace context dependence explicit.

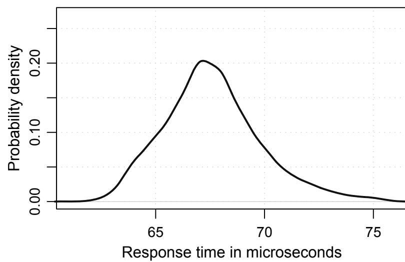


(1) Probability density distribution for stack context $f()1$.

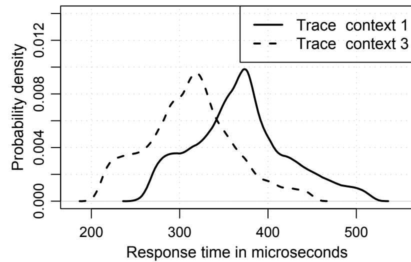


(2) Probability density distribution for stack context $f()2$.

Figure 4.6.: Stack context analysis identifies two stack contexts for operation $f()$.



(1) Probability density distribution for trace context $f()2$.



(2) Probability density distribution for trace context $f()1$ and $f()3$.

Figure 4.7.: Trace context analysis identified three contexts for operation $f()$.

4.2.4. Step 4 (optional) - Model Size Optimization

The trace context analysis can produce in some cases results with undesired characteristics. For this case, we present in the following an optional optimization step. It structures the results from caller-, stack-, and trace context analysis into a tree structure and iteratively applies optimization operators.

Possible undesired result characteristics

The trace shape analysis may produce a partitioning of software response times with undesired properties:

- Trace contexts with an insufficient number of measurements: Many statistical methods require a minimum number of observations in each partition to provide robust results.
- Too many trace contexts: The efficiency and feasibility of subsequent performance analysis steps can depend on the number of contexts and may perform badly for too many contexts. This correlates often with the previous undesired property. An example for too many trace contexts can be found in case study CS-4.3 on Page 87.
- Contexts may be distinguished that have very similar timing behavior distributions. This unnecessarily reduces the efficiency of subsequent analysis steps, such as anomaly detection.
- Trace context analysis is used in cases for which the computationally cheaper stack- or caller contexts produce equal results.
- For some software operations, there might be no benefit from using trace context analysis over the computationally cheaper caller- or stack context analysis.

Construction of the trace shape context tree

To get a response time partitioning free of the undesired properties presented above, the results of trace share context analysis are first structured into a tree, denoted *trace shape context tree* (*TSCT*) as illustrated in Figure 4.8:

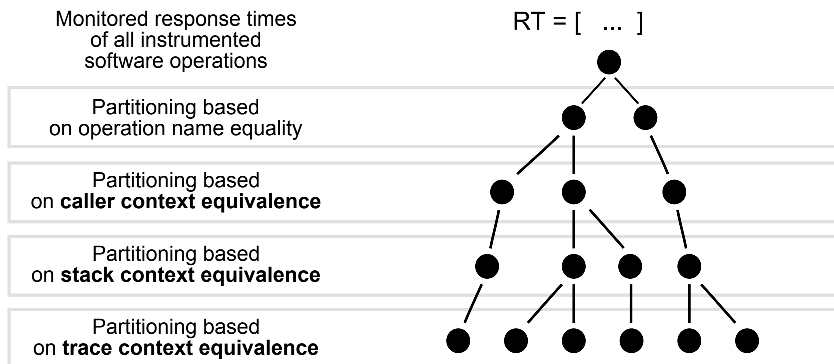


Figure 4.8.: The trace shape context tree organizes the response times by trace shape.

- The tree's root represents all monitored observations from all operations.
- Each monitored software operation and (its response times) are represented by a node on the first level of the TSCT.
- The nodes of the second level of the TSCT represent the caller contexts. Based on the callee's operation name, each second level node is connected to its corresponding first level node.
- The third tree level is defined by stack context equivalence. Each third level node is connected to its corresponding second level node.
- The fourth level of the TSCT is the partitioning defined by trace context equivalence. Each trace context node has an edge to its corresponding stack context node.

The tree's leafs together are a complete partitioning of all monitored observations. These partitions are provided to subsequent timing behavior analysis steps, such as anomaly detection.

Optimization of the trace shape context tree

We define three tree operators to remove or at least minimize the undesired characteristics. The three operators are illustrated in Figure 4.9.

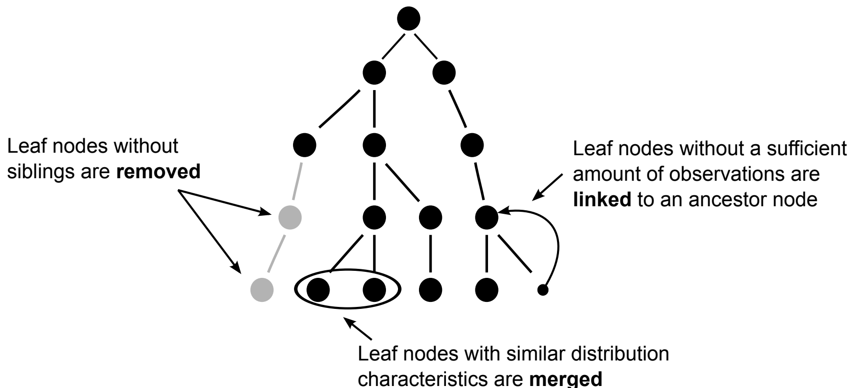


Figure 4.9.: Tree optimization operators: Node merging, removing, and linking.

1. A leaf node is **merged** to the most similar sibling if it has less than a user-specified minimum number of observations. Similarity can be defined by any user-defined similarity metric for multisets, such as the reciprocal of the distance between the median response times of two partitions.

2. Leaf nodes that have no siblings are **removed** from the tree. This reduces the size and computational costs in subsequent analysis. For instance, a trace context node that has no siblings is removed, since it makes no sense to compute and evaluate the complete trace for trace context analysis, while stack context analysis already provides the same response time partitioning.

3. Nodes in the TSCT without a sufficient number of observations and no similar siblings are **linked to** an ancestor node that has a sufficient number of observations. The linking semantics is that all corresponding response times of the linked node are used for the node that links to it.

These three operators are repeatedly applied in random order to the TSCT until no further application of the operators is possible or a user-defined stop criterion is satisfied. Possible stop criteria are for example a maximum number of contexts and a minimum number of observations per partition. The final context-sensitive timing behavior model is given by the leaf nodes of the TSCT. Case study CS-4.3 on Page 88 applies this model size optimization.

4.3. Empirical Evaluation

This section provides empirical results from several case studies. The evaluation focuses on the following research questions to address the hypotheses for TracSTA stated on Page 68:

1. How large is the standard deviation reduction in response times by using TracSTA's trace context analysis?

2. Is there a significant benefit from using the full trace shape information compared with using less trace shape information (i.e., stack context analysis and caller context analysis)?

3. Can multimodality in response time distributions be related to trace shape information?

4. How does the number of monitoring points relate to the number of resulting contexts?

5. Is TracSTA applicable in real enterprise software systems for reducing standard deviation from the perspective of subsequent timing behavior analysis steps (see Figure 1.2 in Section 1.2), such as anomaly detection?

Research questions 1, 2, and 3 address the effectiveness for dealing with high variance and multimodality in software timing behavior. Research question 3 is evaluated in the case studies by providing positive examples. Research question 4 is about efficiency – too many trace shape contexts require too much computational resources and threaten statistical robustness. Research question 5 is about the applicability of the approach.

Research questions 1, 2, and 4 use the metric *standard deviation reduction* to quantify the effect of using trace shape information for partitioning response times. The standard deviation itself is a common variance metric to characterize the dispersion of data and it quantifies the (root mean square) error in the context of prediction or estimation. For each software operation, the original standard deviation of all response times is compared with the standard deviations of all the response times of all partitions weighted by the frequency of the partitions (partition size). The resulting metric is denoted *standard deviation reduction* for a single operation; a formal definition of this metric is provided in Appendix B.

Table 4.3.: Summary of the settings of the three TracSTA case studies.

Case study	Study type	Software system	Workload	Mon. points	Research questions
CS-4.1	Lab study	iBATIS JPetStore	Simulated	2 – 198	1, 2, 3, 4
CS-4.2	Industry study	Telecommunication system	Simulated	8	1, 2, 3, 4, 5
CS-4.3	Industry study	Online photo printing service	Real	2 – 161	1, 2, 4, 5

To make a statement about all operations in a system together, all operation standard deviation reductions are weighted by the call frequency of each operation. This results in the *standard deviation reduction* over all operations. A high standard deviation reduction corresponds to a good partitioning, which usually promises better decisions (e.g., anomaly detection) than without partitioning. A random partitioning tends to result in average in a standard deviation reduction from about zero percent up to a few percent depending on the distribution; a bad partitioning can result in a negative standard deviation reduction.

Three case studies have been performed, as summarized in Table 4.3. The evaluations took place in the lab with a demo software application and in two real industry systems outside of our lab. The first industry system is a software product installed at many client sides all over the world. The second industry system is not a software product – it is developed and used only by the same company. All three software systems are modern distributed Java software enterprise applications. In the first two case studies, workload generators were used during the experiments; in the third case study, the monitoring was performed in the production system, i.e., the workload is real user workload.

The optional step 4 of TracSTA for model size optimization (Section 4.2.4) is not part of the empirical evaluation.

4.3.1. TracSTA Case Study CS-4.1

Setting

The software application analyzed in this case study is the iBATIS[2] JPetStore 5 Web application which represents an online shopping store that

[2] `http://ibatis.apache.org/`

offers pets. It is an implementation of the Sun Java Pet Store Demo application scenario [Sun Microsystems, Inc., 1994-2006]. The software is deployed in the Apache Tomcat Servlet container (version 5.5.23) running on a desktop computer equipped with an Intel Pentium 4 3 GHz hyper-threaded CPU, 1 GB physical memory, Linux 2.6.17, and Sun Java SE 1.6.0_03. Business data is stored in the database management system MySQL 5.0.18 running on Linux 2.6.15 system with two Intel Xeon 3 GHz CPUs and 2 GB physical memory. The application server and the database are connected via 100 Mbit Ethernet. A workload generator runs on a separate desktop computer equivalent to the application server.

The workload is generated by the workload driver Apache JMeter 2.2 extended by the probabilistic workload driver Markov4JMeter [van Hoorn et al., 2008]. This tool emulates users based on an application model and a mix of corresponding probabilistic user behavior models specified as Markov chains. The number of concurrent users is set to 10, which is a load that can be handled without any performance problems by the system under monitoring. A detailed workload description can be found in van Hoorn et al. [2008].

The open-source framework Kieker [van Hoorn et al., 2009] was used for monitoring. The evaluation abstracts from the problem selecting monitoring points by evaluating many random partial instrumentations of the 2^{199} possible partial instrumentations. The first 3 minutes are considered the warm-up period and are ignored in the evaluation. Three instrumentation scenarios are used:

E1 Partial instrumentation: 18 manually selected monitoring points (see van Hoorn, 2007).

E2 Full instrumentation: All operations and application entry points are monitored. This results in 199 different instrumented operations.

E3 Random instrumentation: 95,000 random instrumentations are created that have 2 to 198 monitoring points. The traces for these instrumentations are generated from the monitoring run of the full instrumentation by ignoring random subsets of monitoring points.

Table 4.4 outlines characteristics of the monitoring data of CS-4.1.

Results Reduction of Standard Deviation

Table 4.5 shows the standard deviation reduction from using the three different types of trace shape information for the first two instrumen-

Table 4.4.: CS-4.1: Summary of the instrumentation scenarios and monitoring data.

Instrumentation	Partial (E1)	Full (E2)	Random (E3)
# Instrumented operations	18	199	2–198
# Monitored executions	121,323	2,032,573	2–2,032,573
# Traces	36,190	36,036	1–36,036

Table 4.5.: CS-4.1: Standard deviation reduction results for E1 and E2.

	Standard deviation reduction	
	E1 (18 mon.pts.)	E2 (199 mon.pts.)
Caller context analysis	0.2 %	6.8 %
Stack context analysis	0.6 %	11.0 %
Trace context analysis	3.3 %	42.2 %

tation scenarios. With E1, only a small standard deviation reduction is achieved for all context types. The strong benefit from the example on Page 66 plays a minor role in these numbers, because only 1,748 of 121,323 executions are for operation $newOrder()$.

The results for E2 are quite different to those for E1: Trace context analysis results in 42.2 % standard deviation reduction in average over all operations. Additionally, trace context analysis clearly shows better results than stack context analysis and caller context analysis. A detailed analysis (see [Rohr et al., 2008a]) reveals that most operations (more than 75 %) benefit from trace context analysis in E2; most operations have at least a standard deviation reduction of 10 %. E2 has much better results than E1 because too few operations are instrumented in E1.

Figure 4.10 presents the results for E3's 95,000 random instrumentations. For certain numbers of monitoring points, many different random instrumentations have been simulated. This allows us to determine for the relation between monitoring points and the average, the first quartile, and the third quartile of all standard deviation reductions. Figure 4.10 shows that caller context analysis results in 6.8 % standard deviation reduction for full instrumentation. If half of the operations are instrumented, 75 % of the instrumentations result in a standard deviation reduction of more than 6.2 % using caller context analysis. Stack context

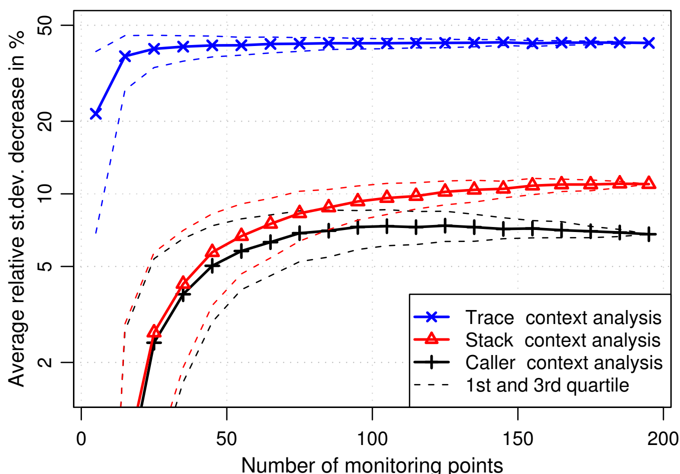


Figure 4.10.: CS-4.1: Standard deviation reduction for different numbers of monitoring points.

analysis shows better results than caller context analysis; especially for a high number of monitoring points. The standard deviation reduction rises up to 11 % for full instrumentation. Trace context analysis performs best in the comparison, independently from the number of monitoring points. This suggests that more standard deviation is connected to trace context information than to the other two context types. 40 % of average standard deviation is removed for more than the half of the evaluated instrumentations with around 40 monitoring points. For most (> 75 %) instrumentation scenarios with more than 50 monitoring points, more than 40 % standard deviation reduction was observed.

In summary, a large part of the standard deviation in this system is connected to trace context information. This is valid for the majority of all possible random instrumentations. Stack context analysis performs slightly better than caller context analysis. If only a few operations are instrumented, such as for instrumentation scenario E1, only a minor benefit may occur.

Results on the Number of Contexts

Table 4.6 presents the number of contexts for each instrumentation scenario. Similar to the results in standard deviation reduction, there are

Table 4.6.: CS-4.1: Distinct trace shape contexts per instrumentation and context type.

Instrumentation	Partial (E1)	Full (E2)	Random (E3)
# Instrumented operations	18	199	2–198
# Caller contexts	20	290	2–312
# Stack contexts	21	368	2–368
# Trace contexts	31	7,021	2–7,021

some more stack contexts than caller contexts and the number of trace contexts is much larger than the other both numbers. This data indicates that the number of trace contexts, caller contexts, and stack contexts grow by the number of monitoring points. The number of trace contexts increases faster with the number of monitoring points than both other trace shape contexts.

For full instrumentation, there are many trace contexts to define trace-context-specific partitions for most operations. As illustrated in Figure 4.11, 25 % of the operations have more than 25 trace contexts, 50 % of the operations have more than 13 trace contexts, and 75 % of the operations have more than 3 trace contexts. 39 operations (about 20 % from 80 % to 100 % in the graph) have one trace context. The average number of trace contexts per operation is 35.3 in this instrumentation scenario.

The random instrumentation scenario (E3) explores the relation between the number of monitoring points and the resulting number of contexts, as displayed in Figure 4.12. The number of stack contexts and caller contexts both tend to grow linearly with a similar rate by the number of monitoring points. In most cases (82 %), there are more stack contexts than caller contexts for an identical instrumentation, in the other cases the numbers are equal. Figure 4.12(2) visualizes the numbers of trace contexts resulting from random instrumentation scenarios. The number of trace contexts linearly increases much faster than the number of stack contexts and caller contexts.

1,344 of the 7,021 trace contexts of the full instrumentation scenario E2 had only one response time in the monitoring data. This mainly resulted from traces that contained initialization methods which were only executed once during an experiment run. Section 4.2.4 presents a method to iteratively merge similar partitions to ensure a desired partition size. However, an anomaly detection algorithm could benefit from

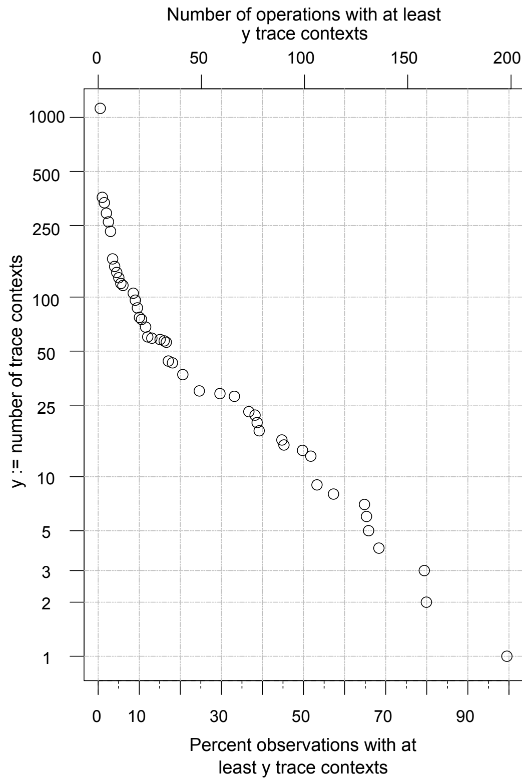
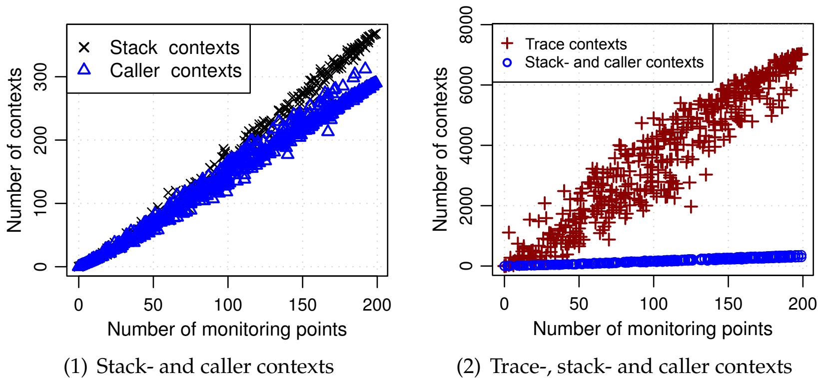


Figure 4.11.: CS-4.1: Trace contexts per operation for full instrumenation.

keeping these 1,344 response times separate if these belong to an initialization phase, since these may else trigger false alarms because of "normal unusual" timing behavior.

Results on the Relation to Multimodality

Section 4.1 on Page 64 already presented a positive example for a positiv answer to research question 3 (whether multimodality can be connected to trace context information): The operation newOrder() has multimodal timing behavior as displayed in Figure 4.1(2) on Page 66. In this case, instrumentation E1 is used (18 monitoring points). Trace context analysis defines partitions that "split" the original multimodal distribution into



(1) Stack- and caller contexts (2) Trace-, stack- and caller contexts

Figure 4.12.: CS-4.1: Number of monitoring points in relation to the number of contexts.

the two unimodal response time distributions, shown in Figures 4.1(5) and 4.1(6). Caller- and stack context analysis cannot distinguish both partitions in this scenario, as the distinction between two cases of newOrder() is within a subcall of newOrder(): only one of the two cases involves a database subcall, which results in much larger timing behavior. This shows that multimodality can be related to trace contexts and that TracSTA can "remove" in such cases the multimodality from the perspective of a subsequent timing behavior analysis.

4.3.2. TracSTA Case Study CS-4.2

Setting

Case Study CS-4.2 is in a telecommunication signaling system product of Nokia Siemens Networks. Eight monitoring points have been placed in one particular module of the large system that provides management and billing services for mobile telecommunication. Two load-balanced identical execution environments were monitored.

The workload was generated by Nokia Siemens Networks' workload driver. In total, 2.5 million software operation executions were recorded in 450 thousand traces.

Table 4.7.: CS-4.2: Standard deviation reduction and number of contexts.

	Average st.dev. reduction	# Contexts
Caller context analysis	0.2 %	10
Stack context analysis	0.2 %	10
Trace context analysis	17.8 %	88

Results

Table 4.7 presents the results for this case study. Caller contexts and stack contexts correspond to the same very small standard deviation reduction of 0.2 %. Both caller- and stack context analysis are unable to distinguish many contexts – just 10 contexts for 8 monitoring points. This explains the low results in standard deviation reduction.

Trace context analysis results in a standard deviation reduction of 17.8 % based on 88 partitions for the 8 monitoring points. Similar to the first case study, trace context analysis distinguishes more partitions than the other two types of trace shape information.

Dividing the 2.5 million measurements into 88 partitions still provides a suitable number of observations per partition for most additional statistical analysis steps. Concerning research question 4 (effectiveness), trace context analysis provides for this system and this instrumentation enough but not too many contexts. For caller- and stack context analysis, more monitoring points should be added, as this possibly increases the number of contexts per monitoring point. The computation of the caller-, stack-, and trace contexts took for all 450.000 traces about 30 seconds on a standard laptop with one CPU core.

The connection between multimodality and trace contexts (research question 3) is described in the ongoing example of Section 4.2. The multimodal distribution of operation $f()$, shown in Figure 4.5(2) on Page 73, is divided into the three partitions by trace context analysis, which have unimodal distributions, as shown in Figures 4.7(2) and 4.7(1). In contrast to trace context analysis, caller context analysis and stack context analysis were unable to resolve multimodality in this case. For this operation $f()$, trace context analysis reduced the standard deviation by 74 %. The sequence diagrams in Figure 4.2 on Page 69 explain the three different trace contexts for $f()$; the multimodality arises from the two subsequent calls of $f()$ in the first sequence diagram. The second

Table 4.8.: CS-4.3: Standard deviation reduction and number of contexts.

	Average st.dev. reduction	# Contexts
Caller context analysis	1.11 %	207
Stack context analysis	1.12 %	208
Trace context analysis	39.01 %	271,372

call is much faster than the two other calls of f(), most likely because of caching in the execution environment or in the CPU.

4.3.3. TracSTA Case Study CS-4.3

Setting

This case study took place in an online service of CeWe Color AG, Europe's largest digital photo service provider. Customers use the online service to order photo prints and other related photo products. In a part of this system, we instrumented many of the software operations and monitored the production system for several days under real user workload.

This case study uses the monitoring data of a single day, which consists of 1.5 million operation executions for the 161 instrumented software operations. Monitoring was performed using the monitoring framework Kieker. A single execution environment of the production environment was instrumented.

Results

Table 4.8 shows the results for this case study. Trace context analysis provides a strong standard deviation reduction of 39 % in average over all observations. Only a small benefit of standard reduction results from caller- and stack context analysis.

The strong standard deviation reduction for trace context analysis falls together with a very large number of trace contexts: trace context analysis distinguishes 271,372 trace contexts for 161 monitoring points. These are too many trace contexts compared with the total number of monitored observations (1,232,112); having only 4.5 observations per class is unsuitable for providing robust statistical results in additional analysis steps. For instance, experiments with the workload intensity

Table 4.9.: CS-4.3: Standard deviation reduction and number of contexts with model size optimization.

	Average st.dev. reduction	# Contexts
Caller context analysis	1.11 %	207
Stack context analysis	1.12 %	208
Trace context analysis	23.13 %	633

analysis introduced in Chapter 5 suggest that at least several hundred observations are required. Furthermore, a high number of classes leads to high resource consumption, such as demand for memory and CPU.

A detailed analysis of the software system and the monitored traces explained that the instrumentation of looped methods produces the large number of trace contexts: Several software operations (e.g., getPrice, getProduct) are called in a loop n-times during a trace, with n as the number of items in a user's shopping card. Two traces are not equivalent from the perspective of trace context equivalence if they differ in the number of calls within the loop. The instrumented loops in this system resulted in very long monitoring traces (e.g., 400 executions). This problem could have been avoided by not instrumenting operations that are within a loop, but this also reduces the level of detail in monitoring.

To reduce the number of trace contexts after monitoring, we introduced in Section 4.2.4 an additional optimization step. For instance, it iteratively merges small trace contexts to similar ones. Table 4.9 shows the results with applying this model size optimization by setting the minimum partition size to 600 response times. This merged 271,372 trace contexts into 633 contexts. After the model size optimization, still a strong standard deviation reduction of 23.13 % exists.

Regarding research question 5 (applicability), the additional model size optimization is required for this system to have a reasonable partitioning (in terms of number of contexts and observations per context). Alternatively, the monitoring instrumentation could be adapted, such that the software operations within those loops that vary according to shopping card size are not instrumented. The trace analysis without model size optimization took for monitoring data of a day of a single execution environment 15 seconds on a normal desktop PC. The prototypical implementation of the model size optimization took about 15 minutes.

4.4. Summary

This chapter presented our approach TracSTA to partition response time measurements in dependence to their trace shape for reducing the standard deviation. For this, we introduced trace context equivalence, which extends the related work and profiling practice of equivalence in caller context and stack context.

The empirical evaluation supports in three case studies our hypothesis $\mathbf{H}_{\text{T1a}}$ that a significant part of the variance of operation response times is correlated to trace contexts. It also supports hypothesis $\mathbf{H}_{\text{T1b}}$ (and research question 2) that this correlation is significantly stronger than the corresponding correlation to caller contexts and stack contexts. Furthermore, the evaluation showed that our trace shape analysis significantly reduced the standard deviation in all three case studies (research question 1, Page 78), which supports subsequent statistical analysis steps, such as anomaly detection. Our trace shape analysis TracSTA strongly outperformed related work (i.e., the caller- and stack context analysis) in reducing the standard deviation. Additionally, the case studies showed that there are cases in which multimodal distributions can be related to trace shape contexts and can be removed by trace context analysis. This positively answers research question 3.

The applicability of the approach in practice was demonstrated in two real enterprise software systems from the telecommunication and photo service domain. This supports hypothesis $\mathbf{H}_{\text{T2}}$ and provides a positive answer to research question 5 (applicability). One of the case studies was even executed in the real production system with live user workload. The overhead was suitably low to be not a major issue for applicability: TracSTA required about 15 seconds for computing trace contexts for a complete day of real workload monitoring data in CS-4.3. An overhead of this magnitude seems to be acceptable for many application scenarios, as it allows continuous application during regular operation. (A more detailed discussion of the overhead is in Section 7.2).

There results from CS-4.1 provides an answer for research question 4: The number of trace contexts, caller contexts, and stack contexts tend to grow by the number of monitoring points and the number of trace contexts increases faster with the number of monitoring points than both other trace shape contexts.

In some cases, the software architecture and the selection of monitoring points leads to very long traces which can result in too many trace contexts. This can cause too fine grained partitioning, too few observations

per partition, and reduce efficiency. To improve the applicability, TracSTA has a model size optimization step, for compensating those possible "over-partitioning" of trace context analysis. This optimization, demonstrated in case study CS-4.3, organizes the trace shape contexts into a tree structure (called trace shape context tree) and iteratively applies tree operators to merge similar trace contexts.

5. WiSTA: Workload-Intensity-Sensitive Timing Behavior Analysis

This chapter introduces the second of the two primary contributions of this thesis for dealing with high variance in software timing behavior. The first contribution analyzed traces shapes; this chapter analyzes workload intensity.

As mentioned before, software timing behavior is of high variance (see Foundations on Page 16), which can make it more difficult to draw statistical conclusions from measurements [Menascé and Almeida, 2001, pp. 168]. The author of this thesis expects that workload intensity, i.e., the amount of usage exposed to the system, is a major cause of variance in multi-user enterprise software systems. Software response times are typically longer during times of high workload intensity than during times of low workload intensity, as result of resource sharing (see Foundations on Page 21). However, workload intensity is often ignored in timing behavior analysis, such as by profiling tools or in anomaly detection.

This chapter introduces the *WiSTA* (Workload-Intensity-Sensitive Timing Behavior Analysis) approach for coping with varying workload intensity in software timing behavior. WiSTA quantifies the workload intensity during the execution of a software operation to create workload-intensity-specific timing behavior partitions.

The empirical studies of this Chapter quantify the influence of workload intensity on timing behavior variance in real world systems and in the lab, and show that this variance can be efficiently reduced using our approach compared with not considering workload intensity. This can be beneficial in typical timing behavior analysis applications, in terms of higher confidence, requiring fewer or shorter simulation runs [Jain, 1991], and increased statistical robustness.

This chapter is structured as follows. Section 5.1 presents an example to demonstrate that considering workload intensity can be required for

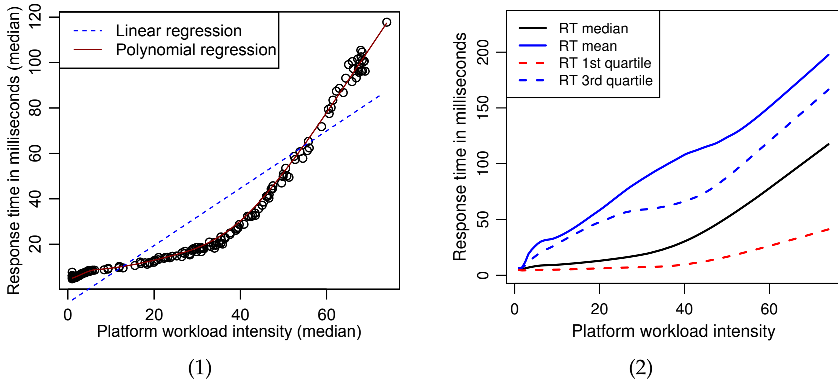


Figure 5.1.: Relation between response time statistics and workload intensity.

suitable timing behavior anomaly detection. Section 5.2 presents a new approach to workload-intensity-sensitive timing behavior analysis and presents the hypotheses of WiSTA. An empirical evaluation of the method is presented in Section 5.3. A summary of the chapter is in Section 5.4. Related work and the discussion are in the Chapters 6 and 7.2.

5.1. Correlation between Timing Behavior and Workload Intensity

In the following, the correlation between timing behavior and workload intensity in enterprise software systems is described in more detail and illustrated in an anomaly detection example.

As defined in the Foundations 2.1.1.3, the term workload intensity is used to refer to the *amount* of usage in a software system during some time period. Jain [1991] explains that response times of computer systems often increase as a function of the workload intensity. Besides this, there are other timing behavior distribution characteristics that are individual for different levels of workload intensity.

In the following, an example is provided on how workload intensity and software timing behavior are related. Figure 5.1 displays the relation between workload intensity and response times for operation *Action-Servlet.doGet()* of the iBATIS JPetStore Demo application (see the case

studies in Sections 4.3.1 and 5.3.1). The metric *platform workload intensity* (*pwi*) will be explained later in this Chapter; higher *pwi*-values represent higher workload intensity. Figure 5.1(1) shows that the response times of this operation follow a similar curve as the one characterized by [Jain, 1991, p.38], described in Foundations 2.1.4.5 on Page 21. The curve's shape indicates a nonlinear relation.

Figure 5.1(2) displays polynomial regressions for the first and third quartile, the median, and the mean response time in relation to workload intensity. 12 of the 19 operations of this case study showed similar relation curves (see Appendix A) and similar curves were observed in other systems in the context of creating this thesis. Our observations can be summarized in the following points:

- The mean response time is often above the third quartile. Such an order is typical for right-skewed heavy-tailed distributions, such as the log-normal distribution.
- The mean, median, first quartile, and third quartile of the response times tend to increase by increasing workload intensity.
- The first quartile only slowly increases by increasing workload intensity. This means that some small response times still occur during high workload intensity.
- The standard deviation tends to increase by increasing workload intensity, as indicated by the increasing distance between first and third quartile.

These points were present in most software operations of the systems that were analyzed. However, there were also software operations that showed different behavior. Therefore, their usage for purposes such as improving anomaly detection should be tested in each application scenario.

There are many possible explanations for the connection between workload intensity and varying timing behavior characteristics in software systems. For instance:

- The execution of software operations uses hardware resources, such as CPU, network, or physical storage. In multi-user systems resources are shared. Internal wait times for resource access in the lower system layers could appear as longer software response times on higher system layers.

- Similarly to the sharing of hardware resources, some software resources may cause wait times. These wait times result in longer response times. For instance, many software data types do not support real concurrent writing; an execution environment may organize simultaneous writing by defining a sequential order.
- The sharing of hardware and software resources itself needs some computational resources. During times of high workload intensity, scheduling algorithms may have to perform non-trivial coordination to provide acceptable system performance.
- Some systems adapt depending on workload intensity. At least three subcategories can be distinguished:
  - Adaptation of the amount of available (virtual) hardware resources. Cloud Computing can be used to provide computing resources on-demand [Armbrust et al., 2010].
  - Adaptation of the software architecture (e.g., Garlan et al. [2004]; Huber et al. [2012]; van Hoorn [2014])
  - Adaptation of the level of service that is provided to users. For instance, Arlitt et al. [2001] introduced a concept to reduce personalization during times of high workload intensity.

In the following, the possible consequences of ignoring workload intensity are demonstrated in an anomaly detection scenario. Figures 5.2 and 5.3 illustrate an anomaly detection example without and with increasing workload-intensity. Figure 5.2 shows simulated response times (following a normal distribution, with exponentially distributed interarrival times). Circles represent normal response times and two groups of anomalies with increased response times are represented by triangles. A basic anomaly detector called *Plain anomaly detector* (PAD) detects anomalies if response times exceed some predefined threshold value. The solid red line in Figure 5.2 represents a possible categorization by PAD that would result in an error rate of 0 %.

In Figure 5.3, the scenario is extended by linearly increasing the workload intensity over the experiment time (by decreasing the interarrival times). The response times are modeled to increase linearly by increasing workload, which is a very simplified model of the relation between workload intensity and response times. A detector like PAD cannot provide suitable results in this case: the red line represents a threshold with the minimal error rate (8 %), which has the flaw that it does not detect any

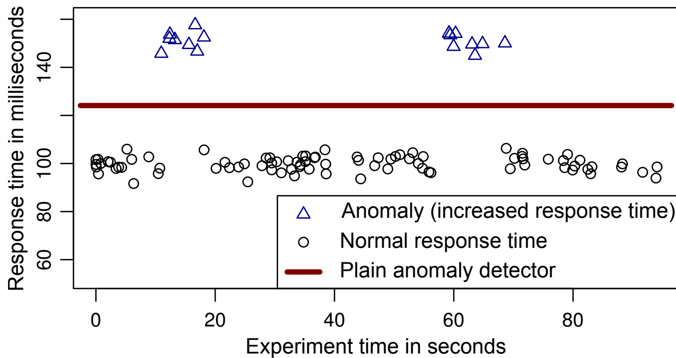


Figure 5.2.: Anomaly detection with constant workload intensity.

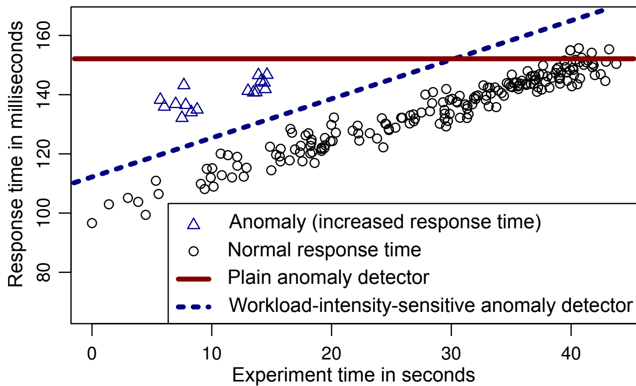


Figure 5.3.: Anomaly detection scenario with changing workload intensity.

of the anomalies. A lower threshold could correctly detect anomalies, but would consider normal values as anomalies, which would lead to a higher error rate than 8 %. A workload-intensity-sensitive anomaly detector which learns the relation between timing behavior and response times could model a threshold like the dotted blue line in Figure 5.3 with an error rate of 0 %.

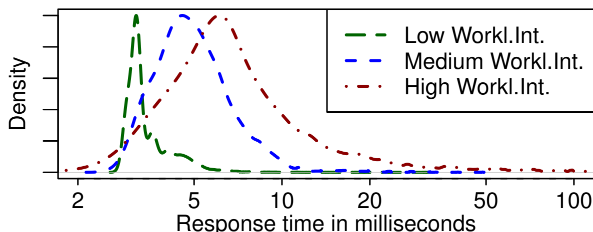


Figure 5.4.: Probability density distributions for low, medium, and high workload intensity of operation `getItemListByProduct(..)` of case study CS-1.

5.2. Workload-Intensity-Sensitive Timing Behavior Analysis

This section presents our new approach WiSTA (Workload-Intensity-Sensitive Timing Behavior Analysis), to consider workload intensity in timing behavior analysis. WiSTA's general idea is to create partitions of timing behavior that correspond to different workload intensity levels. These partitions are defined based on the values provided by our workload intensity metrics. Each partition contains the timing behavior observations for which the corresponding workload intensity is in a particular interval.

Figure 5.4 illustrates this principle for the three partitions corresponding to low, medium, and high workload intensity. The three probability density functions (PDF) for these partitions uncover workload-intensity-specific timing behavior (e.g., distribution shape, mode, median, and standard deviation).

The refined hypotheses of WiSTA of the combined hypothesis (Section 1.2) are:

H$_{W1}$ A significant part of the variance in software operation response times of multi-user enterprise software systems is correlated to workload intensity.

H$_{W2}$ This correlation can be used in practice to "reduce" the variance from the perspective of subsequent timing behavior analysis steps (see Figure 1.2 in Section 1.2), such as anomaly detection.

The key element of WiSTA is a workload intensity metric, denoted *pwi*

Table 5.1.: *pwi* metrics overview.

Metric	Time base	Execution environment	Operation weighting
pwi_1	Response times	Non-distributed	No weighting
pwi_2	Execution times	Non-distributed	No weighting
pwi_3	Execution times	Distributed	No weighting
pwi_4	Execution times	Distributed	Learned

(platform workload intensity). We introduce four alternative workload intensity metrics ($pwi_1 - pwi_4$), ordered by complexity: pwi_1 is relatively simple by being defined as the number of concurrently executing traces in a system, while pwi_4 is the average weighted sum of all concurrently executing operations over a time period within the same execution environment. All four *pwi* metrics use only basic control flow information to quantify workload intensity, such that these metrics and its monitoring can be implemented efficiently. Therefore, it can continuously be applied in real world software systems without causing too much overhead and it does not lead to requirements that typical monitoring infrastructures cannot satisfy. Furthermore, no platform-specific concepts (e.g., particular hardware performance counters) are used.

After computing the *pwi* value for each observation (using one of the four metrics), partitions of timing behavior are formed based on the *pwi* values. In Figure 5.4 three partitions are defined such that each partition has the same number of observations, which results in the *pwi* intervals in the upper-right corner of Figure 5.4. Also other binning methods are possible, such as creating bins with an equal *pwi* interval. However, each bin (i.e., class) should have a sufficient number of observations for subsequent analysis steps. In our anomaly detection context, the partition size is at least 100 observations to create robust probability density distributions for anomaly detection and usually we use 10 to 15 bins.

In the following, our four alternative *pwi*-metrics are introduced more precisely. Table 5.1 compares the metrics in terms of whether they use response- or execution times, whether they consider a measurements location in a distributed system, and whether concurrent executions of other operations are all equally weighted.

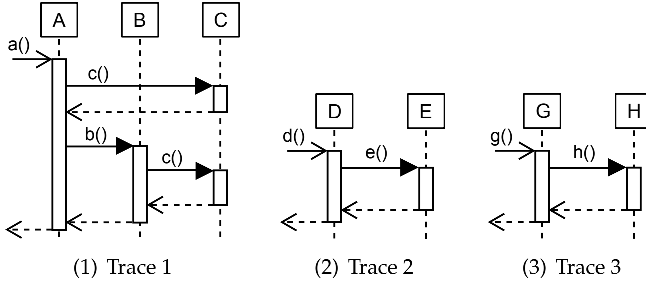


Figure 5.5.: Example traces as UML sequence diagrams.

5.2.1. pwi_1

The pwi_1 metric is defined as the average number of all concurrent traces (i.e., execution sequences) during the time period between the start (call action) and the end of an operation execution. Using the monitoring model of Section 3.2.2, for an execution $e = (o, st, rt)$ with both start time st and response time $rt \in \mathbb{N}$ (i.e., discrete time), the platform workload intensity function pwi_1 is defined in Equation 5.1.

$$pwi_1(e) := \frac{1}{rt} \sum_{t=st}^{st+rt} |AT(t)| \tag{5.1}$$

with $|AT(t)|$ as the number of elements of $AT(t)$ as

$$AT(t) := \{tr \mid \exists e' = (o', st', rt') \in tr : t \in [st', st' + rt']\}. \tag{5.2}$$

In words, for a point in time t, $AT(t)$ is the set of traces containing at least one execution that has been started and has not yet been completed. Hence, $pwi_1(e) : e \mapsto [1, \infty] \subset \mathbb{R}$ denotes the average number of traces executing during the execution time period of e. The values of pwi_1 start at 1 since an execution has always its own trace in the set AT.

Examples for the computation pwi_1 can be found in the timing diagrams in Figures 5.6 and 5.7 for operation executions of the traces of Figure 5.5. The x-axis of Figures 5.6 and 5.7 represents the elapsed time relative to the trace start time and bars are drawn for each execution aligned to the y-axis, similar to timing diagrams. All executions of the same trace are connected by directed edges and the gray shading indicates which of the executions of a trace is currently active.

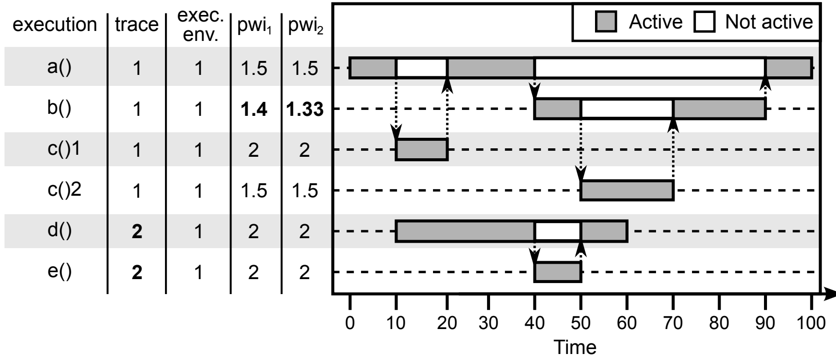


Figure 5.6.: *pwi* example 1: Multiple traces within the same execution environment.

5.2.2. pwi_2

pwi_2 differs to pwi_1 by using the execution time period instead of the response time period. Is uses the simplified assumption that an operation execution does not compete for resources while waiting for a subcall to finish. Therefore, pwi_2 ignores the time periods in which the execution to be evaluated e is waiting for the results of subcalls. This extends Equation 5.1 to

$$pwi_2(e) := \frac{1}{et} \sum_{t=st}^{st+rt} |AT(t)| \cdot et(e,t) \tag{5.3}$$

with the function $et(e,t) \rightarrow 0, 1$ with $et(e,t) = 0$ if the execution e waits at time t for a subcall to complete and 1 else.

Examples for the computation of pwi_2 can be found in Figures 5.6 and 5.7. For instance in the pwi_2 for the execution of operation $b()$ in Figure 5.6 results in $\frac{1}{30}(\sum_{40}^{50} 2) + \frac{1}{30}(\sum_{70}^{90} 1) = \frac{4}{3}$. For $b()$'s execution in Figure 5.7 pwi_2 is $\frac{1}{30}(\sum_{40}^{50} 3) + \frac{1}{30}(\sum_{70}^{80} 2) + \frac{1}{30}(\sum_{80}^{90} 1) = 2$.

5.2.3. pwi_3

In contrast to pwi_1 and pwi_2, the following *pwi* metrics consider the structure of a distributed system. pwi_3 and pwi_4 assume that mostly the active executions within the same execution environment (e.g., a

server node of a distributed system) compete for the same computational resources. For instance the CPU is only used by local operation executions on the same execution environment. However, there is also resource competition that is not covered by the assumption of pwi_3 (and pwi_4): for instance, the access to a remote database can be a competition over the network.

From the definition of a trace as sequence of operation executions (see Section 3.2), it follows that a trace can only be active in one of the execution environment at the same time, although it may contain executions of operations of more than one execution environment. Therefore, computing an execution's *pwi* should include only the activity within its corresponding execution environment, while the activity in the other execution environments must be ignored since they do not directly compete for the same resources.
Mathematically, the Equations 5.3 and 5.2 are extended for pwi_3 as follows:

$$pwi_3(e) := \frac{1}{et} \sum_{t=st}^{st+rt} |AT(t, ev)| \cdot et(e, t) \tag{5.4}$$

with

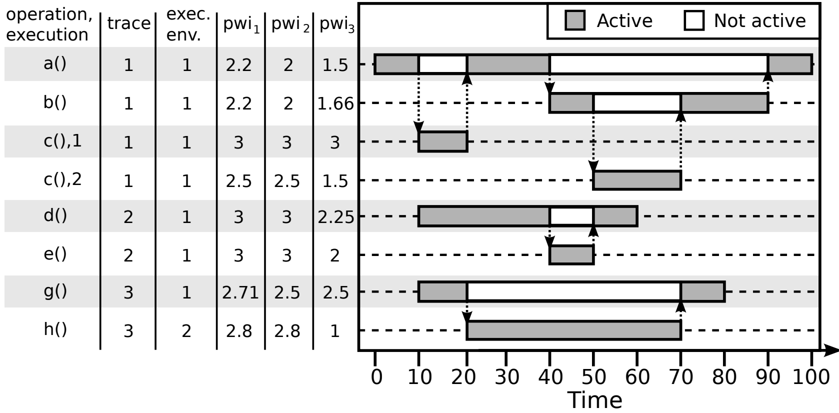
$$AT(t, ev) := \{tr \mid \exists e' \in tr : et(e', t) = 1 \\ \wedge \quad ev(e') = ev\}. \tag{5.5}$$

where $ev(e)$ provides the execution environment ev on which an execution e executes. In words, $AT(t, ev)$ is the number of traces having an execution e' that executes in the execution environment at time t without waiting for subcalls.

Examples for the computation of pwi_3 can be found in Figures 5.7, which has one execution of operation $h()$ in a separate execution environment. For $b()$'s execution, pwi_3 is $\frac{1}{30}(\sum_{40}^{50} 2)+\frac{1}{30}(\sum_{70}^{80} 2)+\frac{1}{30}(\sum_{80}^{90} 1) = 1.\overline{6}$. The execution of $h()$ has a pwi_3 value of 1 since it is in a separate execution environment, where no other executions are active, while pwi_1 and pwi_2 for $h()$ ignore mapping to execution environments.

5.2.4. pwi_4

pwi_1, pwi_2, and pwi_3 account for each operation execution equal resource demands. pwi_4 uses weights to account different resource demands for



operation, execution	trace	exec. env.	pwi₁	pwi₂	pwi₃
a()	1	1	2.2	2	1.5
b()	1	1	2.2	2	1.66
c(),1	1	1	3	3	3
c(),2	1	1	2.5	2.5	1.5
d()	2	1	3	3	2.25
e()	2	1	3	3	2
g()	3	1	2.71	2.5	2.5
h()	3	2	2.8	2.8	1

Figure 5.7.: *pwi* example 2: Trace 3 involves multiple execution environments.

each operation. This weighting addresses that some operations interfere in timing behavior more than others during concurrent execution.

Let W be defined as the weight matrix, where $w_{o,p} \in \mathbb{R}$ is the weight for considering concurrent executions of operation p during an execution of o. A relatively high value of $w_{o,p}$ indicates that executions of p have a strong influence to executions of o, for example because of sharing same resources.

The pwi_4 is computed by aggregating and weighting operation-specific pwi values as defined by Equations 5.6 and 5.7. Let e be an execution of operation o with its execution time et, then $pwi_4(e)$ is defined as:

$$pwi_4(e) := \frac{1}{et} \cdot \sum_{p=1}^{m} w_{o,p} \cdot pwi_4(e,p) \tag{5.6}$$

$$pwi_4(e,p) := \sum_{t=st}^{st+rt} |AT(t,ev,p)| \cdot et(e,t). \tag{5.7}$$

$pwi_4(e,p)$, defined by Equation 5.7, extends Equation 5.4 with a reference to the operation p. $AT(t,ev,p)$ is the set of traces having an execution of operation p that is active at time t deployed in ev:

$$\begin{aligned} AT(t,ev,p) := \{tr \mid \exists e' \in tr : & \; et(e',t) = 1 \;\wedge \\ & ev(e') = ev \;\wedge \\ & op(e') = p\}. \end{aligned} \tag{5.8}$$

Table 5.2.: Example: Two weight vectors (columns).

	W^{wait}	W^{work}
wait()	1	1
work()	0.36	18.4

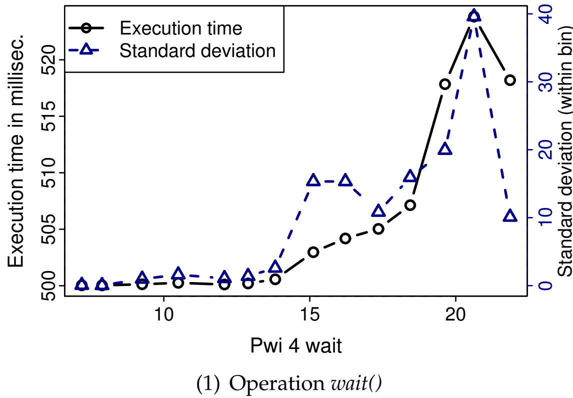
The determination of the weight matrix W is a n dimensional optimization problem, with n as number of operations in the system. The search space is smaller in a distributed system, since only the operations in the same execution environment are considered relevant. For efficient processing, W is heuristically determined from historical observations by iteratively refining the weights randomly to determine those weights that result in the highest standard deviation reduction. Our implementation prevents overfitting by early stopping: training is halted when the results for a validation dataset stop to improve.

Example *pwi*$_4$ Let a program consist of the two operations *wait()* and *work()*. Operation *wait()* waits for some time without requiring much resources during the wait period (non busy wait) and operation *work()* performs some CPU intensive computation. Example source code for this scenario can be found in Appendix C in Listing C.1. It can be expected that executing *wait()* has less impact to other concurrent executions than executing *work()*, since executing *work()* requires more resource sharing with other executions than *wait()*.

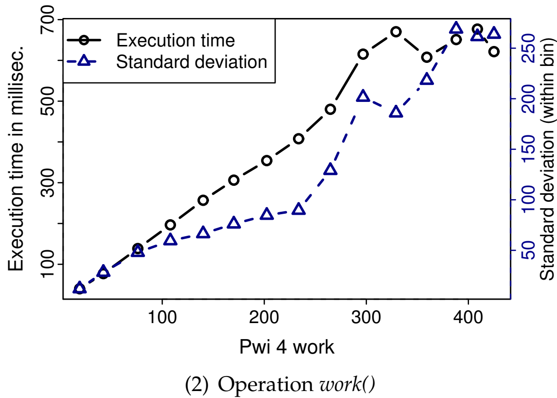
An analysis of monitoring data determined the weight vectors shown in the two columns in Table 5.2. The 18.4 in Figure 5.2 is much larger than the other weights. This shows that executions of $work()$ have the strongest timing behavior influence to other concurrent executions of $work()$; executions of $wait()$ have less influence to other executions of $work()$ or $wait()$.

The standard deviation reduction for pwi_4 is 19 % for operation $wait()$ and 73 % for operation $work()$. The results of pwi_1, pwi_2, and pwi_3 are equal at 16 % and 22 % for the two operations, as it is a non-distributed system and both operations have no subcalls. The larger benefit for $work()$ results from the higher and longer resource demand than $wait()$. In this case 15 bins were used. The benefit of each additional bin becomes relatively small for more than 10 bins in this scenario.

Figure 5.8 supports the claim that pwi_4 correlates to timing behavior



(1) Operation *wait()*



(2) Operation *work()*

Figure 5.8.: Example pwi_4: Timing behavior characteristics correlate to pwi_4.

characteristics (here mean execution time, and mean standard deviation). Therefore, it can be beneficial to consider the pwi_4 in timing behavior analysis.

5.3. Empirical Evaluation

This section provides empirical results from case studies. The evaluation focuses on the following two research questions to address the hypotheses of WiSTA that are stated on Page 96:

Table 5.3.: Summary of the settings of the WiSTA case studies.

Case study	Study type	Workload Intensity	User behavior	% CPU util.	Mon. points	System type
CS-5.0	Example	Linearly incr.	Constant	10–80	2	Two-method example
CS-5.1	Lab	Adapted from real system	Markov Model	0–80	34	iBATIS JPetStore
CS-5.2	Lab	Test scenarios	Constant scenarios	0 – 20	8	Telecommunication signaling system
CS-5.3	Field	Real	Real	0 – 15	161	Online photo printing service

1. How large is the standard deviation reduction in response times by using workload-intensity-specific context analysis?

2. Is WiSTA applicable in real multi-user enterprise software systems for reducing standard deviation from the perspective of subsequent timing behavior analysis steps (see Figure 1.2 in Section 1.2), such as anomaly detection.

Research question 1 is answered by a quantitative analysis evaluation of the effectiveness using WiSTA; research question 2 is answered by a demonstration of the applicability.

As in the previous section and in the evaluation of Chapter 4, the average relative *standard deviation reduction* is the quantitative evaluation metric for the first research question. This metric is presented in more detail on Page 78 and in Appendix B on Page 167. To achieve statistically robust evaluation results, operations with less than 600 monitored observations were excluded from the evaluation and were accounted with a standard deviation reduction benefit of 0 %.

The case studies are summarized in Table 5.3. CS-5.0 denotes the running example used in the previous section. Case studies CS-5.0 and CS-5.3 involve only one execution environment. Case studies CS-5.1, CS-5.2, and CS-5.3 use the same software systems than the evaluation of Chapter 4. The software systems are only summarized here, while more details can be found in the previous chapter.

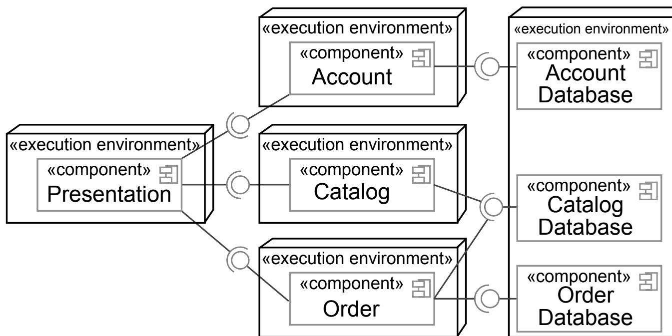


Figure 5.9.: CS-5.1: Deployment architecture of the distributed JPetStore.

5.3.1. WiSTA Case Study CS-5.1

Setting

Case study CS-5.1 uses the iBATIS JPetStore 5 Web application (see Page 79). In contrast to the setting in Section 4.3.1, a workload with varying workload intensity and a partial instrumentation of 14 software operations is used. Furthermore, the JPetStore was divided the JPetStore into four software components as illustrated in Figure 5.9. Each of the components is deployed to a dedicated machine and the components store data in a database management system (MySQL) on a fifth machine. The component communication is implemented via Web-Services.

The workload is generated by the workload driver Apache JMeter[1] extended with Markov4JMeter[2] [van Hoorn et al., 2008]. The user behavior model is identical to the one used in Section 4.3.1; it is probabilistic, specified by a Markov model. The workload intensity curve is created from measurements in a real production system, as illustrated in Figure 5.10. More precisely, it uses the shape of 24 hour monitoring data from an online customer portal of EWE TEL, a German telecommunication company. The curve is scaled to a maximum of 78 concurrent users, which corresponds to 80 % of the maximum overall system capacity, as determined in preparation experiment runs with a linearly increasing number of concurrent users. The experiment was executed 5 times; each 18-minute-run created about 740,000 response times from the

[1] http://jakarta.apache.org/jmeter/
[2] http://markov4jmeter.sourceforge.net/

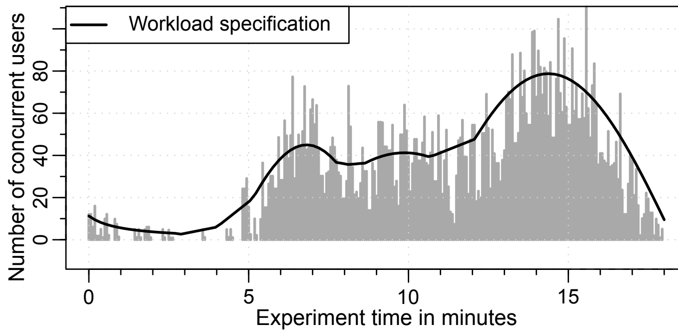


Figure 5.10.: CS-5.1: Workload intensity specification based on 24 hour measurements of a real customer portal.

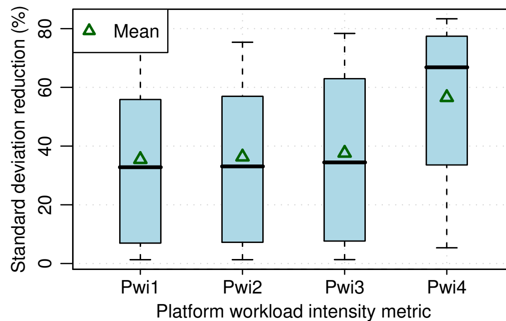


Figure 5.11.: CS-5.1: Standard deviation reduction.

34 monitoring probes in the system.

Results

Figure 5.11 shows the standard deviation reduction results in a boxplot. Each bar summarizes many standard deviation reductions achieved by using one of the four *pwi* metrics. Table 5.4 lists the exact values for average standard deviation reduction in the second row for all four metrics (average over all operations, weighted by operation execution frequency). pwi_4 performs best in the comparison, but all four methods strongly reduce standard deviation – in average from 35 % for pwi_1 up to 57 % for pwi_4.

As shown in Figure 5.12 and listed in the last row of Table 5.4, log-

Table 5.4.: CS-5.1: Average standard deviation reduction (in %).

Logarithm used	pwi_1	pwi_2	pwi_3	pwi_4
No (Non-log.)	35.45	36.31	37.61	56.56
Yes (Log.)	46.36	47.29	52.67	64.60

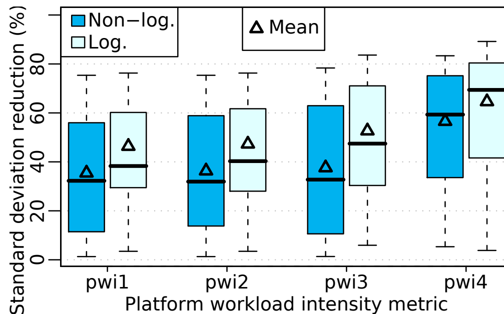


Figure 5.12.: CS-5.1: Log-transformation increases standard deviation reduction.

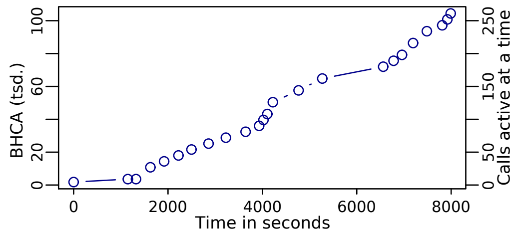
transforming the *pwi* values before defining bins additionally improves the standard deviation reduction by 29 % in average. For pwi_4, this results in an average standard deviation reduction of 65 %.

5.3.2. WiSTA Case Study CS-5.2

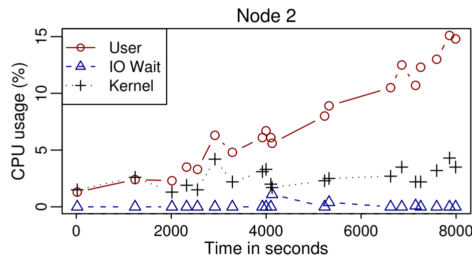
Setting

CS-5.2 uses the telecommunication signaling system of Nokia Siemens Networks, which is also used in case study CS-4.2 in Section 4.3.2 on Page 85. Eight operations were monitored and the workload was generated by special workload driver for that system. The experiment created 2.5 million monitored executions.

Figure 5.13 displays the workload intensity specification, CPU usage, and average load during system execution. The workload intensity increases from 0 to 100.000 BHCA (i.e., busy-hour call attempts, a workload intensity metric from the telecommunication systems domain), as shown in Figure 5.13(1). Figure 5.13(2) shows the CPU utilization during the case study provided by the Linux platform; it indicates that this is a scenario of low workload intensity, as the CPU utilization does not exceed 20 %.



(1) Workload intensity specification



(2) CPU usage in one execution environment

Figure 5.13.: CS-5.2: Workload intensity specification and CPU usage.

Table 5.5.: CS-5.2: Average standard deviation reduction (in %).

Logarithm used	pwi_1	$pwi_2 = pwi_3$	pwi_4
No (Non-log.)	10.34	10.15	20.45
Yes (Log.)	14.75	14.83	32.34

Results

Figure 5.14 and Table 5.5 show the standard deviation reduction results for CS-5.2. In this setting and in CS-5.3, pwi_2 and pwi_3 are equal, since the traces monitored in CS-2 never span over multiple execution environments. As in CS-5.1, pwi_4 performs best in the comparison of the four alternative methods (20 %). For all the different *pwi* metrics presented, the average standard deviation is additionally improved by more than 48 %, if the logarithm of the *pwi* values is used for defining timing behavior classes. Log-transforming the pwi_4 values before binning results in 32 % standard deviation reduction.

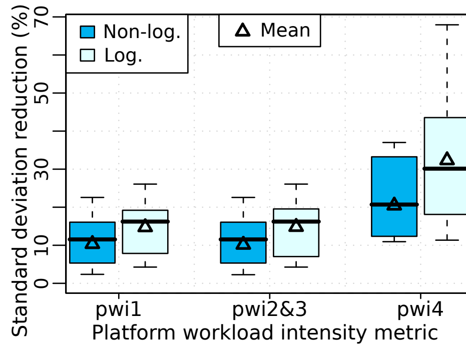


Figure 5.14.: CS-5.2: Standard deviation reduction.

5.3.3. WiSTA Case Study CS-5.3

Setting

The monitoring data in CS-5.3, is from a portal for ordering photo products, which was also used in case study 4.3 (Section 4.3.3, Page 87). The system contains image processing functionality with relatively high computational requirements.

The system was not under high load during the monitored days. The CPU utilization was most of the time between 0 % and 15 % for executing the operation executions. 161 software operations have been instrumented using the Kieker monitoring framework in a single execution environment of the production environment. The workload (shown in Figure 5.15) is the real workload intensity (active user sessions) from the production system. The 100 % line indicates the average of the main order time of Friday. The evaluated monitoring period is one day with about 1.5 million operation executions.

Results

Figure 5.16 and Table 5.6 show the standard deviation reduction results. pwi_4 performs best in the comparison of the four alternative methods (26 %). For all the different *pwi* metrics presented, the average standard deviation is additionally improved in average by 14 % if the logarithm of the *pwi* value is used for defining timing behavior classes. Again, pwi_2 equals to pwi_3, as only a single execution environment was monitored. Several of the instrumented operations had too few executions in the

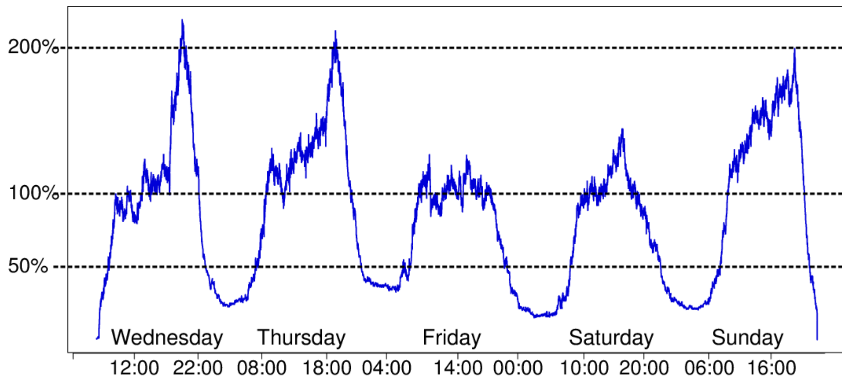


Figure 5.15.: CS-5.3: Workload curve (active sessions).

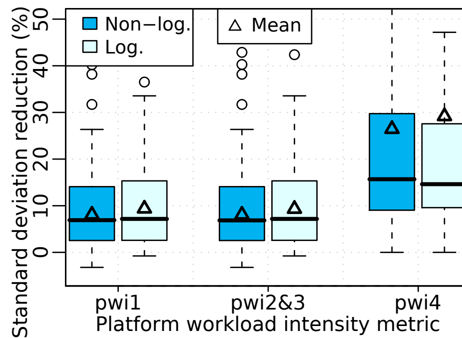


Figure 5.16.: CS-5.3: Standard deviation reduction.

Table 5.6.: CS-5.3: Average standard deviation reduction (in %).

Logarithm used	pwi_1	$pwi_2 = pwi_3$	pwi_4
No (Non-log.)	7.98	8.09	26.46
Yes (Log.)	9.27	9.35	29.15

monitoring data for statistically robust computations. For these operations, a standard deviation reduction of 0 % was accounted in computing the average relative standard deviation reduction.

5.4. Summary

The results of the evaluation support the two hypotheses of WiSTA (Page 96) and answer WiSTA's two research questions on effectiveness and applicability on Page 103: in the three case studies, WiSTA successfully used workload intensity to reduce the variance in response times.

The evaluation quantified WiSTA's effectiveness in all three case studies with strong standard deviation reduction. This result is valid for all four *pwi* metrics and for the largest part of the software operations. In some cases, more than 50 % of the standard deviation was reduced. Case studies CS-5.2 and CS-5.3 were in real systems. Case studies CS-5.2 and CS-5.3 performed less strong than CS-5.1 but still with some significant benefit of about 10 % to 30 % standard deviation reduction, which can be explained with the relatively low utilization during the monitoring period (0 % – 20 % capacity). In case of CS-5.2, a network policy prohibited to send more requests to the system under analysis. The system in CS-5.3 is sized for a seasonal business (photo product ordering) and the monitoring was performed during off-season.

pwi_4 performed best in all the case studies. In fact, pwi_4 was 60 % to 230 % better than pwi_1. These better results of pwi_4 requires a more complex implementation, additional analysis overhead (not within the monitoring system), and an additional training phase for determination of the weights of pwi_4. For the domain of timing behavior anomaly detection, the additional effort of pwi_4 may be acceptable for the expected improvements in anomaly detection quality. The results in CS-5.1 show small benefits by using the pwi_3 metric in a distributed system over pwi_1 and pwi_2. Therefore, in this system, most timing behavior inter-dependencies between concurrent software executions are between executions within the same execution environment of a distributed system. In non-distributed systems, pwi_3 is equal to pwi_2. The results from the case studies indicate only slight benefits by preferring pwi_2 over pwi_1, i.e., by using execution times and execution time periods instead of using response times and the response time period.

The three case studies of the previous section demonstrated the applicability in non-trivial multi-user enterprise software systems. This answers research question 2 (Page 103). Especially, field study CS-5.3 demonstrated the applicability in a large real world production system with real user workload. One limitation in the applicability is that WiSTA is unsuitable for software operations that are rarely executed, such as initialization methods which are executed only once. The application

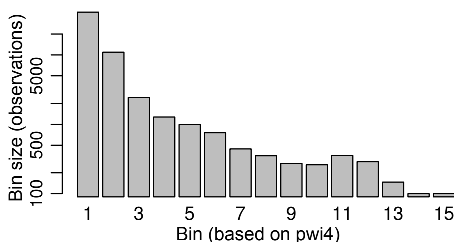


Figure 5.17.: Size of bins (i.e. classes) without log-transformation (operation $work()$).

of WiSTA in these systems was relatively easy, since only very general application-layer monitoring is required that can be provided by several monitoring frameworks – no operating system metrics or resource monitoring are required.

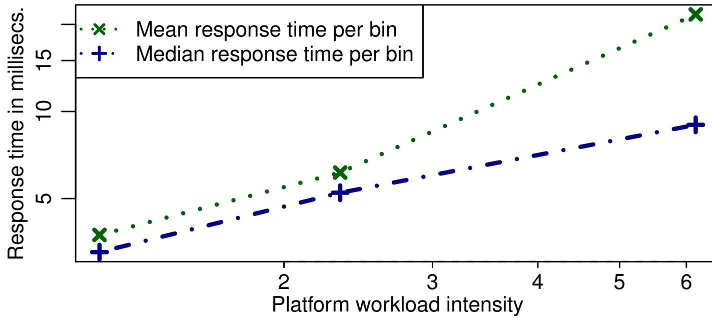
In all case studies, it was beneficial to log-transform the pwi values before binning (or alternatively to specify logarithmic bin ranges). This improved the standard deviation reduction in average over all metrics between 13 % and 40 %. One possible explanation is in the distribution of the pwi values, which tends to follow a right-skewed distribution; if bins are defined using a simple equal-size-value-range method and *no* log-transformation is performed, than most pwi values will be in the first bins, as illustrated in Figure 5.17 for the two-method example.

The vertical box size in the boxplots, for instance in Figure 5.11, is relatively large. This means that the distance between the 1st and 3rd quartile of the reduction of the operations is relatively large – in other words, some operations have a strong benefit from WiSTA while others have a small benefit. An explanation could be provided by the two-method example (Page 100): Some operations, such as $work()$ in that example, require a large amount of computing resources and have a strong competition with other operations or executions of the same operation. This strong requirement for resources and the resource competition leads to a stronger influence of varying workload intensity on timing behavior, and provides possibilities for methods such as WiSTA to reduce large amounts of standard deviation. For operations with less resource competition, such as operation $wait()$ in the two-method example, and trivial operations, such as getter- and setter methods, the execution is less dependent on the workload intensity. WiSTA requires workload intensity dependence for reducing standard deviation.

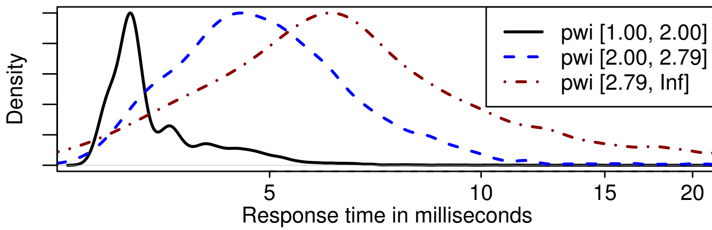
The case studies showed a strong relation between timing behavior and workload intensity, which was used for WiSTA to reduce standard deviation. However, this relation is more complex than “low workload intensity causes low response times; high workload intensity causes high response times”; illustrations such as Figure 2.6 on Page 21 by Jain [1991] or Figure 5.18(1) on Page 114 can lead to such oversimplified assumptions. This can result, for instance in high false alarm rates in anomaly detection. Important details for Figure 5.18(1) are shown in Figure 5.18(2) and Figure 5.18(3): Even at high workload intensity levels, relatively small response times can be still common. There can be several reasons for small response times at high workload intensity levels:

1. In an application under high workload intensity, the caches (e.g., CPU) might be more used for application relevant data than for other operating system tasks. This increases the chance of a cache hit.
2. Mechanisms such as load-balancers, CPU power saving, virtualization, and connection pools often adapt the amount of available resources depending to the amount of workload. For instance, power management adapts the CPU frequency. The adaptation of CPU frequency is relatively slow (e.g., seconds) compared with changes in the number of concurrent executions (milliseconds).
3. In the Java systems that were studied in the context of this thesis, the scheduling mechanisms tended to avoid interrupting small operations (typical response times of only a few microseconds), even during high workload intensity.

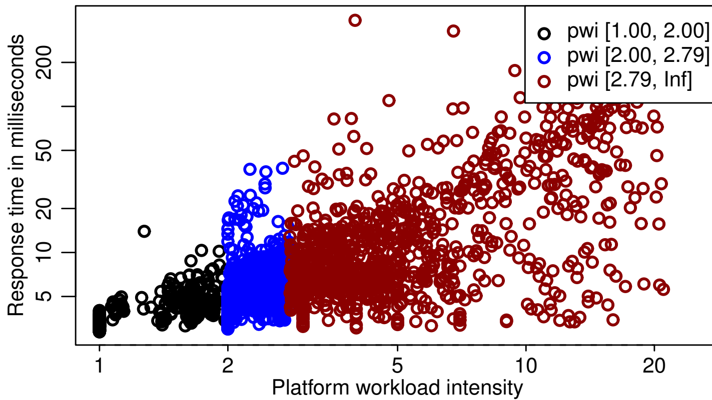
An additional discussion of selected aspects, of the applicability, and threats to validity can be found together with TracSTA in Sections 7.2 and 7.3.



(1) Workload intensity vs. mean and median response time.



(2) Probability distribution for three classes.



(3) Scatterplot of pwi_1 and corresponding response times.

Figure 5.18.: Relation between *pwi* and response times (operation *viewProduct* of CS-5.1).

6. Related Work

Related work to this thesis is in the areas of statistical analysis of software timing behavior measurements, and fault localization and failure diagnosis for software systems. The first area contains the related work for this thesis's primary contribution from Chapters 4 and 5:

- Section 6.1.1 addresses research that analyses software timing behavior measurements in context to traces, similar to our approach in Chapter 4.
- Section 6.1.2 discusses the related research that connects software timing behavior to some notion of workload intensity, as in Chapter 5.
- Section 6.1.3 compares our work (both from Chapters 4 and 5) with research that performs software timing behavior analysis with other context information than traces and workload intensity.

The related work to this thesis's secondary contribution, i.e., the fault localization approach in Chapter 3, is divided into two parts:

- Section 6.2.1 presents fault localization approaches that use timing behavior measurements, similar to our work.
- Section 6.2.2 discusses fault localization approaches that use runtime measurements other than timing behavior measurements.

6.1. Context-sensitive Timing Behavior Analysis

The analysis approaches in this section relate software timing behavior to context information. Our work uses both traces (i.e., software execution sequences) and workload intensity as context information.

6.1.1. Trace-context-sensitive Timing Behavior Analysis

As defined in Section 3.2.2 (Page 44), we consider traces in our work as sequences of software operation executions that correspond to a system or user request. This section also considers research that uses other event sequences or control flow. This related work is from research on performance profilers, automatic performance diagnosis, and more general research on monitoring and automatic analysis of runtime behavior.

Graham et al. [1982] provided foundational work on profilers that connects timing behavior (e.g., CPU time, execution times, and response times) to trace shape information. These authors introduce the *dynamic call graph* concept as model for caller-callee relationships. A dynamic call graph's nodes represent software operations. The graph's directed edges represent calls between software operations. This research resulted in the call graph profiler called gprof. An application example with gprof is in Appendix D.1. Each observed software operation is annotated with estimated execution times.

Graham et al. [1982]'s work is the first approach that performs what we denote caller-context-specific timing behavior analysis (see Page 70). In short, caller-context-specific timing behavior groups software operation timing behavior by callers. Our work extends this concept by considering the complete software operation execution sequence (i.e., our complete trace), instead of only considering callers. In contrast to gprof, our timing behavior analysis approach does not aggregate the times of the single calls, but forms groups of timing behavior for later analysis steps, such as anomaly detection.

Many profilers implement this *call graph profiling* today [Xie and Notkin, 2002], such as Intel's VTune Amplifier, Google's perftools, Valgrind, NetBeans profiler, and Java's HPROF. Some of these profilers are applied to a running example in Appendix D.

Hall [1992], Ball and Larus [1996], and Ammons et al. [1997] extended call graph profiling to (call) *path profiling*. Hall [1992] instruments LISP programs to determine the resource consumption of software operations within the context of a particular trace of connected software operation calls. Ammons et al. [1997] model a trace as direct acyclic graph, called *dynamic call tree*. Modern profilers provide path profiling to

precisely identify full traces, as for instance demonstrated by the output of the Java Virtual Machine's profiler HPROF in Appendix D.4.

Hall [1992] and Ammons et al. [1997] introduced profiling in the context of trace shapes. This associates a metric (e.g., CPU time, cache misses) with a trace shape. Besides a more accurate computation of caller specific timing behavior, Ammons et al. [1997] present stack-context-sensitive timing behavior analysis. Two software operation executions are stack context equivalent, if the execution stack contains the same sequence of operations. Executions that are stack context equivalent are always caller context equivalent.

Our work extends call path profiling and the stack-context-specific timing behavior analysis of Hall [1992] and Ammons et al. [1997]. More precisely, we use the full trace shape as context information, in contrast to using only the path to a call in the call graph.

Aguilera et al. [2003] presented call path profiling in distributed systems. It follows requests beyond the borders of single system nodes, as illustrated in Figure 6.1. The strategy to analyze timing behavior in the context of call paths, also known as *transactional profiling* [Chanda et al., 2007], was reused for instance in the approaches named Magpie [Barham et al., 2003], Whodunit [Chanda et al., 2007], Stardust [Thereska et al., 2006], and in our work.

Aguilera et al. [2003]'s approach called Project 5 combines request trace fragments into end-to-end traces based on precise clock synchronization between the different system nodes. Our and other approaches propagate tokens between trace parts to correctly reconstruct a full trace, as alternative to clock synchronization. Aguilera et al. [2003]'s approach is suitable for black-box components that cannot be internally instrumented as it only requires monitoring of inter-component communication. Therefore, it requires less intrusive monitoring than our and other approaches that look into components.

Both this and our approach analyze timing behavior in the context of traces, but this work analyzes end-to-end response times, while our work analyzes operation response times. Our work can only model request-response communication; Project 5 additionally covers message-based communication.

Barham et al. [2003] and Barham et al. [2004] proposed the Magpie approach for call path profiling in distributed systems. Figure 6.2 illus-

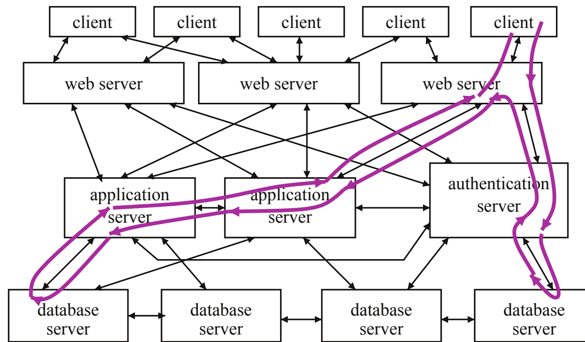


Figure 6.1.: Tracing a path through a distributed multi-tier system (Source: [Aguilera et al., 2003]).

trates an example trace visualization for a single request. Magpie collects more metrics than most call path profiling approaches. Magpie's traces include general event sequences (e.g., disk reads, network activity, and low-layer signals) from the operating system layer, with no focus on application-layer software operations. This requires operating-system-specific monitoring features.

Our approach is limited to application-layer metrics and is more focused on timing behavior analysis. Both our work and Magpie apply some kind of clustering on traces. However, Magpie's clustering is on the event order and resource consumption within a trace, while our approach addresses clusters in operation response times.

Chanda et al. [2007]'s Whodunit also extended call path profiling for application in distributed systems, similar to Aguilera et al. [2003] and Barham et al. [2003]. Particular contributions of this approach are to support a special communication type within call paths, i.e., shared memory communication, and that resource consumption measurements are mapped to program source code. This work uses the term *transactional profiling* for call path profiling in distributed systems.

Whodunit goes beyond our approach in monitoring, as it tracks more than call-return communication. In timing behavior analysis, our TracSTA goes beyond this transactional profiling: It considers the complete dynamic call tree and the position of a particular call in the tree.

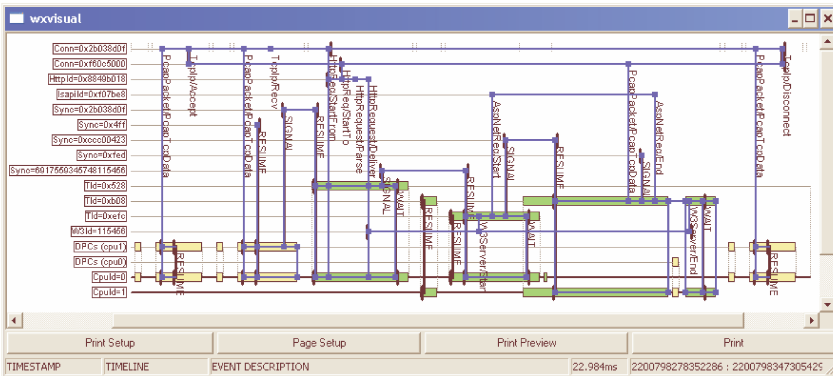


Figure 6.2.: Magpie's visualization of a single request's trace (Source: [Barham et al., 2004]).

Yilmaz et al. [2008] present a response time analysis approach and its application to fault localization. Its fault localization is discussed in the related work section on fault localization on Page 133. The timing behavior analysis compares software operation execution times with learned models of normal behavior. The approach, called Time Will Tell, creates for each software operation a Gaussian Mixture Model (GMM). This is a multi-dimensional probability density model, where the dimensions for an operation's model correspond to the execution times for the callees of the operation. In contrast to our work, this approach only considers a part of the dynamic call tree of a trace. However, this approach is the only one that shares the idea to include operation executions that start *after* the start of the execution (e.g., subcalls) to be evaluated. The approach does not explicitly consider workload intensity, in contrast to our work. The GMM might implicitly tolerate workload intensity changes, as it evaluates timing behavior in the context of times of callers and callees, which are also influenced by workload intensity changes.

Kelly [2005] detects performance anomalies in software systems, such as large Web stores. The approach relates to our work in considering both workload intensity, as well as trace shape in timing behavior analysis. In the following, the description focuses on the part that relates trace information to timing behavior analysis, while its relation to workload-intensity-sensitive timing behavior analysis is discussed on Page 124.

Kelly [2005] distinguishes timing behavior according to transaction types, which refer to end-to-end trace classes similar to use cases. Examples for transaction types are “check out”, and “add to shopping cart”. These transaction types are related to what we denote trace contexts in Chapter 4. For most software systems (e.g., in the iBATIS[1] JPetStore 5 Web application), we expect that one transaction type corresponds to multiple trace contexts. Therefore, our approach is more fine-grained. For example, a transaction “add to shopping cart” might initialize a user session within the first call of a user. This may correspond to different trace contexts for the same transaction type, if it involves additional subcalls.

Our trace context analysis requires more computational resources and more detailed monitoring than Kelly [2005]'s work. We reduce computational resource demands in our approach by merging trace contexts that show similar timing behavior. Kelly [2005]'s approach could also benefit from this merging to avoid distinguishing transaction classes that have identical timing behavior.

Kelly [2005]'s approach aggregates response times using relatively large time windows (e.g., 5 minutes) into a single number per transaction, such as the sum or the average. This can “smooth out” relevant details; additionally, aggregation operators such as the sum can be unstable if the underlying data follows a heavy-tailed distribution. The foundations in Section 2.1.3 show that this is typical for software timing behavior.

Sambasivan et al. [2011] presented Spectroscope for regression benchmarking. Regression benchmarking detects and localizes performance changes during software development (e.g., between two software versions) [Kalibera, 2006]. Spectroscope detects changes based on the occurrence frequency of trace types and based on end-to-end response times.

The trace-based change detection first categorizes traces according to their internal event sequence. Next, the relative frequency of trace types over a time period defines the reference behavior profile. For instance, it might be normal in a system with two trace types that the first has a relative frequency of 60 % and the second of 40 %. A change is detected if the distribution changes, for example to 50 %/50 %.

To detect changes in response times, the end-to-end response times of each trace type are compared with historical measurements. This

[1] `http://ibatis.apache.org/`

comparison uses the distribution-based Kolmogorov-Smirnov test.

Our work and Spectroscope both perform diagnosis using trace analysis and response times. However, there are major differences between the trace analysis of these authors and our trace analysis:

- Our traces are software operation execution sequences. In contrast, Spectroscope's traces are more general event sequences.
- Spectroscope evaluates end-to-end response times, whereas we evaluate response times of software operation.
- Spectroscope assumes a similar workload mix during the historical period and new observation period. This difference to our work results from different goals: Spectroscope is for regression benchmarking in a controlled experiment and our work analyzes timing behavior of production systems.

Our change localization (Chapter 3) also differs from Sambasivan et al. [2011]'s approach: Our work analyzes anomaly ratings (using dependency-graph-based rules) to derive causes from symptoms (i.e., the anomalies). Additionally, Spectroscope detects anomalies both in the trace type distribution and in timing behavior distributions, while our failure diagnosis only detects anomalies in timing behavior and not in traces.

Ehlers and Hasselbring [2011] and Ehlers [2012] presented call path profiling with a self-adaptive monitoring instrumentation for timing behavior analysis and anomaly detection. It contributes a rule-based, automatic adaptation of monitoring instrumentation during runtime. This reduces unrequired monitoring overhead and overhead from processing the measurement data. Most other approaches and also our work, require the manual replacement of monitoring points and an application restart for changes in monitoring. The contribution of these authors to failure diagnosis is addressed on Page 135.

These authors apply stack context analysis (Chapter 4), i.e., timing behavior measurements are correlated to the stack of all callers of a software operation, such as introduced by Ammons et al. [1997]. Our work extended the stack-context-sensitive timing behavior analysis by using the full trace as context for timing behavior analysis. This requires more computation during analysis, but provides more precision in timing behavior analysis than stack context analysis, as indicated by empirical results in Chapter 4. Both this and our work use the Kieker monitoring framework [van Hoorn et al., 2009].

Hrischuk et al. [1995] and Hrischuk et al. [1999] monitor traces (called angiotraces) in early prototypes of a software system. These traces are used to create layered queueing network models for performance prediction. This prediction enables one to evaluate design alternatives and to detect performance problems during design.

Hrischuk et al. [1995]'s approach is similar to our work in the monitoring trace format and in generating a software architectural model from the traces. However, we use the traces and architecture for timing behavior analysis and anomaly detection for production systems, in contrast to generating formal prediction models for software development.

Hrischuk et al. [1999]'s later work additionally supports asynchronous communication and message forwarding, while the earlier work and our approach only support call-return communication.

Bond and McKinley [2007] presented a method to efficiently track the stack context during software execution. This uses a heuristic called *probabilistic calling context*, which represents a stack context by a single number. A major contribution of this work is on the technical implementation of trace monitoring. Our work could use the technical method of Bond and McKinley [2007] to monitor trace shape information more efficiently in large software systems.

Other Trace Analysis Approaches and Commercial Tools

There are several research approaches that are weakly related to our trace analysis. Stardust [Thereska et al., 2006] provides call path profiling for data storage software systems. WAP5 [Reynolds et al., 2006b] extends Aguilera et al. [2003]'s work in practical issues of trace monitoring. Chen et al. [2002]'s, and Kiciman and Fox [2005]'s approach, called Pinpoint, also monitors and analyzes traces but without correlating it to timing behavior. Pinpoint is discussed in detail in the related work on fault localization in Section 6.2.

Furthermore, there are several trace monitoring and analysis approaches used for *program comprehension*. These techniques apply high levels of abstraction on traces to achieve compact models for large traces. This is not directly related work, as it does not focus on correlating timing behavior and traces. Hamou-Lhadj and Lethbridge [2004] and Hamou-Lhadj [2005] provide surveys on program comprehension techniques.

Two types of commercial tools are related: application performance management (APM) software and profilers. APM software operates in

production contexts to operate large multi-user software systems. A comprehensive market overview of commercial tools in this domain is provided by Gartner (see Cappelli and Kowall [2011], and Gartner [2010]). Typical features are user experience monitoring (e.g., end-to-end response times) and recording traces corresponding to user requests through all tiers of large software applications [Cappelli and Kowall, 2011]. These commercial APM tools relate to our trace-context-sensitive analysis by usually providing means to monitor request traces in distributed systems. However, these tools do not provide trace-context-specific timing behavior analysis.

Profiling tools provide a detailed view on the timing behavior and resource demands of a software program (see Foundations 2.1.2). This enables developers to identify bottlenecks and potential optimization points. Xie and Notkin [2002] provide a summary and comparison of commercial profiling tools. Some profiling tools, such as gprof, Google's perftools, Valgrind, NetBeans profiler, and Java's HPROF are demonstrated in Appendix D.

6.1.2. Workload-intensity-sensitive Timing Behavior Analysis

Related work for the WiSTA workload-intensity-sensitive analysis, introduced in Chapter 5, automatically relates timing behavior monitoring data to workload intensity metrics. This assumes that workload intensity is a significant influence to timing behavior.

The literature on formal performance models and performance prediction are more foundations (Section 2.1.2) than related work for this thesis. However, formal performance models often consider workload intensity metrics as input parameter. These techniques are often based on queueing theory, which requires strong mathematical simplifications (e.g., simple distribution families, and only focus on mean values) to be mathematically tractable. Our work statistically analyzes measurement data without these simplifications. Additionally, our work is not on performance prediction, but on the timing behavior of software systems that already exist.

This discussion of related work excludes profiling approaches that do not automatically analyze the relationship between workload intensity and timing behavior.

Maxion [1990] evaluated network characteristics in the context of workload intensity changes by considering the time of the day. Additionally, weekdays and weekend days are modeled separately. The approach is based on the observation that these two time metrics correlate to workload intensity patterns.

Our approach differs from Maxion [1990]'s work by addressing software timing behavior instead of network characteristics. Furthermore, Maxion [1990]'s approach only covers workload intensity changes directly related to the time of day and day type (i.e., weekend or not). Our work indirectly covers these changes also to some extent, because time of day and day type often correlate to the input for our input metrics (e.g., number of parallel requests, and type of requests). Maxion [1990]'s work has lower monitoring requirements than our approach.

A similar approach to Maxion [1990]'s work is by Frotscher [2013]. These authors observed weekly patterns in the response times of a large business-oriented social networking service and use this observation in anomaly detection. The approach is discussed in more detail on Page 136.

Zhang et al. [2005] correlated different types of monitoring data from lower system layers for diagnosing service level objective (SLO) violations. The authors argue that the combination of multiple types of low-layer sensors (e.g., operation response times and CPU utilization) is necessary in diagnosis because single low-layer sensors are inadequate to model SLO states for complex workload. Workload changes are explicitly considered in this analysis by using different combinations of low-layer sensors depending on the workload. Our work uses application-layer monitoring instead of monitoring on lower layers and does not change sensors depending on workload.

Kelly [2005]'s timing behavior anomaly detection approach, which is also related work on trace-context-sensitive timing behavior analysis (see Page 119), indirectly uses workload intensity in timing behavior analysis. A central metric in this approach is the *sum* of response times during a time window (e.g., of 5 minutes). This sum is an indirect workload intensity metric, because the sum obviously depends on the amount of workload. Kelly [2005]'s approach differs in workload intensity analysis from our work mainly in three points:

1. We use a non-linear relationship for workload intensity and response times,

2. we allow different response time probability distributions for each workload level and each operation, and

3. we use workload intensity to evaluate single response times, instead of using a sum of response times.

Concerning the first point, our approach uses a non-linear relationship, in contrast to this work, because a linear relationship between workload intensity and timing behavior is a strong simplification (see foundations on Page 12).

Regarding the second point, our work uses independent timing behavior distributions for each workload level (i.e., each workload-intensity-level has a specific distribution of how likely short and long response times are). This assumes that the timing behavior distribution can change for different workload levels. For instance, some virtual machine environments are able to change the amount of available resources for different workload intensity levels. The empirical data shown in Figures 5.8(1) and 5.8(2) (Page 103) indicates that the standard deviation of timing behavior measurements increases by increasing workload intensity.

Cheung et al. [2011] predict response times for high workload intensity scenarios only based on measurements from low and medium workload intensity scenarios. This is motivated by the costs of high-workload intensity tests in production systems, especially in combination with third party Web services. It joins regression techniques with queueing network modeling for performing the prediction. Cheung et al. [2011] demonstrated that a combination of regression techniques with queueing network analysis outperformed both techniques alone.

Similar to our approach, it only uses response times without modeling resource demands, as these may be unavailable for third-party Web services. Our WiSTA approach assumes that most workload intensity scenarios are within the historical monitoring behavior. If this cannot be sufficiently satisfied or if too many false alarms in anomaly detection in rare workload situations occur, then our approach should integrate Cheung et al. [2011]'s work. However, this work only predicts *average* response times, but our approach requires response time distributions for each workload intensity level. We avoid using the average response time in timing behavior anomaly detection, as it could be an unsuitable representative value: Timing behavior distributions can have heavy tails

and multimodality (see Foundations 2.1.3), which can strongly influence the mean response time and reduce its robustness.

D'Alconzo et al. [2009] present an anomaly detection approach for 3G mobile network traffic. It categorizes monitoring data according to the time of day (e.g., one bin for 5 p.m. to 6 p.m.). To identify anomalies, new observations are compared with historical observations of equivalent bins (e.g., the same time of the previous day). This assumes that the time of day correlates to workload intensity or to other relevant workload changes, such as the type of usage.

The comparison is based on the distributions of historical and current timing behavior, similar to our work. A major difference in analysis is that our work does not use the time of day at all. We monitor workload intensity and directly use this in timing behavior analysis to deal with varying workload intensity.

Zhang et al. [2007]'s approach R-Capriccio shares many elements with our work, but uses different metrics and techniques. It also detects anomalies in distributed software systems based on performance measurements.

R-Capriccio monitors workload and CPU utilization, and creates a statistical model to estimate CPU-utilization for a given workload mix. The workload metric is the number of transactions, which are the end-to-end traces through one execution environment (e.g., an application server). These transactions are identified for instance by URL addresses in middleware log files. The statistical model makes the anomaly detection resistant to workload intensity changes: It can be distinguished whether high CPU-utilization arises from the workload, or whether it indicates a performance problem. This shares the general motivation with WiSTA (Chapter 5).

Both the concrete input and output metrics are different in our workload-intensity-performance model of WiSTA: Our workload metric consists of the number and type of parallel software operation executions and the resulting output are software operation response times partitioned by workload intensity. TracSTA's input metric (Chapter 4) for estimating response time partitions are software operation execution sequences corresponding to a trace shape. Both our metrics require more intrusive monitoring instrumentation than R-Capriccio's black box metrics, but provide a look into a software application.

R-Capriccio aggregates measurements into time windows (e.g., one hour), while we avoid aggregating input data. Furthermore, R-Capriccio's anomaly detection is without an explicit fault localization step, i.e., no event correlation or fault localization technique supplements the anomaly detection, in contrast to our work.

Finally, R-Capriccio creates a layered queueing network model, which is a formal model for analytical performance analysis (see Foundations, at Page 14). In contrast, we use statistical- and measurement-based performance analysis. R-Capriccio uses the queueing network model for capacity planning and performance prediction, which are both not addressed by our work.

Herbst et al. [2014] present a forecasting approach for workload intensity. The time-series-based approach for instance predicts the workload intensity in terms of the number of request per hour and the number of transactions per 30 minutes in the evaluation. A major contribution of this work is that it dynamically selects the most adequate forecasting method for each point in time. The selection is based on the characteristics of the time series for which prediction are to be created. Examples for these characteristics are the length of the time series, the burstiness (based on maximum and median), and the number of consecutive monotonic values.

This approach shares with our work that the timing behavior analysis is workload-intensity-specific. The workload intensity metrics in our work and in this work differ: Our *pwi* metrics quantify the amount of current usage in terms of (weighted) parallel software operation executions during a software operation execution (i.e., usually a very short duration, such as 10 milliseconds). The workload intensity metrics of Herbst et al. [2014] are time series of request arrival rates and transactions per time unit over time periods of minutes and hours. Our work neither uses time series analysis nor forecasting. Herbst et al. [2014] provide a forecast of workload intensity while our work provides workload-intensity-specific software response times.

6.1.3. Other Related Context-Sensitive Timing Behavior Analysis

In the following, timing behavior approaches are presented that are similar to our work in considering context information, but without a focus on

trace shape information or workload intensity. This includes for instance timing behavior approaches that explicitly consider workload characteristics (e.g., file sizes and parameter values), platform metrics (CPU utilization, and other resource demands), relative service call frequency, and calendar time for improving analysis results.

Bailey and Soucy [1983] group service requests into three complexity classes: trivial, intermediate, and complex. This categorization aims to differentiate resource demands, such as CPU, network, and storage. Each class has a predefined response time objective, defined as the 90th percentile of the corresponding historical response times.

Bailey and Soucy [1983] only distinguish three requests classes, while we distinguish requests based on workload intensity and trace contexts which provides much more fine grained analysis. Furthermore, their approach disregards workload intensity changes within the system. Therefore, the occurrence of an unusually high number of users would raise a false alarm.

Menascé and Almeida [2001] use cluster analysis to identify workload classes as preprocessing for performance analysis. The authors see the requirement to separate classes to compensate the large variability in Web-based workload characteristics (e.g., file sizes). Otherwise, this variability "reduces the statistical meaning of measurements" [Menascé and Almeida, 2001, p. 168]. A small example demonstrates the clustering of requests to a Web server based on file size and access frequency [Menascé and Almeida, 2001, p. 243 ff.].

This shares with our work the strategy to categorize measurements before applying performance analysis. The major difference to our approach is that our classes are defined by different metrics. In detail, in Chapter 4 classes are defined by an execution's corresponding trace context (preceding and succeeding operations) and in Chapter 5 executions are categorized using workload intensity metrics. Furthermore, we present an empirical evaluation, while these authors demonstrate by example.

Menascé and Almeida [2001]'s clustering is an alternative to our workload intensity binning method. However, this requires the additional assumption of significant clusters in workload intensity.

Our approach in Chapter 4 also uses a clustering method. However, we cluster trace contexts and timing behavior, instead of clustering workload.

More precisely, our approach iteratively merges trace contexts to *reduce* the total number of trace classes. For this, it uses a different similarity metric than the one used by these authors: Only classes are merged that are both in a particular relation in the trace shape context tree and show similar timing behavior characteristics.

Koziolek [2008] and Koziolek et al. [2008] consider the parameter values of requests to increase performance prediction precision. This approach focuses on performance prediction during early software design, while our focus is on the evaluation of measurement data from software systems in production environments. Our trace context analysis and workload-intensity-sensitive timing behavior analysis are different to the work of these authors, besides the general idea to use context information of a call.

These authors provided empirical evidence for a correlation between parameter values and timing behavior. This suggests that our analysis could benefit from considering parameter values as well.

From the same research group Palladio, Becker et al. [2007] allow one to specify different timing behavior for different types of calls of the same operation (or service). The main focus is on timing behavior modeling for performance prediction before implementation, but it may be also useful in the context of analysis of monitoring data.

Bulej et al. [2005] and Kalibera et al. [2005] analyze timing behavior in the context of observed clusters in timing measurements. This timing behavior analysis specializes in regression benchmarking. Regression benchmarking can identify performance changes between two software versions.

The clusters are identified by k-means clustering on timing behavior measurements. This clustering could decompose multimodal timing behavior distributions (see Foundations 2.1.3.2) into non-multimodal distributions. Therefore, it is a heuristic alternative to our trace-context-sensitive timing behavior analysis.

Our approach requires additional monitoring (trace shape monitoring) for determining trace contexts and workload intensity information. The approach of these authors requires no additional monitoring, because the clustering operates on the timing behavior values that are monitored anyway.

Our approach uses the trace information as additional information in analysis, which allows the precise distinction of timing behavior classes (if there are correlations to the trace contexts). The k-means clustering approach is a heuristic that performs well, if the correct number of clusters is known in advance and the values of the clusters are well separated. Our method is more precise and accurate for clusters that correspond to trace contexts. However, multimodality can also be caused by lower system layers that are not covered by our application-layer monitoring.

Tan et al. [2010]'s ALERT approach *predicts* anomalies using context-sensitive prediction models. The prediction models are based on decision trees and predicates (e.g., CPU utilization larger than 50 %) that evaluate a large set of platform metrics, such as CPU utilization, memory utilization, input data rate, and buffer queue length.

Similar to our approach, it motivates context-sensitive system behavior analysis and it considers workload intensity. However, it is not focused on software timing behavior. From the set of metrics, which are sampled every 10 seconds, time windows are distinguished by clustering. In other words, each time window belongs to a cluster that represents a context. Such a context may for instance represent the low workload period during the early morning and another context may correspond to the afternoon with high workload intensity. Each context has its own anomaly prediction model, which is therefore specialized for that particular system behavior context.

The contexts of Tan et al. [2010] are valid for some time window and cover during that time all evaluation activity. Our trace contexts and workload intensity contexts are specific to software operation executions and not to time periods. Our workload intensity metrics focus on parallel application layer software operation execution activity, while Tan et al. [2010] use many platform metrics.

Williams et al. [2007]'s approach called Tiresias is a failure prediction approach. Failure prediction is different to our work on fault localization and failure diagnosis in Chapter 3, but this approach also uses timing behavior anomaly detection and it is able to tolerate workload intensity changes. Tiresias predicts failures in two steps: (1) anomaly detection, and (2) prediction failures from anomalies. The authors demonstrated the prediction of some failure up to several hundreds of seconds in advance, while spontaneous crash failures were not predicted.

Several monitoring metrics are used and compared with each other, such as network traffic metrics, application response time, platform metrics (CPU utilization, memory utilization, context switches), and protocol metrics. From these metrics, response time preformed best.

The anomaly detection is based on the approach of Maxion [1990], which uses data smoothing with mean and median filters, and then identifies upper and lower thresholds based on standard deviation ($\pm 3\sigma$). Tiresias's prediction applies the Dispersion Frame Technique (DFT), which is commonly used to predict hardware disk drive failures. The DFT analyzes the inter-anomaly-times.

The authors tolerate workload intensity changes by creating multiple normal behavior profiles in correspondence to seasonal workload changes. For instance, it uses different profiles for off-peak times and peak times. We also recognize the need to deal with seasonality in performance but use a different approach in Chapter 5: we always measure the workload and have a generalized workload performance model that can estimated the response times for a given workload intensity and workload mix. Our approach can tolerate also non-seasonal changes in the workload, e.g., when a new special time-based offer attracts many customers to a Web store.

Our approach is similar in using timing behavior anomaly detection, with an emphasis on response times to analyze failure processes. Tiresias is different in using strong aggregation methods (mean and median) before anomaly detection. It is argued that this avoids that strong discontinuities “skew the data”[Williams et al., 2007]. However, it can also be argued that smoothing at this stage can also remove too many outliers and it adds additional parameters that could be wrong.

Rygielski and Tomczak [2011] present a change detection approach that monitors workload intensity in terms of the call frequency of each service. These relative service frequencies form a distribution for a time window. A change is reported, if the current distribution is too different to the historical distribution. It is assumed that changes in the distribution of service usage indicate a context change that might require a system reconfiguration for providing optimal service quality.

This approach basically detects changes in the workload mix. Our approach explicitly aims to *avoid* detecting changes in workload - it aims to detect timing behavior changes that are not caused by workload changes. Therefore, we assume that changes in workload are normal

behavior. Both viewpoints on anomalies are valid depending on the types of anomalies to detect.

This work is similar to our approach in monitoring workload intensity for request types, but it uses this information not for timing behavior analysis.

Furthermore, Rygielski and Tomczak [2011]'s workload intensity metrics go not beyond counting the service requests within a time window. Our metrics use operation-specific weights to address that some software operations require more shared resources than others, if executed in parallel.

6.2. Fault Localization and Failure Diagnosis for Software Systems

This section discusses related work to our fault localization approach presented in Chapter 3. First, research that analyzes timing behavior is discussed in Section 6.2.1. Next, approaches that use other runtime behavior characteristics than timing behavior are addressed in Section 6.2.2.

The discussion of related work excludes high performance computing (HPC) approaches, approaches for fault localization by source code analysis, and debugging techniques that analyze failed and passed test runs.

Many failure detection approaches exist for HPC systems, such as DIDUCE [Hangal and Lam, 2002], AccMon [Zhou et al., 2004], and DM-Tracker [Gao et al., 2007]. These are distant related work, as they are specific to HPC systems. The large class of approaches that localize bugs in source code by code analysis is excluded, as these approaches use static code analysis instead of dynamic analysis (i.e., monitoring). In other words, runtime behavior is neither monitored nor analyzed. Examples for this class are the work of Wasylkowski et al. [2007], PR-Miner [Li and Zhou, 2005], and FindBugs [Hovemeyer and Pugh, 2004]. Typical software fault localization methods that analyze failed and passed test runs to determine bugs in a software program, such as spectrum-based fault localization [Reps et al., 1997], Tarantula [Jones and Harrold, 2005], and the work of Zeller [2002] are not discussed as related work (see Foundations on Page 28). These techniques focus on the development time and localize faults in software programs, while our work is on runtime analysis of software timing behavior monitoring data to localize

problems in software systems. Additionally, our work differs to these approaches by not using test casts and it is not known to our approach whether a single run (or user request) has failed or not.

6.2.1. Fault Localization and Failure Diagnosis based on Software Timing Behavior Analysis

In the following, related work is presented that analyzes timing behavior for failure diagnosis and fault localization.

Bailey and Soucy [1983] is up to our knowledge the first approach to diagnose failures in a software system using response times. As also discussed on Page 128, Bailey and Soucy [1983] group requests into three classes: trivial, intermediate, and complex.

Response time thresholds are determined for each class based on the 90th percentile of historical measurements. The response times are monitored at client side and regularly transferred to a central server to enable evaluation of actual data, for comparison to predefined profiles, and for exception reporting. Our failure diagnosis approach goes beyond this approach by:

1. using more context information: workload intensity and trace context, instead of three manually defined complexity classes,
2. comparing current measurements to historical distributions, in contrast to comparing current measurements to the 90th percentile of historical measurements, and
3. in correlating anomaly scores in a fault localization step, in contrast to just reporting response time objective violations.

Yilmaz et al. [2008]'s Time Will Tell (TWT) approach uses timing behavior analysis for fault localization, like our approach. Furthermore, it also uses caller context information: execution times are evaluated in the context of execution times of subcalls.

A Gaussian Mixture Model (GMM) statistically learns the normal timing behavior. TWT includes the execution times of callees into the GMM model. This means that a software operation execution time is evaluated in the context of execution times of callees. Our approach TracSTA also looks at caller-callee dependencies, but uses the operation type and the

relation of the executions in the corresponding dynamic call tree, and not the timing behavior of those elements.

Yilmaz et al. [2008] do not explicitly consider workload intensity changes, but the GMM might tolerate some of these. However, it may be beneficial to integrate our workload intensity metrics as additional parameter into the GMM, as it directly addresses workload intensity.

The fault localization algorithm in TWT returns a list of the methods, ranked by a score that quantifies its deviation from normal timing behavior. We use an additional analysis step by iteratively employing rules to anomaly ratings. Such an event correlation step could also extend TWT to derive root causes from symptoms.

Wang et al. [2014] present a similar anomaly detection approach to our work that shares the idea to analyze system behavior in the context of workload. More precisely, both approaches use two types of context information to address that system behavior depends on both the type and amount of usage. Furthermore, other concepts, such as using the LOF (local outlier factors) are shared. Wang et al. [2014] focus on anomaly detection in (non-distributed) Web applications, without focusing on software timing behavior.

We use WiSTA and TracSTA to achieve anomaly detection context-specific to workload intensity (e.g., amount of concurrent execution within the same execution environment) and trace shapes. These authors use other workload intensity metrics (request arrival rate and the time between requests within a session) and access patterns. The access patterns are modeled from the usage of external access points of a Web application. Additionally, our work looks deeply into the software application by using information from application-layer monitoring, while Wang et al. [2014] do not look inside the software application (i.e., only the platform and external interfaces are monitored).

Another major difference is that Wang et al. [2014] aim to localize the resource (CPU, memory, disk, network) as root cause, while our approach aims to localize the root cause within the software architecture of the software application under analysis. These are orthogonal views on a system and could be combined. This difference is a reason for the selection of metrics that are used for localization; in our case application-layer metrics (software operation response times), while Wang et al. [2014] use system-layer metrics.

Agarwal et al. [2004] provide a fault localization approach based on timing behavior and dependency graphs. A particular contribution of this approach is to derive timing behavior thresholds for software components-based on end-to-end SLA (service level agreement) violations. These component specific thresholds are used for failure diagnosis. In contrast to our approach, the existence of end-to-end service level agreements is assumed. The training requires historical data for both violations and non-violations. Our approach and most other anomaly detection approaches for failure diagnosis based on timing behavior only use historical data that is assumed to be free of anomalies. A dependency graph analysis uses the anomalies to localize the root cause from the anomalies, similar to our approach. The final result is a ranked sequence of components ordered by their likeliness to be the source of the problem. The approach uses average response times as reference for normal and anomalous timing behavior - our approach uses probability distributions for defining normal behavior.

Ehlers et al. [2011]'s and Ehlers [2012]'s work, also discussed on Page 121, demonstrates an adaptive monitoring approach in the context of timing behavior anomaly detection. Additionally, the authors compare two ways (time-series forecasting and probability distributions) to specify a normal behavior reference in anomaly detection. A major difference between both is that distribution-based methods do not use information about the order in which historical measurements occurred and that trends and seasonality are not continued. The evaluation showed similar performance for both methods [Ehlers, 2012].

Major differences to our work are that these works aggregate timing behavior over time windows, different anomaly detection algorithms, such as population-mean-based hypothesis tests are used, and that forecasting methods are applied. Furthermore, it uses different event correlation methods to determine the root cause of an anomaly. Our approach's monitoring overhead could be optimized by Ehlers [2012]'s adaptive monitoring.

Bielefeld [2012] introduces an online anomaly detection approach, called ΘPAD for identifying anomalies in end-to-end response times. A focus is on forecasting algorithms, such as ARIMA [Box and Jenkins, 1990] and exponential smoothing [Hyndman and Khandakar, 2008].

It is demonstrated that forecasting methods can deal with workload

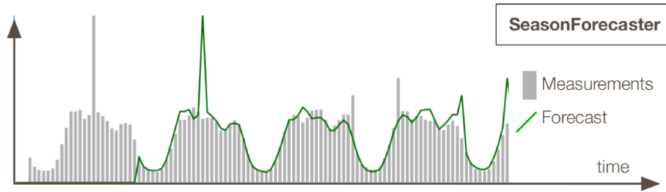


Figure 6.3.: Response times during four days of seasonal data and a seasonal forecasting (green line) for a window size of 24 hours. Image by Bielefeld [2012].

intensity changes, as shown in Figure 6.3. This is an alternative to our approach, which explicitly models this relationship at the cost of having higher monitoring requirements than ΘPAD. ΘPAD's case study showed clearly seasonal patterns, as displayed in Figure 6.3 (i.e., the response times highly depend on the time of day), but still it is relatively difficult to achieve better anomaly detection with forecasting than by using historical observations.

Major differences of our and their work are that both workload intensity and trace context information are ignored in ΘPAD, and it does not contain a separate fault localization step (e.g., by using event correlation). Our approach uses historical distributions instead of forecasting. Additionally, our work considers single software operation response times, in contrast to end-to-end response times aggregated over time windows (i.e., of one minute). ΘPAD calculates anomaly scores based on a threshold, while our approach computes the score based on probability distributions without a threshold.

Frotscher [2013] approach called ΘPADx extends Bielefeld [2012]'s ΘPAD. Similar to ΘPAD, it applies time-series-analysis-based timing behavior anomaly detection in a large social-network platform.

Compared with ΘPAD, ΘPADx supports more performance metrics and has improved applicability. A major difference to ΘPAD is that collective anomalies are distinguished from point anomalies. Collective anomalies are a number of consecutive (point) anomalies. These cause false positives *after* a longer period of anomalies ends, if the anomalies are considered normal behavior after a while and are used for forecasting. After a collective anomaly is detected, Frotscher [2013]'s anomaly detector switches from forecasting-based anomaly detection to a pattern matching

anomaly detection. This compares current timing behavior to historical measurements for the same day of the week and for the same time of the day of previous weeks. ΘPADx provides some improvements compared with the earlier work of Bielefeld [2012], such as fewer false alarms especially at the end of longer anomalies.

Most differences between this work and our work are similar to the differences described in the discussed of the approach of Bielefeld [2012] on Page 136. Compared with ΘPAD, ΘPADx is more similar to our work, as it uses in some cases historical observation as reference.

Avritzer et al. [2005, 2006, 2007] use anomaly detection and timing behavior analysis to detect performance degradation. Degradation can be an early indicator of a failure and may be addressed by rejuvenation means, such as a system restart, to prevent that a failure occurs.

Avritzer et al. [2006] analyze software response times with three different algorithms. These mainly use aggregation and the central limit theorem for change detection. The authors average single response times, based on the assumption that short-term effects of high response times are normal. The central limit theorem states that if sufficiently many values are used to compute a single average value, then the distribution of the average values will tend to follow a normal distribution. This approach detects an anomaly if an average value (i.e., a sample mean) is above the 0.975 percentile of the normal distribution (of all averages). The central limit theorem requires statistically independent values. Forecasting methods that are also used by other approaches for timing behavior analysis and anomaly detection approaches, such as Ehlers and Hasselbring [2011] and Bielefeld [2012], have contrary assumptions (trends and seasonality). Avritzer et al. [2006] regard the self-correlation observed in their experiments to be sufficiently low to apply the central limit theorem.

Both our and this approach use a distribution-based computation of anomaly scores. A major difference between this and our approach is that we evaluate each single response time independently in contrast to response time averages. Another difference to our work is that workload intensity and trace shapes are not explicitly considered within the anomaly detection by these authors.

Diaconescu and Murphy [2005] and Diaconescu et al. [2004] present an approach for online software timing behavior analysis and automatic reconfiguration. When a component is considered responsible for a

performance problem, the system is automatically reconfigured, e.g., by using a functionally equivalent component that shows better performance for the current workload. Various timing behavior anomaly detection metrics are briefly discussed without evaluation.

In contrast to our work, both workload intensity variations and trace context analysis are not addressed. The authors identify timing behavior anomalies for instance by using static thresholds, while our approach uses mainly distribution functions. It uses the COMPAS monitoring [Mos and Murphy, 2004], which automatically adapts the monitoring infrastructure during runtime, similar to the work of Ehlers [2012].

Pitakrat et al. [2014] and Pitakrat [2013] describe an online failure prediction approach for component-based software systems, called Hora. On the component layer (soft- and hardware), each component is monitored (e.g., software response times) and a prediction model is created to estimate the future quality of the service of the component. On the system layer, the component layer predictions are combined into a 2nd-order continuous-time Markov chain to predict if and when the component layer problems lead to a system layer failure.

Both approaches share the general idea to start with a component layer analysis of online monitoring data and combine the component layer results into a system layer model that uses dependencies from the structural system architecture. Hora goes beyond our approach by focusing on failure prediction and supports architectural changes while our approach focuses on fault localization and does not address architectural changes. Our work focuses on response times, while Hora is not limited to software timing behavior.

6.2.2. Fault Localization and Failure Diagnosis based on other Runtime Behavior than Timing Behavior

In the following, failure diagnosis or fault localization approaches are presented that perform the analysis on other runtime behavior characteristics than timing behavior, such as trace shapes, component interaction patterns, event sequences or heterogeneous metrics from all system layers (e.g., CPU utilization, network statistics).

Chen et al. [2002] and Kiciman [2005] introduce an approach called Pinpoint for failure diagnosis based on anomaly detection in enterprise

systems. Chen et al. [2002] detect anomalies in trace shapes and use clustering to determine the root cause of a failure. Kiciman [2005] extends this work and detects and localizes anomalies from both component interactions and path shapes. Path shapes are an alternative representation for component execution sequences of requests. For anomaly detection, statistical tools compare the current system behavior with a historical reference model. Kiciman and Fox [2005] combine three methods for failure diagnosis: 1. request-path-based anomaly detection, 2. inter-component interaction frequency anomaly detection, and 3. decision-tree-based diagnosis.

In contrast to our work, these authors focus on component interaction anomalies and trace shape anomalies, while our focus is on timing behavior anomalies.

Chen et al. [2007] use component interaction patters for failure detection and diagnosis. Note, this strategy is similar to the work of Chen et al. [2002] discussed above and even use the pinpoint approach for monitoring, but the first authors of both works are different persons. This approach represents component interactions as a matrix containing the directed interaction frequencies. For n components the matrix size grows to n^2. Chen et al. [2007] introduce a way to find a subspace and particular statistics that efficiently represent the component interactions during a particular time (window) as density distribution. Failures are detected by comparing the current distribution representing all current interactions to a distribution corresponding to historical interactions. A final step localizes faults by comparing each component's current interactions to its historical interaction frequencies.

As our and this approach use different fault indicators, both approaches can be combined for improving diagnosis quality. The monitoring demands of Chen et al. [2007] are a proper subset of our monitoring demands.

Williams et al. [2007] diagnose failures in distributed systems by local (i.e., for one node) anomaly detection and global anomaly correlation. For each node of a distributed system, several metrics on the operating system layer or protocol layer are monitored, such as available memory, transferred packets per second, and context switches per second. Anomalies are detected independently on each node by comparing the newest observation for each metrics to a weighted combination of previous ob-

servations. In more detail, upper and lower thresholds are based on the mean and standard deviation of weighted previous observations. The more recent observations have a higher weight than older observations. For instance, the current packets/sec measurement is considered an extreme anomaly if it is outside $[\mu - 6\sigma, \mu + 6\sigma]$. To filter noise, anomalies are only logged if at least 50 % of the values of some window (e.g., of 15 observations) are anomalies. Global failure diagnosis is automatically triggered if at least 3 of 7 logged values are anomalies.

The authors present three different types of event correlation methods to compute a global failure diagnosis from local anomaly detections. The first is a heuristic approach based on rules (e.g., the node showing anomalies in most of the metrics is considered to be faulty). The other two metrics use variants of k-means clustering. Our approach in Chapter 3 also uses rules in this context.

This and our approach use disjunct monitoring data for diagnosis. Therefore, both approaches can be combined to achieve better diagnosis quality. Another major difference is that this approach uses σ-based thresholds (similar to Shewhart [1931]; see Foundations 2.3) to identify anomalies while we determine anomaly scores based on probability distributions.

Cohen et al. [2004, 2005], Zhang et al. [2005] support the diagnosis of SLOs (service level objectives) violations using combinations of low-layer metrics (below software application layer), such as CPU utilization and memory utilization. The core of the approach is named *metric attribution* [Cohen et al., 2004]. Out of many low-layer metrics, those with the strongest correlation to the SLO violation are presented to administrators as support in diagnosis. For instance, for a particular SLO violation, this technique might present “available database connections” as diagnosis advice.

Zhang et al. [2005] explicitly address workload changes. Depending on the workload conditions, the diagnosis approach uses a different combination of metrics. It is argued that a combination of multiple types of metrics is necessary, because single metrics only perform best for particular workload situations.

Cohen et al. [2005] recognize known system states based on combinations of low-layer metrics. The metric combinations act as fingerprint for the system state over a time window. A SLO violation’s corresponding fingerprint may match to a historical fingerprint with a known diagnosis.

This can reduce diagnosis time.

Both this and our work share the idea to statistically analyze monitoring data for improving diagnosis. Besides this, both approaches use different strategies. A major difference is that these authors use many low-layer metrics, while our approach focuses on software operation response times and software operation execution sequences (i.e., the traces).

Jiang et al. [2006] detect anomalies in changes in the *relationship* between measurement data from heterogeneous monitoring probes (e.g., the relationship between CPU load and the number of SQL queries during 10 seconds). This differs from our and most other approaches that detect anomalies directly in measurements. These linear pairwise relationships are called invariants. The applicability of the approach is demonstrated. However, no empirical evidence was provided that invariants are superior to the classical approach.

A lab study with a three-node distributed system instrumented with 111 monitoring probes from all system layers (e.g., CPU load, number of requests, number of SQL queries, network statistics) resulted in 975 sufficiently robust linear invariants for anomaly detection. The measurement data is aggregated (10 second windows) to reduce overhead. Furthermore, the anomaly detection uses time series forecasting.

The work of Jiang et al. [2006] shares general elements with our approach, such as applicability in distributed multi-user systems, considering workload intensity metrics, and automatic determination of normal behavior in anomaly detection. However, in some aspects, it is orthogonal to our work:

- The authors use heterogeneous monitoring probes from different system layers. In contrast, our approach focuses on detailed analysis of a few application-layer metrics. Monitoring on multiple layers should provide better *direct* fault coverage. However, timing behavior metrics on the software application layer also cover lower layers to some extent, as they are highly influenced by lower system layer activity (see e.g., [Yilmaz et al., 2008]).

- Our work does not aggregate measurement data into time windows. Jiang et al. [2006] claim that it is unrealistic to track all user requests in large systems. However, we demonstrated this in the Web portal of Europe's largest digital photo service provider [Rohr

et al., 2010]. Aggregation reduces the amount of data and may filter noise. However, faults may also cause anomalies in single requests and aggregation introduces additional potentially critical parameters, such as the window size and the aggregation method.

- Jiang et al. [2006]'s does not consider individual trace shapes and does not reconstruct a software architectural model. Such architectural models play an important role in many event correlation methods for fault localization and in the visualization of anomalies.

Our timing behavior analysis contributions (Chapter 5 and 4) can be integrated into Jiang et al. [2006]'s approach, by using our timing behavior analysis as source for workload-sensitive timing behavior expectations in anomaly detection.

Brutlag [2000] extended the network and system monitoring tools RRDtool and Cricket with anomaly detection. It uses the Holt-Winters forecasting algorithm, which can cope with trends and seasonality, to define a confidence band. An anomaly is detected, if more than a specified number of observations are outside of the band for a fixed-size moving time window on the monitored time series data. This approach focuses on network monitoring, in contrast to our focus on software timing behavior.

Another major difference is that our anomaly detection computes anomaly scores for single observations and uses these for fault localization under the assumption of an already correctly detected failure. Brutlag [2000] mainly focuses on failure detection. Furthermore, measurements are aggregated over a sliding window to achieve a suitably low false alarm rate. If our approach is used for failure detection instead of fault localization, it may require such an aggregation step as well.

Using (Hold-Winters) forecasting is an alternative to our workload-intensity-sensitive timing behavior analysis for satisfying the relationship between workload intensity and other system metrics, such as response times. Our approach has the advantage that it avoids a false positive for low response times caused by an unpredictable workload burst, for which Brutlag [2000]'s approach may raise a false alarm. However, our approach can only compute the workload metrics some time after monitoring, which makes it unsuitable for fast failure detection. Furthermore, our approach only covers trends and seasonality that are visible in our workload intensity metrics, which reflect the amount of workload in terms of parallel software operation calls and traces. Other trends or sea-

sonality (e.g., slower performance during a daily regular backup activity) cannot be considered by our approach.

Elbaum et al. [2007] presented an anomaly detection approach for failure detection that analyzes the sequence of software operation executions and the sequence of user interactions. Those sequences are compared with the historical failure-free sequences using three algorithms:

1. Foreign symbol anomaly detection: An anomaly is alerted if a software operation that was unseen during training is executed. This algorithm was earlier used by Maxion and Tan [2000].

2. Symbol frequency-based anomaly detection: An anomaly is detected if a window contains a particular symbol more or less frequent than in any training data window.

3. Sequence-based or n-gram anomaly detection: Instead of analyzing the sequence for single symbols, as the foreign symbol anomaly detection, sequences of the length n are used to detect anomalies. For instance, the 2-gram sequence (main, exit) is rated as anomaly if not observed during training.

Elbaum et al. [2007]'s evaluation showed the highest true positive rate for the sequence-based approach, while the foreign symbol approach resulted in the lowest false alarm rate.

Our approach differs by detecting anomalies based on timing behavior, in contrast to operation execution sequences and user action sequences. However, before using timing behavior in anomaly detection, we apply the trace-context-sensitive timing behavior analysis, which uses operation execution sequences. These sequences differ in being limited to the operation executions corresponding to a user request, while Elbaum et al. [2007] use the sequence of all executions of a program run. Furthermore, this approach computes binary anomaly scores (i.e., yes or no), in contrast to our continuous anomaly scores.

Forrest et al. [1996]'s work analyzes the sequence of system calls of a software program. Therefore, no instrumentation is required within the software application under analysis. The sequence analysis is slightly more complex than the third algorithm of Elbaum et al. [2007] presented above, as it computes an anomaly score instead of a binary rating. This

sets the number of mismatches in a sequence in relation to the maximum number of possible mismatches.

Timing behavior analysis, workload intensity, fault localization, and analysis of internal software operation sequences are not part of Forrest et al. [1996]'s anomaly detection approach, in contrast to our work.

Reynolds et al. [2006a] introduce a language for expressing expectations on request traces through distributed systems. These expectations are specified manually, or created with help of a tool that records behavior from the system. The approach detects problems by monitoring the system behavior during runtime and comparing this behavior to the expectations.

Our request traces only contain software execution sequences, while Reynolds et al. [2006a]'s traces also contain parallelism, messages (i.e., communication between system nodes or threads) and other events such as log messages. In contrast to our work, timing behavior is not a focus in this work, varying workload intensity is not addressed, and it uses no trace information for context-sensitive timing behavior analysis.

Dallmeier et al. [2005] use the call/return action sequences of objects for fault localization in object-oriented software systems. Similar to the approach of Chen et al. [2004] discussed above, it needs for each request (or program run) a label whether it was successful or it failed. Only parts of the complete call action sequence are used, such as all subsequences with a length of two. The localization step relates the subsequences of the failed requests with those of the successful requests. Dallmeier et al. [2005]'s approach can localize bugs that corresponds to particular object-specific sequence, such as the execution of an operation “close” twice, or before an “open”.

This approach analyzes operation execution sequences that correspond to software objects, while ours analyzes operation execution sequences that correspond to requests. We do this for refining timing behavior analysis for improving anomaly detection, while these authors directly use it for fault localization. The monitoring requirements of this approach are a proper subset of our requirements, except that it must be known for each request whether it failed or not.

7. Conclusions

This chapter concludes this thesis. A summary of the thesis and its key insights is in Section 7.1. A discussion of the approach and threats to validity follow in Sections 7.2 and 7.3, before future work is presented in Section 7.4.

7.1. Summary

This thesis presents TracSTA and WiSTA for analyzing software timing behavior of multi-user enterprise software systems in the context of trace shape and workload intensity. Additionally, a fault localization approach is presented that uses TracSTA and WiSTA in timing behavior anomaly detection.

Software timing behavior shows characteristics, such as high variance and multimodality that makes it often difficult to analyze [Menascé and Almeida, 2001, pp. 168]. This could question the applicability of timing behavior analysis approaches, such as failure diagnosis, online performance management, and regression benchmarking. The combined hypothesis of our two contributions (detailed in Section 1.2) is that a correlation between timing behavior and both trace shape and workload intensity can exists that this can be used to improve timing behavior analysis. TracSTA and WiSTA create context-specific timing behavior models that have lower variance and less undesired characteristics than timing behavior models that ignore that context information. Our lab studies and field studies provide empirical support for the effectiveness and applicability of TracSTA and WiSTA.

In summary, the hypothesis of TracSTA is that a significant part of the variance in software operation response times of enterprise software systems is correlated to the full trace shape (i.e., the trace contexts) and that this correlation can be used to "reduce" the variance from the perspective of subsequent timing behavior analysis steps, such as anomaly detection. The precise hypotheses of TracSTA can be found in Section 4.2 on Page 68. TracSTA assumes that software operations exist in traces

with different shapes; for instance that an operation is used by more than one caller operation, or that an operation has conditional subcalls. Earlier approaches analyzed only parts of the trace shape (e.g., just the direct caller [Graham et al., 1982] or the stack of callers [Ammons et al., 1997]) while our approach considers the complete shape of the trace – including subcalls and operation executions that are later in the trace.

The empirical evaluation of TracSTA provided support for TracSTA's hypotheses, and indicated the effectiveness and applicability of our approach: Most software operations in three case studies had strong correlations between response times and trace shapes. TracSTA reduced more variance than related work approaches. Additionally, TracSTA replaced some multimodal timing behavior distributions by multiple unimodal distributions. Furthermore, the field studies demonstrated that TracSTA can be applied in large software systems during real operation.

In summary, WiSTA's hypothesis is that the variance in timing behavior of multi-user software systems strongly correlates to workload intensity and that this variance can be controlled by considering workload intensity. WiSTA creates workload-intensity-specific partitions of software timing behavior using our novel *pwi* metrics that purely base on application-layer monitoring data. WiSTA learns how operations influence each other's timing behavior, if executed concurrently. This influence can vary because different operations may compete for different resources. The empirical evaluation supported the hypothesis and demonstrated that WiSTA is efficient in controlling the influence of varying workload intensity to software operation response times.

Possible benefits of TracSTA and WiSTA in measurement-based timing behavior analysis are higher confidence, fewer simulation runs, shorter simulation time, and increased statistical robustness. For instance in an anomaly detection scenario, our approach compares a response time during high workload intensity with other response times that correspond to a similar workload intensity level. This prevents that a detector for anomalous response times makes a false alarm because of unusual high or low workload intensity.

The applicability of both TracSTA and WiSTA was demonstrated in large software systems and it is indicated that the two applicability requirements of Section 1.1 (Page 3) can be satisfied by TracSTA and WiSTA. A major characteristic of our approach, which is critical to the continuous operation in production systems, is the relatively low overhead (see also Section 7.2). This mainly results from the low monitoring requirements: only application-layer monitoring is required; lower-layer,

platform-specific monitoring is not required. An additional reduction of overhead is still possible by not monitoring every trace (i.e., sampling). Furthermore, only the instrumentation of a subset of all software operations is sufficient for getting good results with TracSTA and WiSTA. Additionally, our approach is suitable for the application in distributed systems. Both TracSTA and WiSTA require little manual configuration, except from the selection of monitoring points.

Both TracSTA and WiSTA can help software developers, software architects, and performance engineers to develop a deeper understanding of the timing behavior of their software systems in the production environment with real usage. WiSTA reveals timing behavior that is specific to workload intensity; TracSTA reveals the timing behavior that is specific to trace share scenarios. Furthermore, TracSTA could be used as profiling tool. In contrast to typical profiling tools (e.g., see Appendix D), TracSTA can be continuously applied in production environments because of its low monitoring overhead. In combination with a regression benchmarking technique, WiSTA could enable a software developer to evaluate whether software code changes results in performance changes in the production system, without having to set up a controlled experiment to minimize the impact of workload intensity changes.

We presented an automatic fault localization approach for enterprise software systems based on timing behavior anomaly detection that includes TracSTA and WiSTA. The need for automatic fault localization grows because many enterprise software systems are increasingly both business-critical and complex. Additionally, manual failure diagnosis is time-consuming and error-prone. The anomaly detection uses TracSTA and WiSTA to compare response times to context-sensitive timing behavior partitions. This provides starting points to administrators for additional diagnosis and repair. The fault localization results could also be used in an automatic self-healing system – the localization result could trigger a component micro-reboot [Candea et al., 2004] or a change of components [Diaconescu and Murphy, 2005]. The applicability of our fault localization approach was studied in a distributed Java Web application, which was subject to fault injection of different types and severity.

7.2. Discussion

In the following selected aspects and the applicability of our approach are discussed.

Requirements on the software system under analysis. Our approach is intended for enterprise software systems and multi-user Web applications, such as online stores. TracSTA and WiSTA require the presence of concepts such as software operations, software operation response times, and traces (e.g., sequences of software operation executions connected by synchronous call actions and corresponding returns; details on the conceptual requirements are in Sections 3.2, 4.2, and 5.2). Common programming languages, such as Java, C++, and C# support these concepts and probably a large part of the software written in these languages uses these concepts. TracSTA and WiSTA are also applicable but less effective, in systems that are dominated by opposing concepts, such as asynchronous communication, message-based communication, or Publish-Subscribe. WiSTA cannot provide a benefit in software with no variation in workload intensity (e.g., single user software).

Application in distributed systems. Our approach explicitly supports distributed systems. For this, it is required that the monitoring can follow traces that span across multiple execution environments. We implemented this by passing trace identifiers (i.e., markers that identify the trace and the order within the execution sequence). A major applicability issue (of the monitoring) is that the monitoring requires explicit support for each distributed communication framework. The reason for this is that distributed communication is usually implemented by particular frameworks that are not part of the standard framework provided by the platform that belongs to the programming language itself. For our case studies, trace monitoring was implemented for several Web-Service frameworks, such as the Hessian Web Service protocol[1] and parts of Apache CXF[2]. Alternatively to instrumenting specific frameworks, one could reconstruct distributed traces purely based on timestamps. For instance, Aguilera et al. [2003] follow this approach. These authors assume that both the NTP (Network Time Protocol) synchronization keeps the timestamp deviation in a local network below a millisecond and that

[1] http://hessian.caucho.com/
[2] http://cxf.apache.org/

this is a sufficient timestamp resolution. Our experiments with a NTP-synchronization showed that this can provide a good heuristic for trace reconstruction. However, some experiment runs contained invalid traces and it could not be guaranteed that all incorrect traces are identified. Therefore, it was decided to use the reliable method of passing trace identifiers at the price of a more complex monitoring instrumentation with the need for communication-framework-specific extension.

Suitable number of observations. Both WiSTA and TracSTA define context-specific classes by splitting monitoring data into subsets. For later steps of statistical analysis (e.g., anomaly detection, regression benchmarking, and scalability analysis for platform sizing), each class should have a minimal number of elements. This number depends on the concrete statistical methods to be applied: For instance 10 response times may be fine for determining a median, and at least 100 response times were used for determining a probability density function. In the context of the fault localization approach in Chapter 3, which applies TracSTA and WiSTA in sequence, the author of this thesis suggests using at least 2500 response times per software operation. Typical multi-user enterprise software systems should easily provide sufficient monitoring data for many central software operations in hours or days of monitoring.

Selection of monitoring points. In production systems, a trade-off between monitoring overhead, monitoring coverage and monitoring detail has to be satisfied. Full instrumentation of all operations is computationally relatively expensive and produces very long traces in TracSTA. For large software systems a partial instrumentation can be considered more suitable than full instrumentation. We suggest instrumenting at least the entry-points of major components, so that the resulting architectural model shows the major structuring of the software. Furthermore, we avoid software operations that are executed extremely often and software operations with extremely small response times, such as getter- and setter operations, and data transfer objects. This reduces relative overhead and avoids very long traces. For fault localization, too few monitoring points or monitoring points not well-distributed over the architecture may lead to an imprecise localization.

Computational overhead and monitoring overhead. The overhead is a critical applicability issue because TracSTA, WiSTA, and our fault

localization approach are all intended for continuous operation during regular operation in production systems. The overhead can be divided into four categories:

1. **Monitoring overhead:** In summary, our work requires timestamps for the start and end of software operation executions and information to reconstruct traces (see Section 3.2 for details). In a concrete setting, the monitoring overhead depends on the number and position of monitoring points, the execution frequency of the monitored software operations, and the implementation of the monitoring framework. Several monitoring frameworks can fulfill these requirements in general. In summary, our monitoring framework Kieker causes less than a microsecond overhead per software operation execution [van Hoorn et al., 2009]. This corresponds to less than 10 % overhead [van Hoorn et al., 2012]. Other comparable tools have reported up to 10 % [Govindraj et al., 2006] and 3 % [Chanda et al., 2007] in different settings.

2. **TracSTA computation:** Our TracSTA implementation can evaluate about 80,000 software operation executions per second on a standard desktop PC. Case study CS-5.2 required about 30 seconds for 2.5 million monitored executions in 447,471 traces that were monitored during a system run of several hours. In all three TracSTA case studies, the computation time for trace context analysis was much lower than the actual monitoring period. TracSTA's overhead (for each monitored execution) increases linearly by the number of distinct monitoring points.

3. **WiSTA computation:** During runtime, 100,000 executions are analyzed within a minute on a standard desktop PC. It scales linearly with the number of executions. A training phase is required for learning the pwi_4 weight vectors. The training is a multidimensional optimization with about quadratic computations by the number of monitoring points within the same execution environment. Our implementation required several hours training to determine the weight vectors for case study CS-3 with in total 161 monitoring points in the same execution environment. We consider this overhead acceptable, since it is only required during training and a couple of hundreds of monitoring points per execution environment is a minor limitation.

4. **Computation for anomaly detection and fault localization:** The anomaly detection and fault localization implementation can analyze 41,000 traces containing 262,000 executions per minute on a standard desktop PC. A scalability discussion of the selected anomaly detection and fault localization algorithms is outside the scope of this thesis.

In summary, it can be expected that the monitoring causes less than 10 % overhead on the production system. The processing and analysis of the monitoring data can be executed on a separate system and requires several minutes for hours of monitoring data. An additional training phase of several hours is required once (with updates from time to time, e.g., once a month or after large changes to the system). The primary scalability factors are the selection and number of monitoring points and the operation execution frequency.

Virtualization and Garbage Collection are two examples for influences to timing behavior from below the software application layer. Both are also addressed in the Future Work on Page 154. Some virtualization technologies can dynamically change the amount of available hardware resources and add an additional layer to hardware resource access. This can have a strong influence on software timing behavior.

Garbage collection (i.e., automatic memory management) is part of many modern software execution platforms, such as the Java Virtual Machine and Microsoft's Common Language Runtime. We studied in more detail the timing behavior influences of Java's default garbage collection in Appendix E. Our experiments showed that the garbage collection paused several times per hour all operation executions for nearly a second. This demonstrates that the garbage collection can be very relevant for approaches that make conclusions based on response time measurements.

Failure diagnosis based on timing behavior anomaly detection. Others have demonstrated before that timing behavior can be used for failure diagnosis (see Related Work 6.2.1, Page 133). As pointed out several times in this thesis, software timing behavior of enterprise software systems has complex distribution characteristics (e.g., high variance, heavy tails, and multimodality). Our approach can control some of this complexity, which can for instance improve automatic failure diagnosis. However, a part of complexity is left, which can still lead to many false positives in

anomaly detection. Nonetheless, it was demonstrated that the remaining complexity can be acceptable for fault localization in this thesis. The state of research denies making general statements about how many faults lead to timing behavior anomalies. We consider failure *detection* (e.g., [Bielefeld, 2012] and [Frotscher, 2013]) and failure *prediction* (e.g.,[Salfner and Malek, 2005] and [Pitakrat et al., 2014]) even more challenging than fault localization if it is only based on software timing behavior; the additional risk of frequent false alarms during normal operation and the difficulty to determine an acceptable threshold for raising an alarm.

7.3. Threats to validity

In the following, the most relevant internal and external threats to validity for TracSTA and WiSTA are discussed. This especially adresses the quantification of the effectiveness and the possibility to generalize the results to other systems.

Both the evaluation of TracSTA and WiSTA use the metric standard deviation reduction (see Section 4.3, Page 78). It compares the original standard deviation with the standard deviation after applying our approach. This also allows one to compare the benefit to the benefit of related work (e.g., stack context analysis). It was selected, because it is a relatively simple, intuitive, and common variance metric. However, the metric has also certain relevant weaknesses: The standard deviation reduction would be high for very small partitions and random partitioning tends in average to result in a slightly positive benefit for some distribution types. The first risk is reduced by specifying a minimum partition size (e.g., 600 observations). Another risk arises from the use of the metric *standard deviation* within our evaluation metric, as the standard deviation has several weaknesses itself [Kreinovich and Kosheleva, 2012]. A major reason for the weaknesses of the metric is that it uses the mean value and it uses directly each individual observation, which makes it sensitive to outliers. The evaluation could have used more complex alternative variance metrics, instead of the standard deviation, that are less sensitive to outliers. Examples for this could be the winsorized standard deviation [Wilcox, 2010], using the trimmed mean in the standard deviation [Wilcox, 2010], and fractal theory metrics as proposed by Kreinovich and Kosheleva [2012]. However, these metrics have also some weaknesses itself in the purpose of comparing the variance before and after using WiSTA and TracSTA, are more complex, and can be unintuitive.

Evaluating TracSTA and WiSTA with such a metric could be part of future work.

A general external threat to validity for research in this domain and not only for the research in this thesis, is the question, how the research results can be generalized to other software systems. The evaluation of TracSTA and WiSTA each uses three software systems in their specific environment. Two of the three systems are large enterprise software systems, and one of the evaluations took place in the production environment with real user workload. Still, it should be noted that it is unknown (and unlikely) whether these three systems are a representative selection for all enterprise software systems. Other enterprise software systems might show different timing behavior than the systems in our case studies, e.g., depending on the software architecture, the workload, and the hardware setting. There is the risk that very different results occur in other enterprise software systems. However, we expect at least roughly similar results and that our approach provides at least a small benefit in most other enterprise software systems; we would consider it unusual if there is a complete absence of workload-intensity-specific timing behavior and trace-context-specific timing behavior in an enterprise software system with real multi-user workload.

All case studies used Java software systems. A large share of all enterprise software systems are developed in Java [King, 2011]. There is some risk that particular Java characteristics caused the good results on the effectiveness of our approach. However, we expect a similar timing behavior relation between both trace shapes and workload intensity and software response times in systems developed with other modern programming languages in the domain of multi-user enterprise software systems.

We used the Kieker monitoring framework [van Hoorn et al., 2009] in our experiments to record response time monitoring data. Theoretically, it could be possible that Kieker itself introduces the variance that is later reduced by our approach. However, this is unlikely, as Kieker was also used by several other researchers for related topics and it was itself subject of a detailed performance analysis (see Waller, 2014; van Hoorn et al., 2012).

7.4. Future Work

7.4.1. Future work for both TracSTA and WiSTA

This thesis suggests analyzing timing behavior of enterprise software systems in the context of trace shapes and workload intensity. It is future work to extend the quantitative comparison to other types of context information, such as parameter values and parameter sizes [Koziolek et al., 2008], and application state [Kapova et al., 2010]. More examples for types of potentially relevant context information can be found in the section on influences to software timing behavior in Foundations 2.1.4 on Page 18 and in the related work on context-sensitive timing behavior analysis (Section 6.1, Page 115). The comparison with other types of context information should quantify the strength of an influence on software timing behavior. This can help performance engineers and administrators to decide which of the many influences to timing behavior should be monitored. Such a comparision could use a more advanced evaluation metric for the quantification and comparision of the benefits, such as a winsorized standard deviation [Wilcox, 2010] or a metic based on fractal theory (see Kreinovich and Kosheleva, 2012) to further increase the robustness of evaluation results.

As discussed in the previous Section, virtualization and garbage collection are timing behavior influences that can have a strong influence on software timing behavior from lower system layers. Some virtualization technologies change the amount of available hardware resources during runtime. Our approach assumes a constant amount of available hardware resources so far. Potential future work is to overcome this limitation by regularly monitoring the amount of available hardware to normalize the performance measurements. Similarly, the garbage collection activity could be monitored, and response times that overlap with major garbage collection runs could be excluded to prevent false positives in anomaly detection. The approach of Tan et al. [2010] provides starting points for this future work.

TracSTA and WiSTA currently require the manual selection of monitoring points; we suggest using a partial instrumentation of software operations (see Page 149). Future work is to provide an automatic selection of monitoring points and to adapt the selection of monitoring points during runtime. A promising approach for adaptive monitoring is provided by Ehlers [2012]. A starting point could be to automatically start with a full instrumentation and then remove the monitoring points that

have very short response times as these have little logic and a relatively high overhead.

A promising additional application scenario for TracSTA and WiSTA is to enable regression benchmarking for production environments. Regression benchmarking is a kind of benchmarking that (e.g., Kalibera, 2006) compares in detail the timing behavior of two versions of a software. This enables software engineers to determine whether and how changes to the software changed its performance. Regression benchmarking has usually to be executed in the lab to control the workload (both amount and type). However, it is more important to optimize the performance for the production environment and not for the lab. This is difficult because the workload is usually not identical between two monitoring periods in the production environment. WiSTA and TracSTA can control a part of the variance caused by changes in both the amount and type of workload. Therefore, using TracSTA and WiSTA can make statements about the performance of a system more independent from the workload. A benefit of enabling regression benchmarking for production environments is that it allows engineers to benchmark some large systems that have no complete lab setting (e.g., because these systems are distributed across different companies, or because of license costs).

7.4.2. Future work for TracSTA

TracSTA could be extended to model parallel communication within traces. More precisely, our system model would be extended to model within a trace asynchronous call actions between software operations. TracSTA currently models a trace as a sequence of software operation executions for a request (i.e., only synchronous call actions between software operations). This is not a limitation for applying TracSTA in systems with asynchronous call actions, but TracSTA simply ignores asynchronous calls, which can lead to missing some timing behavior correlations. With the advent of multi- and many-core CPU architectures, it can be expected that also asynchronous communication becomes more and more relevant, even inside TracSTA's focus of single user requests.

Other future work for TracSTA could integrate the work from the domain of program comprehension on how to detect loops and recursion in monitored traces. A survey of such techniques is presented by Hamou-Lhadj [2005]. This would reduce the resource demands of TracSTA, as traces would have a smaller representation because of higher abstraction. We expect that this can strongly reduce the number of trace contexts in

scenarios with loops, without losing much of TracSTA's effectiveness. For instance, the current TracSTA approach would distinguish the 5th and 6th execution of a looped software operation.

7.4.3. Future work for WiSTA

Future work for WiSTA is to compare $pwi_1 - pwi_4$ to workload intensity metrics from lower system layers and to explore whether the binning should be replaced by a regression model.

Alternative workload intensity metrics from lower system layers (i.e., below software application layer) could be for instance CPU queue length, network throughput, or CPU utilization. However, such metrics may be unavailable in some execution environments because they are platform specific. An evaluation should quantify the benefit with these metrics in comparison to our application-layer metrics, which supports one to decide whether to use only application-layer monitoring, or whether to additionally monitor and process platform-specific metrics.

WiSTA uses a binning algorithm for building discrete categories of workload-intensity-specific timing behavior. It first splits the *pwi* values into intervals of equal length and then extends the bins until a minimum number of observations is reached, even if bins overlap. Alternatively, regression analysis can be used to create a continuous (i.e., no categories) mathematical model of the relation between workload intensity and response time. Nonlinear regression techniques are required because the relation to be modeled is usually nonlinear (see for instance Appendix A). The regression model would have to estimate a description of the distribution for every workload intensity level – since each workload intensity level can have its own timing behavior distribution (e.g., Figure 5.4 on Page 96), it is not sufficient to just estimate a single parameter (e.g., a mean value). Unimodal timing behavior distributions are often well-described by three-parameter log-normal distributions [van Hoorn, 2007, P. 90]. However, some software operations have multimodal timing behavior distributions (as in the caching example in Section 4.1). A starting point could be to train a polynomial regression model that provides a general distribution (e.g., described by 20 or more points) for a given workload intensity.

7.4.4. Future Work on Fault Localization and Failure Diagnosis

The work of others and our work have demonstrated the feasibility of fault localization based on timing behavior monitoring in continuously running enterprise software systems (see Related Work 6.2.1 on Page 133). A next important step for this area of research would be to move forward from isolated demonstrations in this area to a systematic empirical comparison of different approaches. Already the isolated demonstration is very time-consuming, as many details, such as the system, the system's workload, and the fault load have to be specified. The development of a common benchmark for fault localization in enterprise software systems would be a valuable contribution at this point. In the domain of automatic fault localization approaches that localize bugs based on test runs during debugging, the Siemens suite [Hutchins et al., 1994] (several C programs together with bugs and tests) has been established as common evaluation scenario. The benchmark for fault localization techniques, such as the one presented in this thesis, would need a scenario with large continuously-running multiuser enterprise software systems. Possibly it could be based on established performance benchmarks, because these have non-trivial systems and non-trivial workload. Possible starting points in this direction could be the work of Silva [2008], the DaCapo benchmark [Blackburn et al., 2006], and SPEC benchmarks.

In addition to the comparison with other approaches, a next step for our fault localization approach would be a comprehensive empirical evaluation. Ideally, this would be done in two scenarios: First, in the context of a controlled experiment with a fault localization benchmark setting in the lab as described above. Second, the approach needs additional evaluation in real world systems. This would involve a long term application, since failures happen only occasionally.

After the fault coverage of different fault localization approaches are compared, an additional direction for future research could be to combine our approach with multiple other fault localization approaches. This would have the implicit hypothesis that a combined approach would have advantages over single approaches alone.

Furthermore, it is future work to evaluate the hypothesis that forecasting improves our fault localization approach. For instance Ehlers et al. [2011], Bielefeld [2012], and Frotscher [2013] suggest that new timing behavior observations should be compared with an expected value created by a forecasting method, such as single exponential smoothing (SES),

double exponential smoothing (DES), or ARIMA. So far, our anomaly detection is without forecasting – it compares new values with historical data. Timing behavior curves appear to follow trends and software systems are stateful. Some of these trends could origin in the workload intensity and these are already exploited by WiSTA in our approach (i.e., it *measures* workload intensity and knows its influence to timing behavior, instead of forecasting only timing behavior). A first analysis of own measurement data indicated (e.g., by low autocorrelation) that it is a challenge to robustly apply forecasting method.

Section 3.7 presented a visualization of fault localization results. The current implementation does not allow one to switch the level of detail within a single tool and it only creates visualizations on demand. Future work is to realize a more user friendly and interactive visualization that shows live timing behavior anomaly scores. Additionally, it should provide a function to zoom into fault localization results. A promising starting point is work on software visualization, such as the ExploreViz approach [Fittkau et al., 2013, 2014].

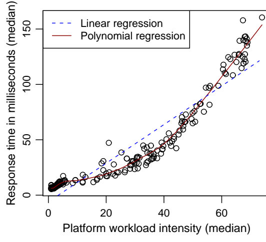
A. Timing Behavior Distribution Examples

Figures A.1–A.5 present details on the relation between the platform workload intensity (pwi_1, see Section 5.2, Page 96) and operation response times. In contrast to the case study CS-5.1 in Chapter 5, the normal non-distributed version of the iBATIS JPetStore is used. Furthermore, 19 operations are instrumented in contrast to the 34 operations of CS-5.1.

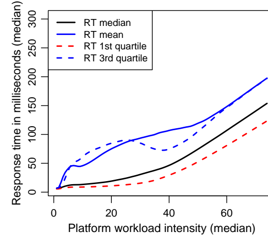
The scatter plots (left images of each pair) present sample measurements in combination with linear and polynomial regression lines. A median filter was applied to the data shown in the scatter plots to remove large outliers. The right image of each pair show additional statistics for distribution characteristics at each workload intensity level. Most operations of Figures A.1–A.5 show a relation between workload intensity and response times that follows the general expected behavior that response times increase by workload intensity. For instance, operation CatalogBean.viewCategory(...) in Figure A.1(5) approximately follows the ideal characteristic workload-response-time-curve of Jain [1991] (see Section 2.1.1.3, Page 12). For all the operations, non-linear regression appears to be more suitable than linear regression for modeling the relation to workload intensity.

Some of the operations, such as those of Figures A.2(1) and A.2(3) do not follow the typical shape, but show distinct behavior for different workload intensity levels. These and some other operations showed the highest response times for medium workload intensity. A possible explanation is that high workload intensity levels activate scheduling strategies that delay the start of operation executions until a relatively fast execution is possible. This prevents high *operation* response times, but not high end-to-end response times that include the additional wait times. Another explanation could be that the operations that have the highest response times for medium workload intensity depend less on the primary bottleneck of the system. In other words, if most workload is, figuratively speaking, in a congestion at one side of a system, then the operations at the other side of the system might have less competition and

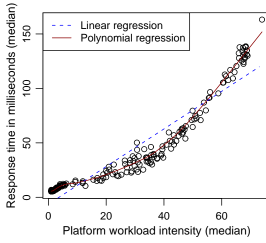
lower response times. Furthermore, our workload intensity metric pwi_1 might be unsuitable in some cases to correctly quantify high workload intensity.



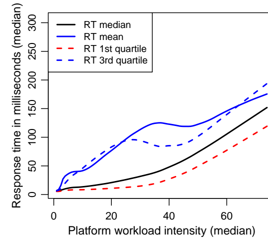
(1) AccountBean.signon(...)



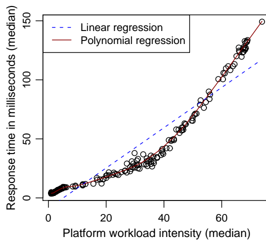
(2) AccountBean.signon(...)



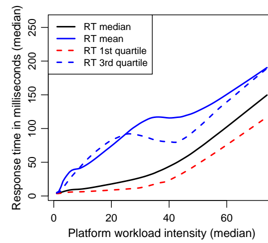
(3) CartBean.addItemToCart(...)



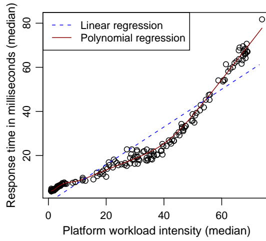
(4) CartBean.addItemToCart(...)



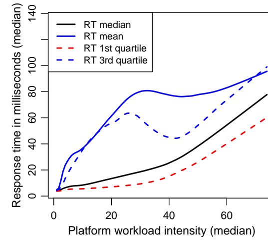
(5) CatalogBean.viewCategory(...)



(6) CatalogBean.viewCategory(...)

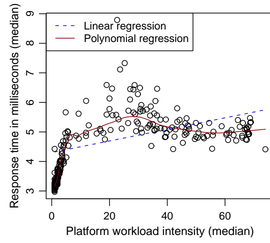


(7) CatalogBean.viewItem(...)

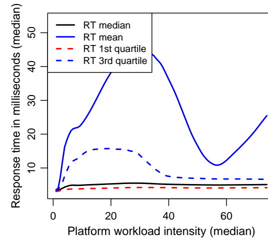


(8) CatalogBean.viewItem(...)

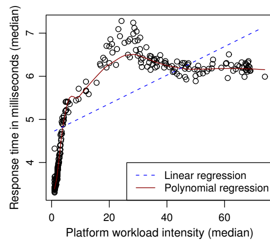
Figure A.1.: Example data (1/5): Relation between platform workload intensity pwi_1 and response time statistics.



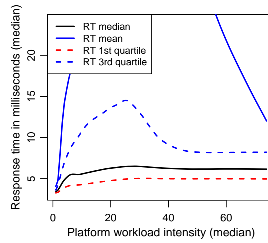
(1) AccountSqlMapDao.getAccount(...)



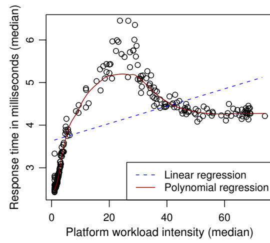
(2) AccountSqlMapDao.getAccount(...)



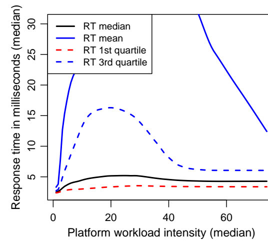
(3) ItemSqlMapDao.getItem(...)



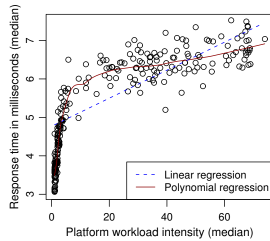
(4) ItemSqlMapDao.getItem(...)



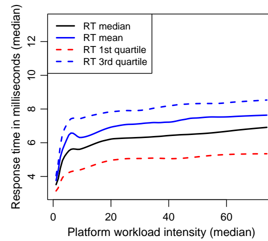
(5) ItemSql...getItemListByProduct(...)



(6) ItemSql...getItemListByProduct(...)

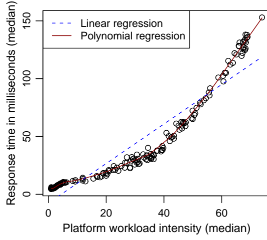


(7) OrderSqlMapDao.insertOrder(...)

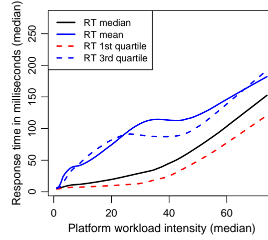


(8) OrderSqlMapDao.insertOrder(...)

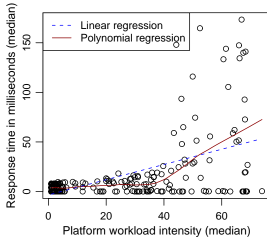
Figure A.2.: Example data (2/5): Relation between platform workload intensity pwi_1 and response time statistics.



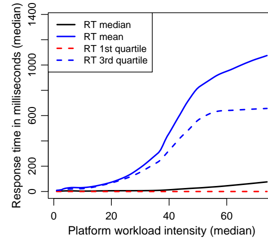
(1) CatalogBean.viewProduct(...)



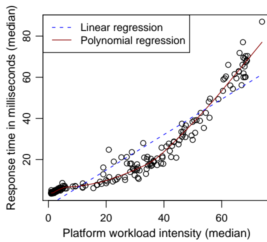
(2) CatalogBean.viewProduct(...)



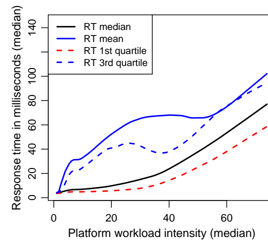
(3) OrderBean.newOrder(...)



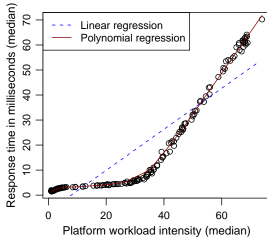
(4) OrderBean.newOrder(...)



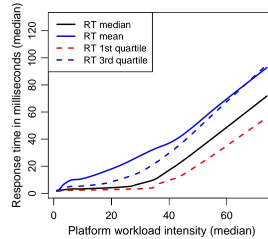
(5) AccountService.getAccount(...)



(6) AccountService.getAccount(...)

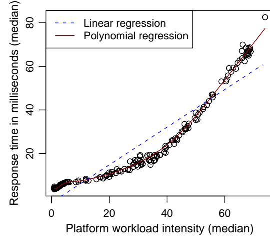


(7) CatalogService.getCategory(...)

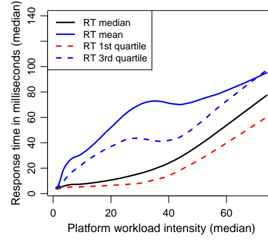


(8) CatalogService.getCategory(...)

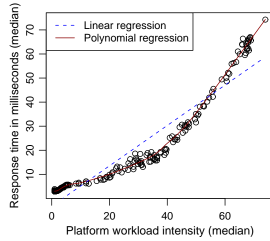
Figure A.3.: Example data (3/5): Relation between platform workload intensity pwi_1 and response time statistics.



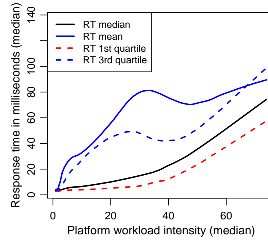
(1) CatalogService.getItem(...)



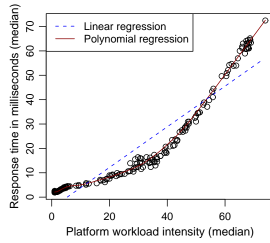
(2) CatalogService.getItem(...)



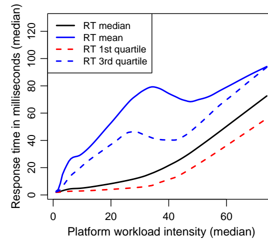
(3) CatalogSer...getItemListByProduct(...)



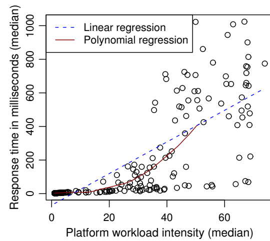
(4) CatalogSer...getItemListByProduct(...)



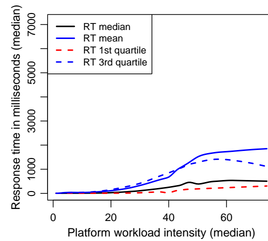
(5) CatalogSer...getProductListByCategory



(6) CatalogSer...getProductListByCategory

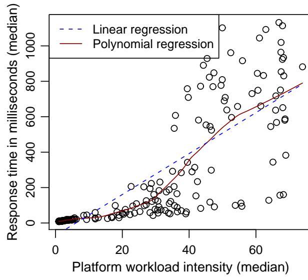


(7) OrderService.getNextId(...)

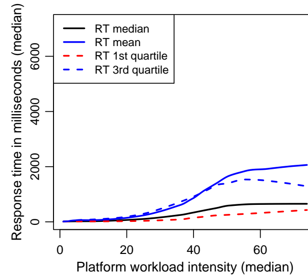


(8) OrderService.getNextId(...)

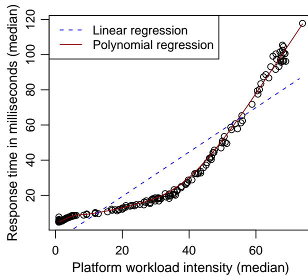
Figure A.4.: Example data (4/5): Relation between platform workload intensity pwi_1 and response time statistics.



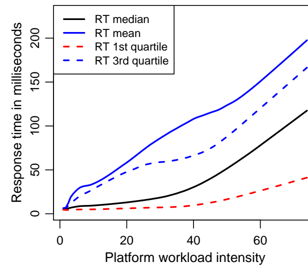
(1) OrderService.insertOrder(...)



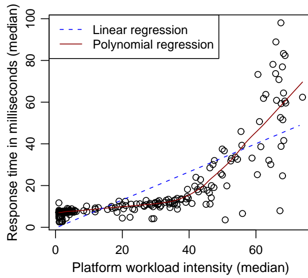
(2) OrderService.insertOrder(...)



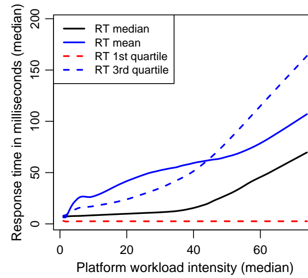
(3) ActionServlet.doGet(...)



(4) ActionServlet.doGet(...)



(5) ActionServlet.doPost(...)



(6) ActionServlet.doPost(...)

Figure A.5.: Example data (5/5): Relation between platform workload intensity pwi_1 and response time statistics.

B. Standard Deviation Reduction

The metric *standard deviation reduction* (for a single software operation) from Page 78 is formally defined as

$$\frac{sd(\mathrm{OR}) - \sum_{i=1}^{n} \left(\frac{|\mathrm{P}_i|}{|\mathrm{OR}|} \; sd(\mathrm{P}_i) \right)}{sd(\mathrm{OR})} \tag{B.1}$$

with:

- $sd(\mathrm{X})$ as the function that computes the (unbiased sample) standard deviation for a multiset X of observations, (a multiset can contain identical elements multiple times, in contrast to a set.)
- OR as the original multiset of observations (i.e., response times of a single operation) before applying WiSTA or TracSTA,
- P_i as the multiset of observations of the ith partition of the n partitions that are created by WiSTA or TracSTA, and
- |X| as the number of elements in a multiset X (i.e., the number of response times).

The resulting value is expressed as percent value; for instance, a value of 0.24 would be expressed as standard deviation reduction of 24 %.

The metric above is the standard deviation reduction for a single operation. For providing a single number for a system with many operations, the weighted average standard deviation reduction is computed, where the weights are given by the operation call frequencies (see also Section 4.3, Page 78).

C. Listing Example Chapter 5

Listing C.1 shows the source code for the two-method example in Chapter 5 on Page 102.

Listing C.1: Java source code for the pwi_4 example.

```
import kieker.tpmon.annotations.TpmonMonitoringProbe;
public class Starter extends Thread{
  static int numberOfRequests = 1500;
  public static void main(String[] args) throws
      InterruptedException {
    for (int i = 0; i < numberOfRequests; i++) {
      new Starter().start();
      Thread.sleep(25);
    }
    System.exit(0);
  }
  public void run() throws InterruptedException {
    wait();
    work();
  }
  @TpmonMonitoringProbe()
  public void wait() throws InterruptedException {
    Thread.sleep(500);
  }
  static boolean boolvar = true;
  @TpmonMonitoringProbe()
  private void work() {
    int a = (int)(Math.random() * 5d);
    for (int i=0; i<2500000; i++) {
      a += i/1000;
    }
    if (a % 10000 == 0 ) {
      boolvar = false;
    }
  }
}
```

D. Call Graph Profiling Tools

Many profiling tools allow performance analysis that is related to the trace context analysis concept in Chapter 4. In the following, the call graph profiling and timing behavior analysis of gprof [Graham et al., 1982], Google's perftools, Valgrind, Java's HPROF, and the NetBeans profiler are presented. All these tools are freely available. The tools are all demonstrated in the context of the running example shown in Listing D.1. First, the example is implemented in C; later in this appendix, it is translated to Java.

A particular focus of this demonstration is to discuss timing behavior that is specific to the caller (see Chapter 4). The source code shows that two different callers call the operation "catalog". Operation "caller2" causes ten times more loops in "catalog" than "caller1". Therefore, the execution of "catalog" should longer if it is called from "caller2" than from "caller1".

Listing D.1: Running example C program "profilertest.c".

```
 1 main() {
 2   int i;
 3   for (i = 0; i < 10; i++) {
 4     caller1(); caller2();
 5   }
 6 }
 7 caller1() { catalog( 1000*1000); } // some iterations
 8 caller2() { catalog(10*1000*1000); } // more iterations
 9
10 catalog(iterations)
11 int iterations; {
12   int j,c;
13   for (j=0; j< iterations; j++){
14     c = c + 1.5 * 0.98; // just some calculation
15   }
16 }
```

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      1.49     1.49       20    74.50    74.50  catalog
  0.00      1.49     0.00       10     0.00    74.50  caller1
  0.00      1.49     0.00       10     0.00    74.50  caller2

Call graph:
index % time    self  children    called     name
                0.74    0.00      10/20          caller1 [3]
                0.74    0.00      10/20          caller2 [4]
[1]    100.0    1.49    0.00      20         catalog [1]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    1.49                 main [2]
                0.00    0.74      10/10          caller2 [4]
                0.00    0.74      10/10          caller1 [3]
-----------------------------------------------
                0.00    0.74      10/10          main [2]
[3]     50.0    0.00    0.74      10         caller1 [3]
                0.74    0.00      10/20          catalog [1]
-----------------------------------------------
                0.00    0.74      10/10          main [2]
[4]     50.0    0.00    0.74      10         caller2 [4]
                0.74    0.00      10/20          catalog [1]
-----------------------------------------------

Index by function name
   [3] caller1       [4] caller2     [1] catalog
```

Figure D.1.: Gprof output for Listing D.1 and the shell commands of Listing D.2.

D.1. Gprof

Since the 1980s, gprof [Graham et al., 1982] is a widely used tool for call graph profiling. It is still part of many Linux and Unix distributions. The shell commands in Listing D.2 compile the C example (Listing D.1) with debugging support, execute the program, and format the gprof output into a file named “formated-output.txt”.

Listing D.2: Shell commands for compiling and applying gprof to code of Listing D.1.

```
1 cp profilertest.c gproftest.c
2 gcc -pg -g -o gproftest gproftest.c
3 ./gproftest
4 gprof -b gproftest gmon.out > formated-output.txt
```

Figure D.1 shows the resulting formatted gprof output. The “Flat profile” area shows the number of calls, the approximated execution time, and response time for each operation. The execution times are approximations

as calculated by using sampling, i.e., the program counter is checked frequently during execution. Therefore, the execution time computation is a statistical approximation [Graham et al., 1982]. The row in the upper blue box explains that the operation "caller1" has no own execution time (self ms/call 0.00 seconds).

The "Call graph"-area in Figure D.1 is much related to our work in Chapter 4; it contains for each operation (catalog, main, caller1, caller2) a separate listing. Each listing shows callers and callees above, and respectively below the line starting with the index of the current operation. For instance, the part in the lower blue box correctly shows that both operations "caller1" and "caller2" call the operation "catalog".

The lower blue box of Figure D.1 shows an important performance analysis limitation of gprof: It shows identical total execution times of 0.74 seconds for catalog for both calls from caller1 and caller2. Gprof strongly simplifies by equally dividing the total time of catalog (1.49 seconds) to all callers based on their relative frequency (here both 10/20). Gprof assumes the same time consumption for subcalls [Hall, 1992].

D.2. Google's Perftools CPU Profiler

Listing D.3 shows shell commands to apply Google's perftools CPU profiler to the same source code of Listing D.1. The gcc compiler is used with the flag "-lprofile", which requires Google's perftools to be installed. Command 3 of Listing D.3 executes the program and creates a raw profile file named hello-googleperftools.profile. Finally, Google's Perl script pprof[1] is used to create the formatted output into the file profiler-output.txt, shown in Figure D.2.

Listing D.3: Shell commands for applying Google's perftools and pprof to Listing D.1.

```
1 cp profilertest.c hello-googleperftools.c
2 gcc -o hello-googleperftools hello-googleperftools.c -
     lprofiler
3 CPUPROFILE=hello-googleperftools.profile ./hello-
     googleperftools
4 ./pprof --text hello-googleperftools hello-googleperftools.
     profile > profiler-output.txt
```

[1] The pprof script is available here: http://code.google.com/p/google-perftools/source/browse/trunk/src/pprof

```
Total: 148 samples
     148 100.0% 100.0%      148 100.0% catalog
       0   0.0% 100.0%      148 100.0% __libc_start_main
       0   0.0% 100.0%      148 100.0%  start
       0   0.0% 100.0%       14   9.5% caller1
       0   0.0% 100.0%      134  90.5% caller2
       0   0.0% 100.0%      148 100.0% main
```

Figure D.2.: Perftools's output for Listing D.1 with commands from Listing D.3.

D.3. Valgrind's Call Graph Generator Callgrind

Listing D.4 shows how to apply the Valgrind[2] [Nethercote and Seward, 2007] call graph profiler. Figure D.3 displays KCachegrind[3]'s visualization.

Listing D.4: Commands for using Valgrind and Kcachegrind.

```
cp profilertest.c valgrindtest.c
gcc -pg -g -o valgrindtest valgrindtest.c
valgrind --tool=callgrind ./valgrindtest
kcachgrind callgrind.out.*
```

D.4. Java's HPROF Profiler

The Java SDK contains the profiler HPROF since Version 5. Listing D.5 shows a Java program for the example of Listing D.1. The number of loops was increased to have execution times that are similar to those of the C program. This seems to be required to compensate compiler optimizations. In the example with C, compiler optimization flags, such as -O2 were avoided, because the gcc compiler would automatically remove the subcalls to catalog.

Hprof is applied to the program of Listing D.5 by using the commands of Listing D.6. Figure D.4 on Page 176 shows the corresponding output.

Listing D.5: Java variant of the running example from Listing D.1.

```
public class profilertest {
  public static void main (String[] args) throws Exception
       {
```

[2]Valgrind http://valgrind.org
[3]KCachegrind http://kcachegrind.sourceforge.net/html/Home.html

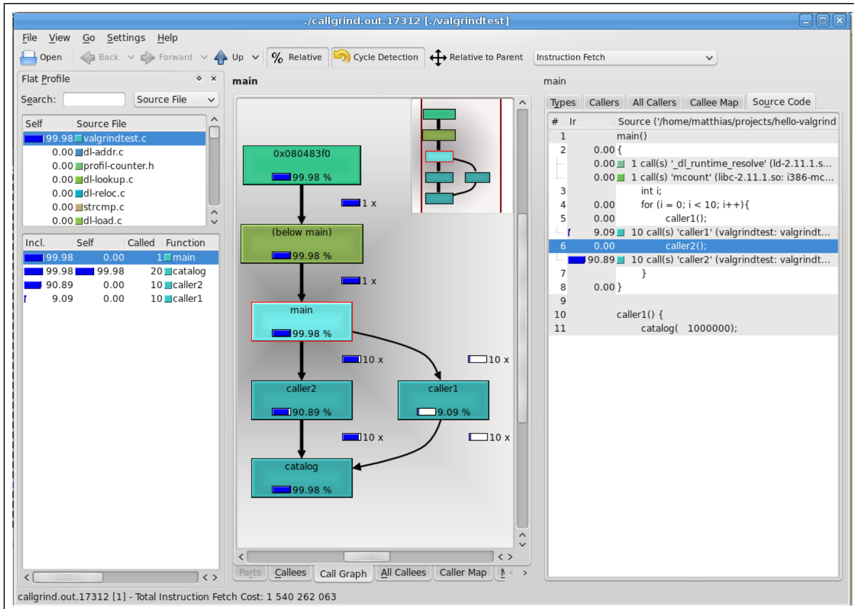


Figure D.3.: KCachegrind's call graph view for the Valgrind profile.

```
3        for (int i = 0 ; i < 10 ; i ++ ) {
4          caller1 ();
5          caller2 ();
6        }
7    }
8
9    public static void caller1 () { catalog ( 1000000 *10 );
          }
10   public static void caller2 () { catalog (10 * 1000000
         *10 ); }
11
12   public static void catalog (long iterations) {
13       double c = 0.0d;
14       for (int j = 0 ; j < iterations ; j ++ ) {
15         c = c + 1.5 * 0.98 ;
16       }
17   }
18 }
```

```
[...]

TRACE 300955:
 profilertest.catalog(profilertest.java:Unknown line)
 profilertest.caller2(profilertest.java:Unknown line)
 profilertest.main(profilertest.java:Unknown line)
TRACE 300953:
 profilertest.catalog(profilertest.java:Unknown line)
 profilertest.caller1(profilertest.java:Unknown line)
 profilertest.main(profilertest.java:Unknown line)

[...]

CPU TIME (ms) BEGIN (total = 1728) Thu Dec 29 10:25:06 2011
rank   self  accum   count trace method
   1 89.06% 89.06%      10 300955 profilertest.catalog
   2  8.85% 97.92%      10 300953 profilertest.catalog

[...]

  35  0.06% 100.00%      10 300954 profilertest.caller1
```

Figure D.4.: Part of the output created by the Java profiler hprof (file java.hprof.txt).

Listing D.6: Commands for applying the Java profiler to Listing D.5.

```
javac -g profilertest.java
java -Xrunhprof:cpu=times profilertest
```

D.5. NetBeans 6.9 Java Profiler

A profiler is part of the NetBeans IDE[4] (Version 6.9). Figure D.5 shows the Back Trace feature of this profiler for the Java program of Listing D.5. The Back Trace shows a caller-context-sensitive performance analysis – it shows that response times of catalog are larger from "caller2" (1571 ms) than from "caller1" (172 ms).

[4]NetBeans Profiler `http://profiler.netbeans.org/`

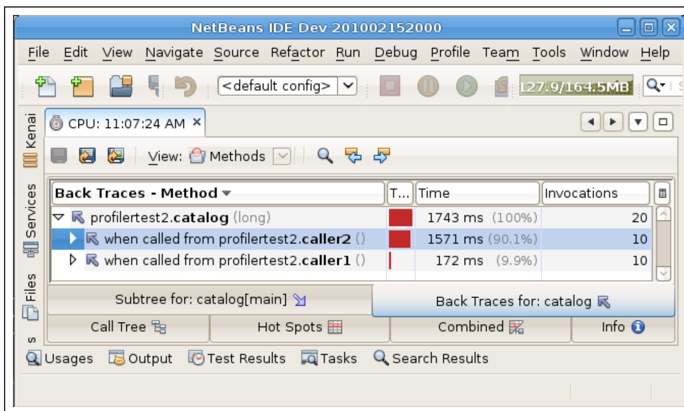
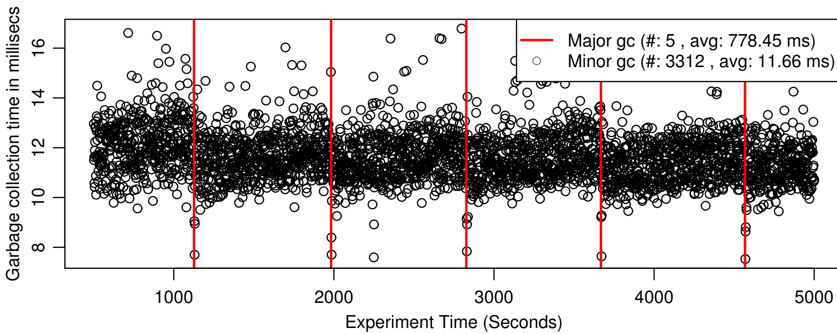


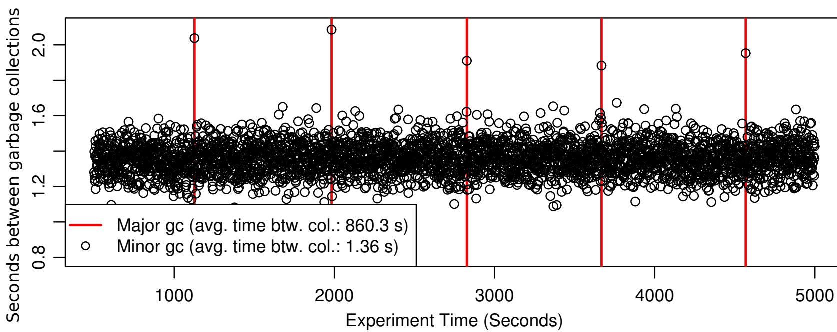
Figure D.5.: NetBeans 6.9 Profiler's "Back Trace" function provides caller context analysis.

E. Garbage Collection Analysis

Garbage collection is a known performance influence to software applications (see Foundations 2.1.4.6 on Page 22). In the context of the case studies of the Chapters 4 and 5, some garbage collection measurements were taken to estimate its relevance to the timing behavior analyses presented in this thesis.



(1) Major garbage collection runs and execution times of minor runs.



(2) Time between subsequent minor garbage collections.

Figure E.1.: Example: Garbage collection activity.

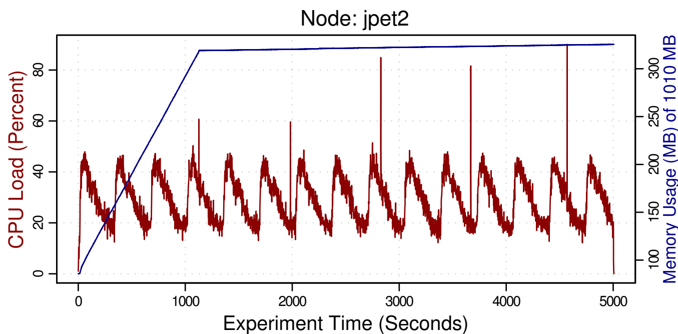


Figure E.2.: Example: CPU utilization and total system memory allocation during the experiment of Figure E.1(1).

Figure E.1 displays garbage collection monitoring data from one experiment run of the iBATIS Java JPetStore application. A probabilistic workload with a constant workload-intensity of 30 concurrent users was used. Figure E.1(1) shows that there were a large amount (3312) of the so-called minor garbage collections and five so-called major garbage collections during the 5000 seconds experiment. The minor collections required in average 11.7 milliseconds and the major collections in average 778.5 milliseconds. Therefore, the pauses caused by garbage collection can lead to significantly increased operation response times. Figure E.2 shows CPU utilization and memory consumption during the same experiment. The five vertical CPU load peaks (at about sec. 1120, 1980, 2830, 3670, and 4570) indicate a connection to the major garbage collections.

The monitoring data shown in Figure E.3 is from an experiment with linearly increasing workload (up to approximately 50 users). The measurements indicate the same connection between major collections and CPU load peaks. The time between minor collections appears to decrease by increasing workload intensity. The origin of the regular delays of minor collections (the dots in the upper chart at approximately 3 seconds between collections) seems to correlate to the down-peaks in CPU load.

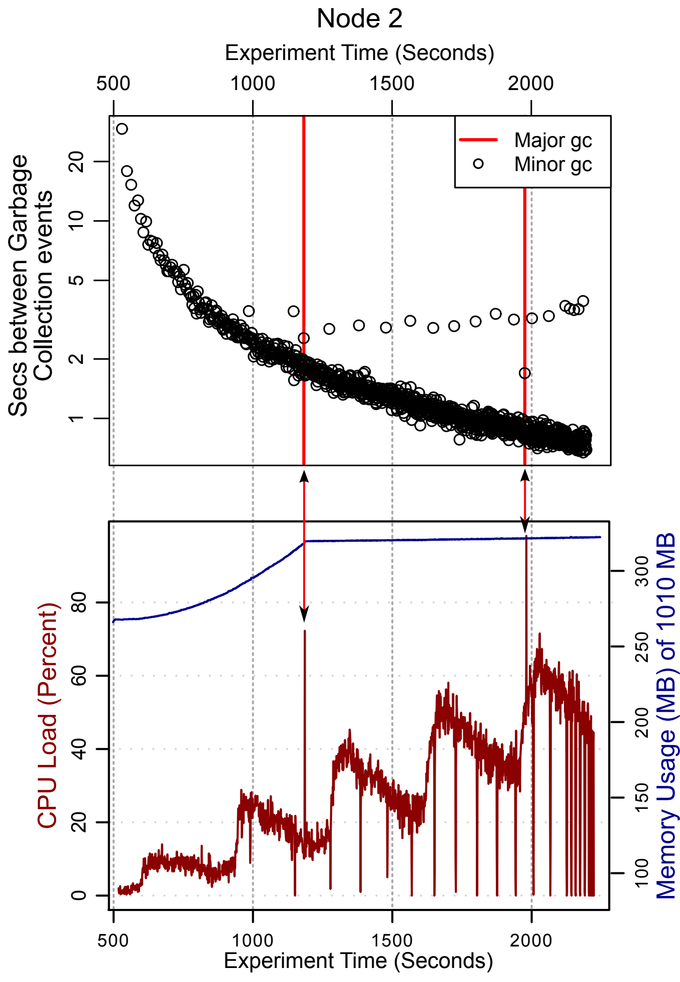


Figure E.3.: Example for periodic CPU utilization peaks and its relations to major garbage collection events.

List of Figures

1.1. Response times of ten internal software operations of an online store. 2
1.2. Overview of the fault localization approach and the two central contributions of this thesis. 4

2.1. Efficiency in the ISO/IEC 9126-1 standard 9
2.2. Response time metrics. 11
2.3. Response times and execution times. 11
2.4. Fitting distributions to response time data. 17
2.5. Multimodality of response times. 19
2.6. Typical relation between response times and workload intensity. 21
2.7. Measured and predicted average response times in relation to workload intensity. 22
2.8. Error propagation example in a component-based software system. 23
2.9. Fundamental chain of dependability threats [Avižienis et al., 2004]. 24
2.10. Failure class hierarchy. 25
2.11. Failure modes and failure domains. 25
2.12. Fault localization in software systems: concepts and techniques. 29
2.13. Anomaly detection example with three classes of normal behavior. 35
2.14. Software system layers and monitoring targets. Image based on Focke [2006]. 36

3.1. Conceptual steps of the fault localization approach. 40
3.2. Anomaly analysis in component-based systems. 41
3.3. Error propagation, errors cause anomalies, and anomaly propagation. 42
3.4. Schema for a monitored software operation execution. . . . 44

3.5. Bookstore example. “M” boxes indicate monitoring probes. 46
3.6. Dynamic Call Tree for monitoring data of Table 3.1. 48
3.7. Example: Different trace shapes can correspond to different timing behavior. 50
3.8. Workload curve (active user sessions) of an online photo printing service. 51
3.9. Unique visitors of two online photo printing services. Graph by compete.com. 51
3.10. Response time distributions specific to workload intensity. 52
3.11. Response time distribution characteristics for two different numbers of bins. 54
3.12. Demonstration of different anomaly detection functions. . 57
3.14. Failure diagnosis visualization example. 61

4.1. Trace shape specific timing behavior: newOrder() in iBATIS JPetStore. 66
4.2. UML sequence diagrams for the monitoring data of Table 4.1. 69
4.3. Dynamic call trees for the sequence diagrams of Figure 4.2. 70
4.4. All trace shape contexts for operation $f()$. 72
4.5. All monitored response times of operation $f()$. 73
4.6. Stack context analysis identifies two stack contexts for operation $f()$. 74
4.7. Trace context analysis identified three contexts for operation $f()$. 74
4.8. The trace shape context tree organizes the response times by trace shape. 76
4.9. Illustration of tree optimization operators. 77
4.10. CS-4.1: Standard deviation reduction for different numbers of monitoring points. 82
4.11. CS-4.1: Trace contexts per operation for full instrumenation. 84
4.12. CS-4.1: Number of monitoring points in relation to the number of contexts. 85

5.1. Relation between response time statistics and workload intensity. 92
5.2. Anomaly detection with constant workload intensity. . . . 95
5.3. Anomaly detection with changing workload intensity. . . . 95
5.4. Probability density distributions for low, medium, and high workload intensity. 96
5.5. Example traces as UML sequence diagrams. 98

5.6. *pwi* example 1: Multiple traces within the same execution environment. 99
5.7. *pwi* example 2: Trace 3 involves multiple execution environments. 101
5.8. Example pwi_4: Timing behavior characteristics correlate to pwi_4. 103
5.9. CS-5.1: Deployment architecture of the distributed JPetStore.105
5.10. CS-5.1: Workload intensity specification based on 24 hour measurements of a real customer portal. 106
5.11. CS-5.1: Standard deviation reduction. 106
5.12. CS-5.1: Log-transformation increases standard deviation reduction. 107
5.13. CS-5.2: Workload intensity specification and CPU usage. . 108
5.14. CS-5.2: Standard deviation reduction. 109
5.15. CS-5.3: Workload curve (active sessions). 110
5.16. CS-5.3: Standard deviation reduction. 110
5.17. Size of bins (i.e. classes) without log-transformation (operation $work()$). 112
5.18. Relation between *pwi* and response times (operation *viewProduct* of CS-5.1). 114

6.1. Tracing a path through a distributed multi-tier system (Source: [Aguilera et al., 2003]). 118
6.2. Magpie's visualization of a single request's trace (Source: [Barham et al., 2004]). 119
6.3. Response times during four days of seasonal data and a seasonal forecasting (green line) for a window size of 24 hours. Image by Bielefeld [2012]. 136

A.1. Example data (1/5): Relation between platform workload intensity pwi_1 and response time statistics. 161
A.2. Example data (2/5): Relation between platform workload intensity pwi_1 and response time statistics. 162
A.3. Example data (3/5): Relation between platform workload intensity pwi_1 and response time statistics. 163
A.4. Example data (4/5): Relation between platform workload intensity pwi_1 and response time statistics. 164
A.5. Example data (5/5): Relation between platform workload intensity pwi_1 and response time statistics. 165

D.1. Gprof output for Listing D.1 and the shell commands of Listing D.2. 172
D.2. Perftools's output for Listing D.1 with commands from Listing D.3. 174
D.3. KCachegrind's call graph view for the Valgrind profile. . . 175
D.4. Part of the output created by the Java profiler hprof (file java.hprof.txt). 176
D.5. NetBeans 6.9 Profiler's "Back Trace" function provides caller context analysis. 177

E.1. Garbage collection activity. 179
E.2. CPU utilization and total system memory allocation. . . . 180
E.3. CPU utilization peaks compared with garbage collection activity. 181

List of Tables

2.1. Categorization schemes related to fault localization. 30

3.1. Example monitoring data for a request to the bookstore. . 46
3.2. Example for training data sets. 53
3.3. Example for anomaly score computation. 58

4.1. Simplified monitoring data for the ongoing TracSTA example. 68
4.2. Trace shape contexts for the ongoing example. 71
4.3. Summary of the settings of the three TracSTA case studies. 79
4.4. CS-4.1: Summary of the instrumentation scenarios and monitoring data. 81
4.5. CS-4.1: Standard deviation reduction results for E1 and E2. 81
4.6. CS-4.1: Distinct trace shape contexts per instrumentation and context type. 83
4.7. CS-4.2: Standard deviation reduction and number of contexts. 86
4.8. CS-4.3: Standard deviation reduction and number of contexts. 87
4.9. CS-4.3: Standard deviation reduction and number of contexts with model size optimization. 88

5.1. *pwi* metrics overview. 97
5.2. Example: Two weight vectors (columns). 102
5.3. Summary of the settings of the WiSTA case studies. 104
5.4. CS-5.1: Average standard deviation reduction (in %). . . . 107
5.5. CS-5.2: Average standard deviation reduction (in %). . . . 108
5.6. CS-5.3: Average standard deviation reduction (in %). . . . 110

Bibliography

[Abreu et al. 2007] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. On the accuracy of spectrum-based fault localization. In *Academic and Industrial Conference Practice and Research Techniques*, pages 89–98, Sept. 2007. doi:10.1109/TAIC.PART.2007.13. (Cited on page 28.)

[Achtert et al. 2010] E. Achtert, H.-P. Kriegel, L. Reichert, E. Schubert, R. Wojdanowski, and A. Zimek. Visual evaluation of outlier detection models. In *Proceedings of the 15th International Conference Database Systems for Advanced Applications (DASFAA'10)*, volume 5982 of *Lecture Notes in Computer Science*, pages 396–399. Springer, Apr. 2010. ISBN 978-3-642-12097-8. doi:10.1007/978-3-642-12098-5_34. (Cited on page 55.)

[Adams 1984] E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, Jan. 1984. doi:10.1147/rd.281.0002. (Cited on page 27.)

[Agarwal et al. 2004] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs and run-time behavior models. In *Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'04)*, volume 3278 of *Lecture Notes in Computer Science*, pages 171–182. Springer, Nov. 2004. ISBN 3-540-23631-7. doi:10.1007/b102082. (Cited on pages 1, 33, 59, and 135.)

[Aguilera et al. 2003] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 74–89. ACM, 2003. ISBN 1-58113-757-5. doi:10.1145/945445.945454. (Cited on pages 1, 44, 117, 118, 122, 148, and 185.)

[Amin et al. 2012] A. Amin, A. Colman, and L. Grunske. Statistical detection of QoS violations based on CUSUM control charts. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance*

Engineering (ICPE'12), pages 97–108. ACM, 2012. ISBN 978-1-4503-1202-8. doi:10.1145/2188286.2188302. (Cited on page 29.)

[Ammons et al. 1997] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, pages 85–96. ACM, 1997. ISBN 0-89791-907-6. doi:10.1145/258915.258924. (Cited on pages 5, 44, 48, 63, 67, 69, 116, 117, 121, and 146.)

[Arlitt et al. 2001] M. F. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology*, 1(1):44–69, Aug. 2001. ISSN 1533-5399. doi:10.1145/383034.383036. (Cited on pages 1, 67, and 94.)

[Armbrust et al. 2010] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53 (4):50–58, Apr. 2010. doi:10.1145/1721654.1721672. (Cited on page 94.)

[Au-Yeung et al. 2004] S. Au-Yeung, N. Dingle, and W. Knottenbelt. Efficient approximation of response time densities and quantiles in stochastic models. In *Proceedings of the 4th International Workshop on Software and Performance (WOSP'04)*, pages 151–155. ACM, 2004. ISBN 1-58113-563-7. doi:10.1145/974044.974068. (Cited on page 16.)

[Avižienis et al. 2004] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Transactions on Dependable and Secure Computing*, 1(1): 11–33, Jan. 2004. ISSN 1545-5971. doi:10.1109/TDSC.2004.2. (Cited on pages 23, 24, 25, 27, 29, and 183.)

[Avritzer et al. 2005] A. Avritzer, A. Bondi, and E. J. Weyuker. Ensuring stable performance for systems that degrade. In *Proceedings of the 5th International Workshop on Software and Performance (WOSP'05)*, pages 43–51. ACM, 2005. ISBN 1-59593-087-6. doi:10.1145/1071021.1071026. (Cited on page 137.)

[Avritzer et al. 2006] A. Avritzer, A. B. Bondi, M. Grottke, K. S. Trivedi, and E. J. Weyuker. Performance assurance via software rejuvenation: Monitoring, statistics and algorithms. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 435–444.

IEEE, June 2006. ISBN 0-7695-2607-1. doi:10.1109/DSN.2006.58. (Cited on pages 58 and 137.)

[Avritzer et al. 2007] A. Avritzer, A. Bondi, and E. J. Weyuker. Ensuring system performance for cluster and single server systems. *Journal of Systems and Software*, 80(4):441–454, Apr. 2007. ISSN 0164-1212. doi:10.1016/j.jss.2006.07.020. (Cited on page 137.)

[Axelsson 2000] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers University, Mar. 2000. (Cited on pages 30 and 31.)

[Bailey and Soucy 1983] R. M. Bailey and R. C. Soucy. Performance and availability measurement of the IBM information network. *IBM Systems Journal*, 22(4):404–416, 1983. doi:10.1147/sj.224.0404. (Cited on pages 128 and 133.)

[Ball and Larus 1996] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th ACM/IEEE International Symposium on Microarchitecture (MICRO'29)*, pages 46–57. IEEE, Dec. 1996. ISBN 0-8186-7641-8. doi:10.1109/MICRO.1996.566449. (Cited on pages 44, 63, and 116.)

[Balsamo et al. 2004] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *Transactions on Software Engineering*, 30(5):295–310, May 2004. ISSN 0098-5589. doi:10.1109/TSE.2004.9. (Cited on pages 14 and 15.)

[Barham et al. 2003] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HOTOS'03)*. USENIX Association, 2003. (Cited on pages 117 and 118.)

[Barham et al. 2004] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium On Operating Systems Design and Implementation (OSDI'04)*, pages 259–272. USENIX Association, 2004. (Cited on pages 117, 119, and 185.)

[Bause et al. 2008] F. Bause, P. Buchholz, J. Kriege, and S. Vastag. A framework for simulation models of service-oriented architectures. In S. Kounev, I. Gorton, and K. Sachs, editors, *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW'08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 208–227, Heidelberg,

June 2008. Springer. ISBN 978-3-540-69813-5. doi:10.1007/978-3-540-69814-2_14. (Cited on page 14.)

[Becker et al. 2007] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th International Workshop on Software and Performance (WOSP'07)*, pages 54–65. ACM, 2007. ISBN 1-59593-297-6. doi:10.1145/1216993.1217006. (Cited on pages 14 and 129.)

[Bemmerl and Bode 1991] T. Bemmerl and A. Bode. An integrated environment for programming distributed memory multiprocessors. In A. Bode, editor, *Proceedings of the 2nd European Conference on Distributed Memory Computing*, volume 487 of *Lecture Notes in Computer Science*, pages 130–142. Springer, 1991. doi:10.1007/BFb0032930. (Cited on page 37.)

[Bielefeld 2012] T. C. Bielefeld. Online performance anomaly detection for large-scale software systems. Master's thesis, University of Kiel, Mar. 2012. Diploma Thesis. (Cited on pages 58, 135, 136, 137, 152, 157, and 185.)

[Blackburn et al. 2004] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'04 / Performance'04)*, pages 25–36. ACM, 2004. ISBN 1-58113-873-3. doi:10.1145/1005686.1005693. (Cited on page 22.)

[Blackburn et al. 2006] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*. ACM, Oct. 2006. doi:10.1145/1167473.1167488. (Cited on pages 18 and 157.)

[Bocaniala and Palade 2006] C. D. Bocaniala and V. Palade. Computational intelligence methodologies in fault diagnosis: Review and state of the art. In V. Palade, C. D. Bocaniala, and L. C. Jain, editors, *Computational Intelligence in Fault Diagnosis*, Advanced Information and

Knowledge Processing, chapter 1, pages 1–36. Springer, 2006. ISBN 978-1-184628-343-7. (Cited on page 27.)

[Bond and McKinley 2007] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA'07)*, pages 97–112. ACM, 2007. ISBN 978-1-59593-786-5. doi:10.1145/1297027.1297035. (Cited on page 122.)

[Bourne 2004] S. Bourne. A conversation with bruce lindsay. *Queue*, 2 (8):22–33, Nov. 2004. ISSN 1542-7730. doi:10.1145/1036474.1036486. (Cited on pages 27 and 28.)

[Box and Jenkins 1990] G. E. P. Box and G. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990. ISBN 0816211043. (Cited on page 135.)

[Breunig et al. 2000] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 93–104. ACM, 2000. ISBN 1-58113-217-4. doi:10.1145/342009.335388. (Cited on pages 35 and 55.)

[Brutlag 2000] J. D. Brutlag. Aberrant behavior detection in time series for network monitoring. In *Proceedings of the 14th USENIX Conference on System Administration*, pages 139–146. USENIX Association, 2000. (Cited on page 142.)

[Bulej et al. 2005] L. Bulej, T. Kalibera, and P. Tůma. Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 60(1-4):345–358, 2005. ISSN 0166-5316. doi:10.1016/j.peva.2004.10.013. (Cited on pages 15, 18, and 129.)

[Candea et al. 2004] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - A technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*. USENIX Association, 2004. (Cited on pages 27 and 147.)

[Cappelli and Kowall 2011] W. Cappelli and J. Kowall. APM innovators: Driving APM technology and delivery evolution, Sept. 2011. URL `http://www.gartner.com/technology/reprints.`

do?id=1-17DC04V&ct=110920&st=sg. Available online, Last access: 2011-12-15. (Cited on page 123.)

[Chanda et al. 2007] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: transactional profiling for multi-tier applications. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*, pages 17–30. ACM, 2007. ISBN 978-1-59593-636-3. doi:10.1145/1272996.1273001. (Cited on pages 44, 117, 118, and 150.)

[Chandola et al. 2009] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41:1–58, July 2009. ISSN 0360-0300. doi:10.1145/1541880.1541882. (Cited on pages 32, 33, 34, and 35.)

[Chen et al. 2007] H. Chen, G. Jiang, C. Ungureanu, and K. Yoshihira. Online tracking of component interactions for failure detection and localization in distributed systems. *Transactions on Systems, Man, and Cybernetics*, 37(4):644–651, July 2007. ISSN 1094-6977. doi:10.1109/TSMCC.2007.897496. (Cited on page 139.)

[Chen et al. 2002] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 595–604. IEEE, 2002. doi:10.1109/DSN.2002.1029005. (Cited on pages 1, 28, 38, 122, 138, and 139.)

[Chen et al. 2004] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, pages 36–43. IEEE, May 2004. doi:10.1109/ICAC.2004.1301345. (Cited on page 144.)

[Cheng 2008] X. Cheng. Performance, benchmarking and sizing in developing highly scalable enterprise software. In S. Kounev, I. Gorton, and K. Sachs, editors, *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW'08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 174–190. Springer Verlag, June 2008. ISBN 978-3-540-69813-5. doi:10.1007/978-3-540-69814-2_12. (Cited on pages 12, 13, 14, 21, and 22.)

[Cheung et al. 2011] L. Cheung, L. Golubchik, and F. Sha. A study of web services performance prediction: A client's perspective. In *Proceedings*

of the 19th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS'11), pages 75–84, 2011. doi:10.1109/MASCOTS.2011.66. (Cited on page 125.)

[Choi et al. 1999] J. Choi, M. Choi, and S.-H. Lee. An alarm correlation and fault identification scheme based on osi managed object classes. In *Proceedings of the International Conference on Communications (ICC'99)*, pages 1547–1551. IEEE, 1999. doi:10.1109/ICC.1999.765477. (Cited on page 59.)

[Cleve and Zeller 2005] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 342–351. ACM, May 2005. ISBN 1595939632. doi:10.1109/ICSE.2005.1553577. (Cited on page 28.)

[Cohen et al. 2004] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation (OSDI'04)*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association. (Cited on page 140.)

[Cohen et al. 2005] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 105–118. ACM, 2005. ISBN 1-59593-079-5. doi:10.1145/1095810.1095821. (Cited on page 140.)

[Cristian et al. 1995] F. Cristian, H. Aghili, H. R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, Apr. 1995. (Cited on page 25.)

[Crovella and Bestavros 1997] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Transactions Networking*, 5(6):835–846, 1997. ISSN 1063-6692. doi:10.1109/90.650143. (Cited on pages 16 and 17.)

[Crovella et al. 1998] M. E. Crovella, M. S. Taqqu, and A. Bestavros. Heavy-tailed probability distributions in the world wide web. In *A Practical Guide To Heavy Tails: Statistical Techniques and Applications*, pages 3–26. Birkhäuser, 1998. ISBN 0-8176-3951-9. (Cited on pages 16 and 17.)

[CWE/SANS 2009] *Top 25 Most Dangerous Programming Errors*. CWE/SANS, July 2009. URL http://cwe.mitre.org/top25/pdf/2009_cwe_sans_top_25.pdf. Available online, Last access: 2010-10-15. (Cited on page 27.)

[D'Alconzo et al. 2009] A. D'Alconzo, A. Coluccia, F. Ricciato, and P. Romirer-Maierhofer. A distribution-based approach to anomaly detection and application to 3G mobile traffic. In *Proceedings of the 28th Conference on Global Telecommunications (GLOBECOM'09)*, pages 2888–2895. IEEE, 2009. ISBN 978-1-4244-4147-1. doi:10.1109/GLOCOM.2009.5425651. (Cited on pages 16 and 126.)

[Dallmeier 2010] V. Dallmeier. *Mining and checking object behavior*. PhD thesis, Saarland University, 2010. URL http://scidok.sulb.uni-saarland.de/volltexte/2010/3434. Available online, Last access: 2014-05-03. (Cited on pages 26 and 37.)

[Dallmeier et al. 2005] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight bug localization with AMPLE. In *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging(AADEBUG'05)*, pages 99–104. ACM, 2005. ISBN 1-59593-050-7. doi:10.1145/1085130.1085143. (Cited on page 144.)

[DeMillo and Mathur 1995] R. A. DeMillo and A. P. Mathur. A grammar based fault classification scheme and its application to the classification of the errors of TEX. Technical report, Software Engineering Research Center and Department Of Computer Science Purdue Univerity, 1995. (Cited on page 26.)

[Diaconescu and Murphy 2005] A. Diaconescu and J. Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proceedings of the 20th IEEE/ACM International Conference on Automated software Engineering (ASE'05)*, pages 44–53. ACM, 2005. ISBN 1-59593-993-4. doi:10.1145/1101908.1101918. (Cited on pages 137 and 147.)

[Diaconescu et al. 2004] A. Diaconescu, A. Mos, and J. Murphy. Automatic performance management in component based software systems. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*, pages 214–221. IEEE, 2004. ISBN 0-7695-2114-2. doi:10.1109/ICAC.2004.15. (Cited on pages 1, 37, and 137.)

[Downey 2001] A. B. Downey. The structural cause of file size distributions. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'01)*, pages 328–329. ACM, 2001. ISBN 1-58113-334-0. doi:10.1145/378420.378824. (Cited on page 17.)

[Dustin 2002] E. Dustin. *Effective Software Testing: 50 Specific Ways to Improve Your Testing*. Addison-Wesley Longman, 2002. ISBN 978-0201794298. (Cited on page 36.)

[Ehlers 2012] J. Ehlers. *Self-Adaptive Performance Monitoring for Component-Based Software Systems*. Number 2012-1 in Kiel Computer Science Series. Kiel University, Department of Computer Science, Apr. 2012. ISBN 9783844814477. PhD thesis. (Cited on pages 121, 135, 138, and 154.)

[Ehlers and Hasselbring 2011] J. Ehlers and W. Hasselbring. A self-adaptive monitoring framework for component-based software systems. In *Proceedings of the 5th European Conference on Software Architecture (ECSA'11)*, pages 278–286. Springer, 2011. doi:10.1007/978-3-642-23798-0_30. (Cited on pages 121 and 137.)

[Ehlers et al. 2011] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th ACM International Conference on Autonomic computing (ICAC'11)*, pages 197–200. ACM, 2011. ISBN 978-1-4503-0607-2. doi:10.1145/1998582.1998628. (Cited on pages 135 and 157.)

[Eisenstadt 1997] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, Apr. 1997. ISSN 0001-0782. doi:10.1145/248448.248456. (Cited on pages 27 and 28.)

[Elbaum et al. 2007] S. Elbaum, S. Kanduri, and A. Andrews. Trace anomalies as precursors of field failures: an empirical study. *Empirical Softwoftware Engineering*, 12(5):447–469, 2007. ISSN 1382-3256. doi:10.1007/s10664-007-9042-8. (Cited on page 143.)

[Eskin 2000] E. Eskin. Anomaly detection over noisy data using learned probability distributions. In *Proceedings of the 7th International Conference on Machine Learning (ICML'00)*, pages 255–262. Morgan Kaufmann, 2000. ISBN 1-55860-707-2. (Cited on page 52.)

[Eusgeld et al. 2008] I. Eusgeld, F. Fraikin, M. Rohr, F. Salfner, and U. Wappler. Software Reliability. In I. Eusgeld, F. Freiling, and R. Reussner, editors, *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science (LNCS)*, pages 104–125. Springer, 2008. ISBN 978-3-540-68946-1. doi:10.1007/978-3-540-68947-8_10. (Cited on pages 7, 24, and 27.)

[Farshchi 2003] J. Farshchi. Statstical-based intrusion detection, Apr. 2003. URL `http://www.securityfocus.com/infocus/1686`. Available online, Last access: 2014-05-03. (Cited on page 31.)

[Fawcett 2006] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. ISSN 0167-8655. doi:10.1016/j.patrec.2005.10.010. (Cited on page 34.)

[Fenton and Ohlsson 2000] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000. ISSN 0098-5589. doi:10.1109/32.879815. (Cited on page 27.)

[Field et al. 2012] A. Field, J. Miles, and Z. Field. *Discovering Statistics Using R*. SAGE Publications, 2012. ISBN 978-1-4462-0045-2. (Cited on page 16.)

[Fittkau et al. 2013] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The explorviz approach. In *Working Conference on Software Visualization (VISSOFT'13)*, Sept. 2013. doi:10.1109/VISSOFT.2013.6650536. (Cited on page 158.)

[Fittkau et al. 2014] F. Fittkau, P. Stelzer, and W. Hasselbring. Live visualization of large software landscapes for ensuring architecture conformance. In *2nd International Workshop on Software Engineering for Systems-of-Systems 2014 (SESoS'14)*. ACM, Aug. 2014. (Cited on page 158.)

[Focke 2006] T. Focke. Performance Monitoring von Middleware-basierten Applikationen, Mar. 2006. Master's thesis (Diplomarbeit), Carl von Ossietzky University Oldenburg, Software Engineering Group, Department of Computing Science. (Cited on pages 7, 36, and 183.)

[Focke et al. 2007a] T. Focke, W. Hasselbring, M. Rohr, and J.-G. Schute. Ein Vorgehensmodell für Performance-Monitoring von Informationssystemlandschaften. *EMISA Forum*, 27(1):26–31, Jan. 2007a. ISSN 1610-3351. (Cited on pages 37, 46, and 47.)

[Focke et al. 2007b] T. Focke, W. Hasselbring, M. Rohr, and J.-G. Schute. Instrumentierung zum Monitoring mittels Aspekt-orientierter Programmierung. In W.-G. Bleek, H. Schwentner, and H. Züllighoven, editors, *Proceedings of the GI Conference on Software Engineering 2007 (SE'07)*, volume 106 of *GI-Edition – Lecture Notes in Informatics (LNI)*, pages 55–59. Gesellschaft für Informatik (GI), Bonner Köllen Verlag, Mar. 2007b. ISBN 978-3-88579-199-7. (Cited on pages 7 and 46.)

[Forrest et al. 1996] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the Symposium on Security and Privacy*, page 0120. IEEE, 1996. doi:10.1109/SECPRI.1996.502675. (Cited on pages 143 and 144.)

[Franks et al. 2009] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2009. ISSN 0098-5589. doi:10.1109/TSE.2008.74. (Cited on page 14.)

[Frotscher 2013] T. Frotscher. Architecture-based multivariate anomaly detection for software systems. Master's thesis, University of Kiel, 2013. URL `http://eprints.uni-kiel.de/21346/`. Masterarbeit. (Cited on pages 124, 136, 152, and 157.)

[Gao et al. 2007] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'07)*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. doi:10.1145/1362622.1362643. (Cited on page 132.)

[Garlan et al. 2003] D. Garlan, S.-W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 23–46. Springer Verlag, 2003. ISBN 3-540-40727-8. doi:10.1007/3-540-45177-3_3. (Cited on page 1.)

[Garlan et al. 2004] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with

reusable infrastructure. *Computer*, 37(10):46–54, Oct. 2004. ISSN 0018-9162. doi:10.1109/MC.2004.175. (Cited on page 94.)

[Gartner 2010] Gartner. Magic quadrant for application performance monitoring, Feb. 2010. Gartner. (Cited on page 123.)

[Georges et al. 2008] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems, Languagues, and Applications (OOPSLA'08)*, pages 367–384. ACM, 2008. doi:10.1145/1449764.1449794. (Cited on page 20.)

[Ghosh et al. 1998] A. K. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the 14th Computer Security Applications Conference (ACSAC'98)*. IEEE, Dec. 1998. doi:10.1109/CSAC.1998.738646. (Cited on page 31.)

[Giesecke 2008] S. Giesecke. *Architectural styles for early goal-driven middleware platform selection*. PhD thesis, Carl von Ossietzky University of Oldenburg, Department of Computing Science, 2008. (Cited on page 19.)

[Giesecke et al. 2006] S. Giesecke, M. Rohr, and W. Hasselbring. Software-Betriebs-Leitstände für Unternehmensanwendungslandschaften. In *Proceedings of the Workshop "Software-Leitstände: Integrierte Werkzeuge zur Softwarequalitätssicherung"*, volume P-94 of *Lecture Notes in Informatics*, pages 110–117. Gesellschaft für Informatik, Oct. 2006. ISBN 978-3-88579-188-1. (Cited on page 26.)

[Gorton and Liu 2003] I. Gorton and A. Liu. Evaluating the performance of ejb components. *IEEE Internet Computing*, 7(3):18–23, 2003. ISSN 1089-7801. doi:10.1109/MIC.2003.1200296. (Cited on page 19.)

[Govindraj et al. 2006] K. Govindraj, S. Narayanan, B. Thomas, P. Nair, and S. P. On using AOP for Application Performance Management. In M. Chapman, A. Vasseur, and G. Kniesel, editors, *Proceedings of the AOSD 2006 Industry Track (Technical Report IAI-TR-2006-3, University of Bonn)*, pages 18–30, Mar. 2006. (Cited on pages 36 and 150.)

[Graham et al. 1982] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Notes*, 17(6):120–126, 1982. doi:10.1145/872726.806987. (Cited on pages 1, 5, 15, 21, 63, 67, 116, 146, 171, 172, and 173.)

[Gray 1986] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems (SRDS-5)*, pages 3–12. IEEE, 1986. (Cited on pages 25, 26, and 27.)

[Grottke et al. 2010] M. Grottke, A. Nikora, and K. Trivedi. An empirical investigation of fault types in space mission system software. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'10)*, pages 447–456. IEEE, June 2010. doi:10.1109/DSN.2010.5544284. (Cited on page 27.)

[Grubbs 1956] F. E. Grubbs. Procedures for detecting outlying observations in samples. *Technometrics*, 11:1–21, 1956. (Cited on page 32.)

[Gruschke 1998a] B. Gruschke. Integrated event management: Event correlation using dependency graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'98)*, Oct. 1998a. (Cited on page 30.)

[Gruschke 1998b] B. Gruschke. A new approach for event correlation based on dependency graphs. In *Proceedings of the 5th Workshop of the OpenView University Association*, Apr. 1998b. (Cited on pages 40 and 59.)

[Gülcü 2003] C. Gülcü. *The Complete Log4j Manual: The Reliable, Fast and Flexible Logging Framework for Java*. QOS.ch, 2003. ISBN 2-9700369-0-8. (Cited on page 35.)

[Gupta 2005] S. Gupta. *Pro Apache Log4j*. Apress, 2nd edition, 2005. ISBN 978-1590594995. (Cited on pages 35 and 36.)

[Hall 1992] R. J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 296–306. ACM, 1992. ISBN 0-89791-504-6. doi:10.1145/143062.143147. (Cited on pages 116, 117, and 173.)

[Hamou-Lhadj 2005] A. Hamou-Lhadj. *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension*. PhD thesis, Ottawa-Carleton Institute for Computer Science, School of Information Technology and Engineering (SITE), University of Ottawa, 2005. (Cited on pages 122 and 155.)

[Hamou-Lhadj and Lethbridge 2004] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'04)*, pages 42–55. IBM Press, 2004. (Cited on page 122.)

[Hangal and Lam 2002] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 291–301, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi:10.1145/581339.581377. (Cited on page 132.)

[Harchol-Balter 2002] M. Harchol-Balter. Task assignment with unknown duration. *Jounal of the ACM*, 49(2):260–288, 2002. ISSN 0004-5411. doi:10.1145/506147.506154. (Cited on pages 16, 17, and 18.)

[Harchol-Balter 2008] M. Harchol-Balter. Scheduling for server farms: Approaches and open problems. In S. Kounev, I. Gorton, and K. Sachs, editors, *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW'08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 1–3, Heidelberg, June 2008. Springer Verlag. ISBN 978-3-540-69813-5. doi:10.1007/978-3-540-69814-2_1. (Cited on page 15.)

[Hauswirth et al. 2004] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-priented applications. *SIGPLAN Notes*, 39(10):251–269, 2004. ISSN 0362-1340. doi:10.1145/1035292.1028998. (Cited on pages 15 and 37.)

[Hawkins 1980] D. M. Hawkins. *Identification of Outliers*. Monographs on applied probability and statistics. Chapman and Hall, London, UK, 1980. ISBN 0-412-21900-X. (Cited on page 32.)

[Herbst et al. 2014] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning. *Concurrency and Computation - Practice and Experience, Special Issue with extended versions of the best papers from ICPE 2013*, 2014. doi:10.1002/cpe.3224. (Cited on page 127.)

[Herzog 2000] U. Herzog. Formal methods for performance evaluation. In E. Brinksma, H. Hermanns, and J.-P. Katoen, editors, *Euro Summer School on Trends in Computer Science*, volume 2090 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2000. ISBN 3-540-42479-2. doi:10.1007/3-540-44667-2_1. (Cited on pages 13, 14, 15, and 21.)

[Hocko and Kalibera 2010] M. Hocko and T. Kalibera. Reducing performance non-determinism via cache-aware page allocation strategies. In *Proceedings of the Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW'10)*, pages 223–233. ACM, Jan. 2010. ISBN 978-1-60558-563-5. doi:10.1145/1712605.1712640. (Cited on pages 19 and 20.)

[Hodge and Austin 2004] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22:85–126, Oct. 2004. ISSN 0269-2821. doi:10.1023/B:AIRE.0000045502.10941.a9. (Cited on pages 32 and 33.)

[Hoffman 2005] B. Hoffman. Monitoring, at your service. *Queue*, 3(10): 34–43, 2005. ISSN 1542-7730. doi:10.1145/1113322.1113335. (Cited on pages 1 and 38.)

[Hovemeyer and Pugh 2004] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, Dec. 2004. ISSN 0362-1340. doi:10.1145/1052883.1052895. (Cited on page 132.)

[Hrischuk et al. 1995] C. Hrischuk, J. Rolia, and C. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In *Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 399–409. IEEE, Jan. 1995. doi:10.1109/MASCOT.1995.378659. (Cited on page 122.)

[Hrischuk et al. 1999] C. Hrischuk, C. Murray Woodside, and J. Rolia. Trace-based load characterization for generating performance software models. *Transactions on Software Engineering*, 25(1):122–135, Jan. 1999. ISSN 0098-5589. doi:10.1109/32.748921. (Cited on page 122.)

[Huber et al. 2012] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev. S/T/A: Meta-modeling run-time adaptation in component-based system architectures. In *Proceedings of the 9th International Conference on e-Business Engineering (ICEBE'12)*, pages 70–77. IEEE, Sept. 2012. doi:10.1109/ICEBE.2012.21. (Cited on page 94.)

[Humble and Farley 2010] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010. ISBN 978-0321601919. (Cited on page 36.)

[Hutchins et al. 1994] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 191–200. IEEE, 1994. ISBN 0-8186-5855-X. (Cited on page 157.)

[Hyndman and Khandakar 2008] R. J. Hyndman and Y. Khandakar. Automatic time series forecasting: The forecast package for R. *Journal of Statistical Software*, 27(3):1–22, July 2008. ISSN 1548-7660. (Cited on page 135.)

[IEEE Standards Board 2002] IEEE Standards Board. *IEEE Standard Glossary of Software Engineering Terminology*. New York, NY, USA, 2002. IEEE Standard 610.12-1990, Initial release 1990, Reaffirmed 2002. (Cited on page 35.)

[Isermann 2006] R. Isermann. *Fault-Diagnosis Systems*. Springer Verlag, 1st edition, 2006. ISBN 3-540-24112-4. (Cited on pages 28, 29, 30, 31, 40, and 41.)

[Isermann and Ballé 1997] R. Isermann and P. Ballé. Trends in the application of model-based fault detection and diagnosis of technical processes. *Control Engineering Practice*, 5(5):709–719, 1997. ISSN 0967-0661. doi:10.1016/S0967-0661(97)00053-1. (Cited on page 27.)

[ISO/IEC 9126-1] *ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model*. ISO/IEC, June 2001. (Cited on page 9.)

[ISO/IEC 42010] *Recommended Practice for Architectural Description of Software-Intensive Systems. Also IEEE Standard 1471-2000.* ISO/IEC, 2006. (Cited on page 20.)

[Jain 1991] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1st edition, Apr. 1991. ISBN 0471503363. (Cited on pages 3, 5, 10, 11, 12, 15, 21, 32, 37, 91, 92, 93, 113, and 159.)

[Jiang et al. 2006] G. Jiang, H. Chen, and K. Yoshihira. Modeling and tracking of transaction flow dynamics for fault detection in complex systems. *IEEE Transactions on Dependable and Secure Computing*, 3:312–326, 2006. ISSN 1545-5971. doi:10.1109/TDSC.2006.52. (Cited on pages 58, 141, and 142.)

[Jones and Harrold 2005] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 273–282. ACM, 2005. ISBN 1-58113-993-4. doi:10.1145/1101908.1101949. (Cited on pages 28 and 132.)

[Jones et al. 2002] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 467–477. ACM, 2002. ISBN 1-58113-472-X. doi:10.1145/581339.581397. (Cited on page 28.)

[Jung et al. 2004] H.-W. Jung, S.-G. Kim, and C.-S. Chung. Measuring Software Product Quality: A Survey of ISO/IEC 9126. *IEEE Software*, 21 (5):88–92, 2004. ISSN 0740-7459. doi:10.1109/MS.2004.1331309. (Cited on page 9.)

[Kalibera 2006] T. Kalibera. *Performance in Software Development Cycle: Regression Benchmarking*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2006. (Cited on pages 1, 120, and 155.)

[Kalibera et al. 2005] T. Kalibera, L. Bulej, and P. Tuma. Automated detection of performance regressions: The mono experience. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, pages 183–190. IEEE, 2005. ISBN 0-7695-2458-3. doi:10.1109/MASCOT.2005.18. (Cited on page 129.)

[Kao et al. 1993] W.-I. Kao, R. Iyer, and D. Tang. FINE: A fault injection and monitoring environment for tracing the unix system behavior under faults. *Transactions on Software Engineering*, 19(11):1105–1118, 1993. ISSN 0098-5589. doi:10.1109/32.256857. (Cited on pages 32 and 40.)

[Kapova et al. 2010] L. Kapova, B. Buhnova, A. Martens, J. Happe, and R. H. Reussner. State dependence in performance evaluation of component-based software systems. In *Proceedings of the Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW'10)*, pages 37–48. ACM, Jan. 2010. ISBN 978-1-60558-563-5. (Cited on pages 19, 20, and 154.)

[Katzela and Schwartz 1995] I. Katzela and M. Schwartz. Schemes for fault identification in communication networks. *IEEE/ACM Transactions on Networking*, 3(6):753–764, Dec 1995. ISSN 1063-6692. doi:10.1109/90.477721. (Cited on pages 28 and 29.)

[Kelly 2005] T. Kelly. Detecting performance anomalies in global applications. In *Proceedings of the 2nd Conference on Real, Large Distributed Systems (WORLDS'05)*. USENIX Association, 2005. (Cited on pages 119, 120, and 124.)

[Kernighan and Pike 1999] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley Longman, 1999. ISBN 978-0201615869. (Cited on page 35.)

[Kiciman 2005] E. Kiciman. *Using Statistical Monitoring to Detect Failures in Internet Services*. PhD thesis, Stanford University, Sept. 2005. (Cited on pages 28, 44, 138, and 139.)

[Kiciman and Fox 2005] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041, Sept. 2005. doi:10.1109/TNN.2005.853411. (Cited on pages 40, 122, and 139.)

[King 2011] R. King. The top 10 programming languages. *IEEE Spectrum*, 48(10):84–84, Oct. 2011. ISSN 0018-9235. doi:10.1109/MSPEC.2011.6027266. (Cited on page 153.)

[Knuth 1989] D. E. Knuth. The errors of TEX. *Software–Practice and Experience*, 19(7):607–685, July 1989. (Cited on page 26.)

[Kogon and Williams 1998] S. M. Kogon and D. B. Williams. Characteristic function based estimation of stable distribution parameters. In R. J. Adler, R. E. Feldman, and M. S. Taqqu, editors, *A Practical Guide to Heavy Tails: Statistical Techniques and Applications*, pages 311–335. Birkhauser Boston Inc., 1998. ISBN 0-8176-3951-9. (Cited on page 16.)

[Koziolek 2008] H. Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, Carl von Ossietzky University of Oldenburg, Department of Computing Science, 2008. URL `http://sdqweb.ipd.uka.de/publications/pdfs/koziolek2008g.pdf`. Available online, Last access: 2014-04-12. (Cited on page 129.)

[Koziolek et al. 2008] H. Koziolek, S. Becker, and J. Happe. Predicting the Performance of Component-based Software Architectures with different Usage Profiles. In *Proceedings of the 3rd International Conference on the Quality of Software Architectures (QoSA'07)*, volume 4880 of *LNCS*, pages 145–163. Springer, 2008. doi:10.1007/978-3-540-77619-2. (Cited on pages 13, 21, 129, and 154.)

[Kreinovich and Kosheleva 2012] V. Kreinovich and O. Kosheleva. How to define mean, variance, etc., for heavy-tailed distributions: a fractal-motivated approach. Technical Report UTEP-CS-12-32, University of Texas at El Paso, Departmental Technical Reports (CS), 2012. URL `http://digitalcommons.utep.edu/cs_techrep/710`. (Cited on pages 152 and 154.)

[Kriegel et al. 2009] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek. LoOP: local outlier probabilities. In D. W.-L. Cheung, I.-Y. Song, W. W. Chu, X. Hu, and J. J. Lin, editors, *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM'09)*, pages 1649–1652. ACM, Nov. 2009. doi:10.1145/1645953.1646195. (Cited on page 55.)

[Kruegel and Vigna 2003] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 251–261. ACM, 2003. ISBN 1-58113-738-9. doi:10.1145/948109.948144. (Cited on page 31.)

[Kumar 1995] S. Kumar. *Classification and detection of computer intrusions*. PhD thesis, Purdue University, 1995. (Cited on pages 30 and 31.)

[Küng and Krause 2007] P. Küng and H. Krause. Why do software applications fail and what can software engineers do about it? a case study. In *Proceedings of the IRMA Conference on Managing Worldwide Operations and Communications with Information Technology*, pages 319–322. IGI Publishing, 2007. ISBN 978-1-59904-929-8. (Cited on page 26.)

[Lai and Wang 1995] M.-Y. Lai and S. Y. Wang. Software fault insertion testing for fault tolerance. In Lyu [1995], chapter 13, pages 315–333. ISBN 0471950688. (Cited on page 26.)

[Lamport 1978] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. ISSN 0001-0782. doi:10.1145/359545.359563. (Cited on page 37.)

[LaPadula 1999] L. J. LaPadula. State of the art in anomaly detection and reaction. Technical Report MP 99B0000020, MITRE, 1999. (Cited on page 30.)

[Laprie et al. 1995] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Architectural issues in software fault tolerance. In Lyu [1995], chapter 3, pages 47–80. ISBN 0471950688. (Cited on page 24.)

[Lashari and Srinivas 2003] G. Lashari and S. Srinivas. Characterizing java application performance. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003. doi:10.1109/IPDPS.2003.1213265. (Cited on pages 9, 15, 19, and 20.)

[Lee and Anderson 1990] P. A. Lee and T. Anderson. *Fault Tolerance : Principles and Practice*. Dependable computing and fault-tolerant systems. Springer-Verlag, 2nd edition, 1990. (Cited on page 29.)

[Li and Zhou 2005] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, pages 306–315. ACM, 2005. doi:10.1145/1081706.1081755. (Cited on page 132.)

[Littlewood and Strigini 2000] B. Littlewood and L. Strigini. Software reliability and dependability: a roadmap. In *Proceedings of the International Conference on Software Engineering (ICSE'00)*, pages 175–188. ACM, 2000. ISBN 1-58113-253-0. doi:10.1145/336512.336551. (Cited on page 24.)

[Liu et al. 2005] Y. Liu, A. Fekete, and I. Gorton. Design-level performance prediction of component-based applications. *Transactions on Software Engineering*, 31(11):928–941, 2005. ISSN 0098-5589. doi:10.1109/TSE.2005.127. (Cited on pages 14 and 19.)

[Lyu 1995] M. R. Lyu, editor. *Software Fault Tolerance*, 1995. John Wiley & Sons, Inc. ISBN 0471950688. (Cited on pages 207 and 208.)

[Lyu 2007] M. R. Lyu. Software reliability engineering: A roadmap. In *Proceedings of the International Conference on Software Engineering (ICSE'07), Future of Software Engineering (FOSE'07)*, pages 153–170. IEEE, 2007. ISBN 0-7695-2829-5. doi:10.1109/FOSE.2007.24. (Cited on pages 2 and 24.)

[Mansouri-Samani 1995] M. Mansouri-Samani. *Monitoring of Distributed Systems*. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, Department of Computing, 1995. (Cited on page 37.)

[Marick 1990] B. Marick. A survey of software fault surveys. Technical report, Department of Computer Science. University of Illinois at Urbana-Champaign, Dec. 1990. (Cited on page 26.)

[Marquard and Götz 2008] U. Marquard and C. Götz. Sap standard application benchmarks - it benchmarks with a business focus. In S. Kounev, I. Gorton, and K. Sachs, editors, *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW'08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 4–8, Heidelberg, June 2008. Springer Verlag. ISBN 978-3-540-69813-5. doi:10.1007/978-3-540-69814-2_2. (Cited on page 13.)

[Marwede et al. 2009] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In A. Winter, R. Ferenc, and J. Knodel, editors, *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, pages 47–57. IEEE, Mar. 2009. ISBN 978-0-7695-3589-0. doi:10.1109/CSMR.2009.15. (Cited on pages 2, 7, 28, and 58.)

[Marwede 2008] N. S. Marwede. Automatic failure diagnosis based on timing behavior anomaly correlation in distributed Java Web applications, Aug. 2008. Master's thesis (Diplomarbeit), Carl von Ossietzky University of Oldenburg, Department of Computing Science, Software Engineering Group. (Cited on page 7.)

[Maxion and Tan 2000] R. Maxion and K. Tan. Benchmarking anomaly-based detection systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'00)*, pages 623–630. IEEE, 2000. doi:10.1109/ICDSN.2000.857599. (Cited on page 143.)

[Maxion 1990] R. A. Maxion. Anomaly detection for diagnosis. In B. Randell, editor, *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS'90)*, pages 20–27. IEEE, June 1990. ISBN 0-8186-2051-X. doi:10.1109/FTCS.1990.89362. (Cited on pages 22, 30, 58, 124, and 131.)

[Meier 2007] R. Meier. Techniques for minimizing the performance impact of java garbage collection across your system landscape. *SAP Professional Journal*, 9(3), May 2007. (Cited on page 22.)

[Menascé and Almeida 2001] D. A. Menascé and V. A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, Oct. 2001. ISBN 0-13-065903-7. (Cited on pages 2, 4, 5, 12, 13, 16, 17, 21, 91, 128, and 145.)

[Menascé et al. 1999] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of e-commerce sites. In *Proceedings of the Conference on Electronic Commerce (EC'99)*, pages 119–128. ACM, 1999. ISBN 1-58113-176-3. doi:10.1145/336992.337024. (Cited on pages 1 and 38.)

[Mielke 2006] A. Mielke. Elements for response-time statistics in ERP transaction systems. *Performance Evaluation*, 63(7):635–653, July 2006. doi:j.peva.2005.05.006. (Cited on pages 2, 15, 16, 18, and 40.)

[Mitrani 1982] I. Mitrani. *Simulation techniques for discrete event systems*. Cambridge University Press, 1982. ISBN 0521238854. (Cited on page 3.)

[Montgomery and Runger 2003] D. C. Montgomery and G. C. Runger. *Applied Statistics and Probability for Engineers*. John Wiley & Sons, Inc., 3rd edition, 2003. ISBN 0-471-20454-4. (Cited on page 35.)

[Mos and Murphy 2004] A. Mos and J. Murphy. Compas: Adaptive performance monitoring of component-based systems. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS'04)*, pages 35–40. IEEE, May 2004. ISBN 0-86341-430-3. doi:10.1049/ic:20040348. (Cited on page 138.)

[Munoz-Garcia et al. 1990] J. Munoz-Garcia, J. L. Moreno-Rebollo, and A. Pascual-Acosta. Outliers: A formal approach. *International Statistical Review / Revue Internationale de Statistique*, 58(3):215–226, 1990. ISSN 03067734. (Cited on page 32.)

[Musa 2004] J. D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. Author House, 2nd edition, 2004. ISBN 978-1418493882. (Cited on pages 24 and 25.)

[Musa et al. 1987] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, New

York, 1st edition, 1987. ISBN 0-07-044093-X. (Cited on pages 10, 24, 25, and 29.)

[Nagappan et al. 2006] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 452–461. ACM, 2006. ISBN 1-59593-375-1. doi:10.1145/1134285.1134349. (Cited on page 27.)

[Nethercote and Seward 2007] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 89–100. ACM, 2007. ISBN 978-1-59593-633-2. doi:10.1145/1250734.1250746. (Cited on page 174.)

[Network Reliability Council 1993] Network Reliability Council. Network reliability: A report to the nation, June 1993. (Cited on page 26.)

[Nou et al. 2008] R. Nou, S. Kounev, F. Julia, and J. Torres. Autonomic QoS control in enterprise Grid environments using online simulation. *Journal of Systems and Software*, 2008. (Cited on page 5.)

[Object Management Group (OMG) 2007] Object Management Group (OMG). Unified Modeling Language: Superstructure Version 2.1.1, Feb. 2007. (Cited on page 43.)

[Oracle 2013] Oracle. Java hotspot garbage collection: The Garbage-First garbage collector, Oct. 2013. URL `http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html`. Available online, Last Access: 2014-05-08. (Cited on page 22.)

[Oracle 2014] Oracle. Java SE 8 - Java platform stnadrd edition tools reference, Mar. 2014. URL `http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html`. Available online, Last Access: 2014-06-07. (Cited on page 23.)

[Ostrand and Weyuker 2002] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 55–64. ACM, 2002. ISBN 1-58113-562-9. doi:10.1145/566172.566181. (Cited on page 27.)

[Patcha and Park 2007] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007. ISSN 1389-1286. doi:10.1016/j.comnet.2007.02.001. (Cited on pages 30 and 32.)

[Patterson and Hennessy 2008] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition, 2008. ISBN 978-0123744937. (Cited on page 10.)

[Paxson 1994] V. Paxson. Empirically-derived analytic models of wide-area tcp connections. *IEEE/ACM Transactions on Networking*, 2(4), 1994. (Cited on pages 16 and 17.)

[Pertet and Narasimhan 2005] S. Pertet and P. Narasimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, School of Computer science & Electrical and Computer Engineering, Dec. 2005. (Cited on page 28.)

[Pitakrat 2013] T. Pitakrat. Hora: Online failure prediction framework for component-based software systems based on kieker and palladio. In *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*, pages 39–48. CEUR-WS.org, Nov. 2013. (Cited on page 138.)

[Pitakrat et al. 2014] T. Pitakrat, A. van Hoorn, and L. Grunske. Increasing dependability of component-based software systems by online failure prediction. In *Proceedings of the Tenth European Conference on Dependable Computing (EDCC'14)*, pages 66–69. IEEE, 2014. doi:10.1109/EDCC.2014.28. (Cited on pages 138 and 152.)

[Ploski et al. 2007] J. Ploski, M. Rohr, P. Schwenkenberg, and W. Hasselbring. Research Issues in Software Fault Categorization. *SIGSOFT Software Engineering Notes*, 32(6):1–8, Nov. 2007. ISSN 0163-5948. doi:10.1145/1317471.1317478. (Cited on pages 7 and 26.)

[Printezis 2004] T. Printezis. Garbage collection in the java hotspot virtual machine, Sept. 2004. URL `http://www.devx.com/Java/Article/21977/1954?pf=true`. Available online, Last access: 2014-05-03. (Cited on page 22.)

[Psounis et al. 2005] K. Psounis, P. Molinero-Fernández, B. Prabhakar, and F. Papadopoulos. Systems with multiple servers under heavy-tailed workloads. *Performance Evaluation*, 62(1-4):456–474, 2005. ISSN 0166-5316. doi:10.1016/j.peva.2005.07.030. (Cited on page 18.)

[Reps et al. 1997] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Conference on Software Engineering (ESEC'97)*, pages 432–449. Springer, 1997. ISBN 3-540-63531-9. doi:10.1145/267895.267925. (Cited on pages 28 and 132.)

[Reynolds et al. 2006a] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. PIP: detecting the unexpected in distributed systems. In *Proceedings of the 3rd Symposium on Networked Systems Design & Implementation (NSDI'06)*, Berkeley, CA, USA, 2006a. USENIX. (Cited on page 144.)

[Reynolds et al. 2006b] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th International Conference on World Wide Web (WWW'06)*, pages 347–356. ACM, 2006b. ISBN 1-59593-323-9. doi:10.1145/1135777.1135830. (Cited on page 122.)

[Rohr 2007] M. Rohr. Timing behavior anomaly detection for automatic failure detection and diagnosis, Apr. 2007. URL `http://d3s.mff.cuni.cz/research/seminar/download/2007-04-10-Rohr-TimingAnomaly.pdf`. Available online, Last access: 2014-04-03. Presentation at Charles Univerity Prague. (Cited on page 1.)

[Rohr et al. 2007] M. Rohr, S. Giesecke, and W. Hasselbring. Timing Behavior Anomaly Detection in Enterprise Information Systems. In J. Cardoso, J. Cordeiro, and J. Filipe, editors, *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS'07)*, volume DISI, pages 494–497. INSTICC Press, June 2007. ISBN 978-972-8865-88-7. (Cited on page 7.)

[Rohr et al. 2008a] M. Rohr, A. van Hoorn, S. Giesecke, J. Matevska, and W. Hasselbring. Trace-context sensitive performance models from monitoring data of software systems. In C. Lebsack, editor, *Proceedings of the Workshop on Tools Infrastructures and Methodologies for the Evaluation of Research Systems (TIMERS'08) at IEEE International Symposium on Performance Analysis of Systems and Software 2008*, pages 37–44, Apr. 2008a. (Cited on pages 6, 18, 19, and 81.)

[Rohr et al. 2008b] M. Rohr, A. van Hoorn, S. Giesecke, J. Matevska, W. Hasselbring, and S. Alekseev. Trace-context sensitive performance

profiling for enterprise software applications. In S. Kounev, I. Gorton, and K. Sachs, editors, *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW'08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 283–302, Heidelberg, June 2008b. Springer Verlag. ISBN 978-3-540-69813-5. doi:10.1007/978-3-540-69814-2_18. (Cited on page 6.)

[Rohr et al. 2008c] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering 2008*, pages 80–85. ACTA Press, Feb. 2008c. ISBN 978-0-88986-715-4. (Cited on pages 4, 7, 46, and 47.)

[Rohr et al. 2010] M. Rohr, A. van Hoorn, W. Hasselbring, M. Lübcke, and S. Alekseev. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In *Proceedings of the Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW'10)*, pages 87–92. ACM, Jan. 2010. ISBN 978-1-60558-563-5. doi:10.1145/1712605.1712621. (Cited on pages 7, 10, 49, and 141.)

[Rolia and Sevcik 1995] J. Rolia and K. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, 1995. ISSN 0098-5589. doi:10.1109/32.403785. (Cited on page 14.)

[Rygielski and Tomczak 2011] P. Rygielski and J. M. Tomczak. Context change detection for resource allocation in service-oriented systems. In *Proceedings of the 15th International Conference on Knowledge-based and Intelligent Information and Engineering Systems (KES'11) - Volume Part II*, pages 591–600, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23862-8. (Cited on pages 131 and 132.)

[Sabetta and Koziolek 2008] A. Sabetta and H. Koziolek. Measuring performance metrics: Techniques and tools. In I. Eusgeld, F. Freiling, and R. Reussner, editors, *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science (LNCS)*, pages 226–232. Springer, 2008. ISBN 978-3-540-68946-1. doi:10.1007/978-3-540-68947-8_21. (Cited on pages 9, 12, 13, and 15.)

[Salfner and Malek 2005] F. Salfner and M. Malek. Proactive fault handling for system availability enhancement. In *Proceedings of the*

DPDNS Workshop in conjunction with IPDPS 2005, Denver, Colorado, 2005. IEEE. (Cited on page 152.)

[Sambasivan et al. 2011] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*, Berkeley, CA, USA, 2011. USENIX Association. (Cited on pages 15, 20, 37, 40, 44, 120, and 121.)

[Scherr 1965] A. L. Scherr. An analysis of time-shared computer systems. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1965. (Cited on page 21.)

[Schroeder and Gibson 2006] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 249–258. IEEE, 2006. ISBN 0-7695-2607-1. doi:10.1109/DSN.2006.5. (Cited on page 26.)

[Schwenkenberg 2007] P. Schwenkenberg. Auswirkung von Programmierfehlern auf Softwarezeitverhalten, Aug. 2007. Master's thesis (Diplomarbeit), Carl von Ossietzky University of Oldenburg, Department of Computing Science, Software Engineering Group. (Cited on pages 7 and 62.)

[Shewhart 1931] W. A. Shewhart. *Economic Control of Quality of Manufactured Product*. D. Van Nostrand Company, 1931. (Cited on pages 3, 29, 33, and 140.)

[Sigelman et al. 2010] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and A. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure, 2010. Technical Report. (Cited on pages 36 and 37.)

[Silva 2008] L. M. Silva. Comparing error detection techniques for web applications: An experimental study. In *Proceedings of the 7th International Symposium on Network Computing and Applications (NCA'08)*, pages 144–151. IEEE, July 2008. doi:10.1109/NCA.2008.57. (Cited on page 157.)

[Silverman 1986] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, New York, 1986. (Cited on page 55.)

[Simon 2006] D. Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley & Sons, 1st edition, Aug. 2006. ISBN 0471708585. (Cited on page 18.)

[Smith and Williams 2001a] C. U. Smith and L. G. Williams. Software performance antipatterns; common performance problems and their solutions. In *Proceedings of the 27th International Computer Measurement Group Conference (CMG'01)*, pages 797–806, 2001a. (Cited on pages 12, 13, 20, and 22.)

[Smith and Williams 2001b] C. U. Smith and L. G. Williams. *Performance Solutions: A practical guide to creating responsive, scalable software*. Addision-Wesley, 2001b. ISBN 978-09291722291. (Cited on pages 1, 5, 9, 10, 12, 13, 15, 20, and 36.)

[Smith and Williams 2002] C. U. Smith and L. G. Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *Proceedings of the 28th International Computer Measurement Group Conference (CMG'02)*, pages 667–674. Computer Measurement Group, 2002. (Cited on page 20.)

[Sommer 2007] N. Sommer. Evaluation of control flow traces in software applications for intrusion detection, 2007. 4 month undergraduate thesis, Carl von Ossietzky University of Oldenburg, Department of Computing Science, Software Engineering Group. (Cited on page 7.)

[Spivey 2004] J. M. Spivey. Fast, accurate call graph profiling. *Software – Practice & Experience*, 34(3):249–264, 2004. ISSN 0038-0644. doi:10.1002/spe.562. (Cited on page 15.)

[Steinder and Sethi 2001] M. Steinder and A. S. Sethi. The present and future of event correlation: A need for end-to-end service fault localization. In *Proceedings of the IIIS SCI World Multi-Conference on Systemics, Cybernetics and Informatics 2001*, pages 124–129, 2001. (Cited on page 40.)

[Steinder and Sethi 2004] M. Steinder and A. S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165–194, Nov. 2004. doi:10.1016/j.scico.2004.01.010. (Cited on pages 28, 29, 30, and 31.)

[Stewart and Shen 2005] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the Symposium on Networked Systems Design & Implementation (NSDI'05)*, pages 71–84. USENIX, 2005. (Cited on pages 9, 13, 15, 21, and 22.)

[Stransky 2006] F. Stransky. Automatisierte Lokalisierung von Fehlerursachen bei Performance-Problemen in J2EE Anwendungen, 2006. University of Oldenburg, Software Engineering Group, Department of Computing Science, (4 month undergraduate thesis). (Cited on page 7.)

[Sun Microsystems 2006] Sun Microsystems. Memory management in the java hotspot virtual machine, Apr. 2006. URL `http://java.sun.com/javase/technologies/hotspot/gc/memorymanagement_whitepaper.pdf`. Available online, Last access: 2008-08-29. (Cited on page 22.)

[Sun Microsystems, Inc. 1994-2006] Sun Microsystems, Inc. Java BluePrints: Guidelines, patterns, and code for end-to-end applications. http://java.sun.com/reference/blueprints/, 1994-2006. Available online, Last access: 2007-10-10. (Cited on page 80.)

[Tan et al. 2006] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2006. (Cited on page 32.)

[Tan et al. 2010] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'10)*, pages 173–182. ACM, 2010. ISBN 978-1-60558-888-9. doi:10.1145/1835698.1835741. (Cited on pages 130 and 154.)

[Thereska and Ganger 2008] E. Thereska and G. R. Ganger. Ironmodel: robust performance models in the wild. *SIGMETRICS Performance Evaluation Review*, 36(1):253–264, 2008. ISSN 0163-5999. doi:10.1145/1384529.1375486. (Cited on page 14.)

[Thereska et al. 2006] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. In *Proceedings of the joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'06)*, pages 3–14. ACM, 2006. ISBN 1-59593-319-0. doi:10.1145/1140277.1140280. (Cited on pages 36, 37, 117, and 122.)

[Tiffany 2002] M. Tiffany. A survey of event correlation techniques and related topics, 2002. URL `http://ns1.tiffman.com/netman/netman.pdf`. Available online, Last access: 2013-01-17. (Cited on pages 30 and 31.)

[Vallamsetty et al. 2003] U. Vallamsetty, K. Kant, and P. Mohapatra. Characterization of e-commerce traffic. *Electronic Commerce Research*, 3: 167–192, 2003. ISSN 1389-5753. doi:10.1023/A:1021585529079. (Cited on page 18.)

[van Hoorn 2007] A. van Hoorn. Workload-sensitive timing behavior anomaly detection in large software systems, Sept. 2007. Master's thesis (Diplomarbeit), Carl von Ossietzky University of Oldenburg, Department of Computing Science, Software Engineering Group. (Cited on pages 2, 7, 10, 15, 16, 18, 37, 80, and 156.)

[van Hoorn 2014] A. van Hoorn. *Model-Driven Online Capacity Management for Component-Based Software Systems*. Phd thesis, Faculty of Engineering, Kiel University, Kiel, Germany, Oct. 2014. URL `http://eprints.uni-kiel.de/25969/`. (Cited on pages 1 and 94.)

[van Hoorn et al. 2008] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In S. Kounev, I. Gorton, and K. Sachs, editors, *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW'08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 124–143, Heidelberg, June 2008. SPEC, Springer Verlag. ISBN 978-3-540-69813-5. doi:10.1007/978-3-540-69814-2_9. (Cited on pages 62, 80, and 105.)

[van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Kiel University, Department of Computer Science, Nov. 2009. (Cited on pages 4, 15, 36, 37, 46, 69, 80, 121, 150, and 153.)

[van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE'12)*, pages 247–248. ACM, Apr. 2012. doi:10.1145/2188286.2188326. (Cited on pages 150 and 153.)

[Viswanathan and Liang 2000] D. Viswanathan and S. Liang. Java virtual machine profiler interface. *IBM Systems Journal*, 29(1):82–95, Feb. 2000. (Cited on page 15.)

[Waheed and Rover 1995] A. Waheed and D. Rover. A structured approach to instrumentation system development and evaluation. In *Proceedings of the IEEE/ACM Conference on Supercomputing (SC'95)*, page 30, 1995. doi:10.1109/SUPERC.1995.242930. (Cited on page 35.)

[Waller 2014] J. Waller. *Performance Benchmarking of Application Monitoring Frameworks*. Number 2014/5 in Kiel Computer Science Series. Department of Computer Science, Kiel University, Dec. 2014. ISBN 978-3-7357-7853-6. Dissertation, Faculty of Engineering, Kiel University. (Cited on page 153.)

[Wang et al. 2014] T. Wang, J. Wei, W. Zhang, H. Zhong, and T. Huang. Workload-aware anomaly detection for web applications. *Journal of Systems and Software*, 89:19–32, 2014. doi:10.1016/j.jss.2013.03.060. (Cited on page 134.)

[Wasylkowski et al. 2007] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the 6th European Software Engineering Conference and the Symposium on Foundations of Software Engineering (ESEC/FSE'07)*, pages 35–44. ACM, 2007. ISBN 978-1-59593-811-4. doi:10.1145/1287624.1287632. (Cited on page 132.)

[Wilcox 2010] R. R. Wilcox. *Fundamentals of Modern Statistical Methods - Substantially Improving Power and Accuracy*. Springer, 2nd edition, 2010. ISBN 978-1-4419-5524-1. doi:10.1007/978-1-4419-5525-8. (Cited on pages 152 and 154.)

[Williams et al. 2007] A. W. Williams, S. M. Pertet, and P. Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–8. IEEE, Mar. 2007. doi:10.1109/IPDPS.2007.370345. (Cited on pages 40, 41, 58, 130, 131, and 139.)

[Wong and Debroy 2009] W. E. Wong and V. Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, University of Texas at Dallas, Nov. 2009. (Cited on page 28.)

[Woodside 2008] M. Woodside. The relationship of performance models to data. In *Proceedings of the SPEC International Performance Evaluation*

Workshop (SIPEW'08), pages 9–28. Springer-Verlag, 2008. ISBN 978-3-540-69813-5. doi:10.1007/978-3-540-69814-2_3. (Cited on pages 12, 13, and 36.)

[Woodside et al. 2007] M. Woodside, G. Franks, and D. C. Petriu. Future of software performance engineering. In *Proceedings of the Workshop on the Future of Software Engineering (FOSE'07)*, pages 171–187. IEEE, 2007. ISBN 0-7695-2829-5. doi:10.1109/FOSE.2007.32. (Cited on page 18.)

[Xie and Notkin 2002] T. Xie and D. Notkin. An empirical study of Java dynamic call graph extractors. Technical Report UW-CSE-02-12-03, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Dec. 2002. (Cited on pages 15, 116, and 123.)

[Yilmaz et al. 2008] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: fault localization using time spectra. In *Proceedings of the 13th International Conference on Software Engineering (ICSE'08)*, pages 81–90. ACM, 2008. ISBN 978-1-60558-079-1. doi:10.1145/1368088.1368100. (Cited on pages 1, 9, 28, 33, 40, 41, 119, 133, 134, and 141.)

[Zeller 2002] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'02/FSE-10)*, pages 1–10. ACM, 2002. ISBN 1-58113-514-9. doi:10.1145/587051.587053. (Cited on pages 28 and 132.)

[Zhang et al. 2007] Q. Zhang, L. Cherkasova, G. Matthews, W. Greene, and E. Smirni. R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads. Technical Report HPL-2007-87, Enterprise Systems and Software Laboratory, HP Laboratories Palo Alto, 2007. URL `http://www.hpl.hp.com/techreports/2007/HPL-2007-87.pdf`. Available online, Last access: 2014-05-03. (Cited on pages 22, 40, and 126.)

[Zhang et al. 2005] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'05)*, pages 644–653. IEEE, 2005. ISBN 0-7695-2282-3. doi:10.1109/DSN.2005.44. (Cited on pages 124 and 140.)

[Zhou et al. 2004] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the International Symposium on Microarchitecture (MICRO'04)*, pages 269–280. IEEE, Dec. 2004. doi:10.1109/MICRO.2004.3. (Cited on page 132.)

Index

Anomaly, 32
Anomaly detection, 30, 32
 semi-supervised, 33
 supervised, 33
 unsupervised, 33
Anomaly detector, 53
Anomaly rating, 58
Anomaly score, 54
Arrival rate, 12
Availability, 25

Call action
 asynchronous, 43
 synchronous, 43
Call graph profiling, 116
 tools, 171
Callee, 43
Caller, 43
Caller context, 67, 70
Causal path, 44
Context
 caller, 67, 70
 stack, 67, 70
 trace, 67, 70
CPU time, 10

DCT, 48, 69
Diagnostic checking, 29
Dynamic call graph, 116
Dynamic call tree, 48, 69, 116

e, 44
eoi, 45
Error, 23
ess, 45
ev, 100
Execution, 44
Execution environment, 43, 45
Execution order index, 45
Execution time, 10, 45

Failure, 23
Failure diagnosis, 27
False alarm, 34
Fault, 23
 software, 24
Fault diagnosis, 30
Fault hypothesis testing, 30
Fault isolation, 27
Fault localization, 27
Fault removal, 27
Fault symptom detection, 29

Garbage collection, 22, 151, 179

Heavy-tailed distribution, 16

Individual request characteristics, 13
Instrumentation, 37

Limit checking, 29
Logging, 35

Message, 47
Message trace, 47
Misuse pattern detection, 31
Monitoring, 35
  event driven, 37
  instrumentation, 37
  probes, 37
  sampling, 37
  timer driven, 37
Monitoring data, 44
Multimodal, 18

nt, 44

o, 44
Operation, 10, 43
Outlier, 32

Path profiling, 116
Performance, 9
  prediction, 14
Performance prediction, 14
Platform workload intensity, 97
Profiling, 15, 36
  call graph, 116
  path profiling, 116
  tools, 123, 171
  transactional, 117, 118
Program comprehension, 122
pwi, 97
  pwi_1, 98
  pwi_2, 99
  pwi_3, 99
  pwi_4, 100

Real-time systems, 13
Regression benchmarking, 15
Request flow, 44
Request path, 44
Resource utilization, 9, 12
Resource-sharing systems, 13
Response time, 10, 45
  end-to-end, 10
  operation, 10

Scalability, 13
Service demand, 13
Signature-based detection, 31
Software operation, 10
st, 44
Stack context, 67, 70
Standard deviation reduction, 78
standard deviation reduction, 167
Synchronous communication, 44
System layers, 36

Threshold, 29
Throughput, 12
Timing behavior
  definition, 9
tr, 44
Trace, 37, 44
  message, 47
  monitored, 44
  shape, 64, 69
Trace context, 67, 70
Trace shape context tree, 75
Trace synthesis, 47
$traceid$, 44
TracSTA, 63
Transactional profiling, 117, 118
TSCT, 75

Utilization, 12

Virtualization, 151, 154
$vmid$, 44

WiSTA, 91
Workload, 12
Workload intensity, 12