Trace-context Sensitive Performance Models from Monitoring Data of Software Systems*

Matthias Rohr, André van Hoorn, Simon Giesecke, Jasminka Matevska, and Wilhelm Hasselbring
Software Engineering Group
University of Oldenburg
Oldenburg, Germany
E-Mail: {Rohr | van.Hoorn | Giesecke | Matevska | Hasselbring}@Informatik.Uni-Oldenburg.DE

***Abstract*—Operation response times in software systems are typically modeled by probability distributions. However, particularly in Java EE applications, operation response time distributions are often of high variance or multi-modal. Such characteristics reduce confidence or applicability in various statistical evaluations. We observed that calling-context information, e.g., the complete call path within the system, is often connected to a significant part of this variance and other undesired distribution characteristics.**

This paper introduces an approach to analyzing operation response times in the context of the complete call trace. This results in response time distributions that are specific to trace-contexts. We present empirical results of a medium-size online store demo application on the benefits of using trace-context specific response time distributions. The results are compared to the use of other or no calling-context information.

Empirical support is presented that trace-context analysis can create response time distributions with lower variance compared to using less or no calling-context information. Based on trace-context analysis, multi-modal distributions could be replaced by multiple unimodal distributions.

I. Introduction

Response time monitoring data is a valuable artifact for software performance analysis of software systems, such as enterprise information systems based on Java EE. For instance, response time data from such systems is used for online performance evaluation, such as performance optimization and failure diagnosis, and for offline performance evaluation, such as performance tuning, benchmarking, profiling, and performance prediction. Typically, not only end-to-end response times are considered, but also response times of operations (alternatively called methods, routines, procedures, or (sometimes) services), i.e., software architecture entities that group statements to larger blocks within a software system.

Enterprise software applications are usually deployed in middleware environments that do not provide real-time properties and show non-trivial scheduling and queueing behavior. These systems typically have to serve large numbers of concurrent and heterogeneous user requests competing for computational resources. Therefore, the timing behavior of such systems usually shows high variance and follows complex distributions. Unfortunately, many analytical and statistical performance evaluation approaches may produce low quality results for such timing behavior or cannot handle complex distribution families.

Operation executions show specific timing behavior for the calling-context of an operation execution, which is given by the call trace that corresponds to the execution of an operation. We discovered that a significant part of undesired distribution characteristics result from calling-context specific timing behavior of software operations. Therefore, we conclude that using calling-context information can improve timing behavior evaluations, such as those that depend on the variance of response time distributions.

In this paper, we present our approach to including trace-context information into software timing behavior models. For this, trace-context specific response time distributions are derived by combining response time monitoring to call trace analysis. Each operation's response times are partitioned using trace analysis. This results in multiple, trace-context specific response time distributions for each operation.

We contribute empirical data on trace-context specific timing behavior distributions in a non-trivial Java online store demo application. A workload driver is used to simulate probabilistic concurrent system usage. In the case study, trace-context analysis is also compared to two other types of calling-context types (stack-context, caller-context). Furthermore, it is analyzed how many calling-contexts exist in dependence to the number of monitoring points. Finally, we present quantitative and qualitative data showing that trace-context analysis helps to cope with operation response time distributions, which have high variance or undesired properties, such as multi-modality.

The document is structured as follows. In Section II we present the problem of calling-context dependence in software timing behavior. Our approach to modeling timing behavior in dependence to calling-contexts is presented in Section III. The case study is presented in Section IV. A discussion of the benefits and limitations of our approach is made in Section V before the related work and the conclusions follow in Section VI and VII.

II. Calling-context Dependence of Software Response Time Distributions

A. Software Response Time Distribution Characteristics

The duration of an operation execution, including the time spent in other operations that are invoked by the execution,

*This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

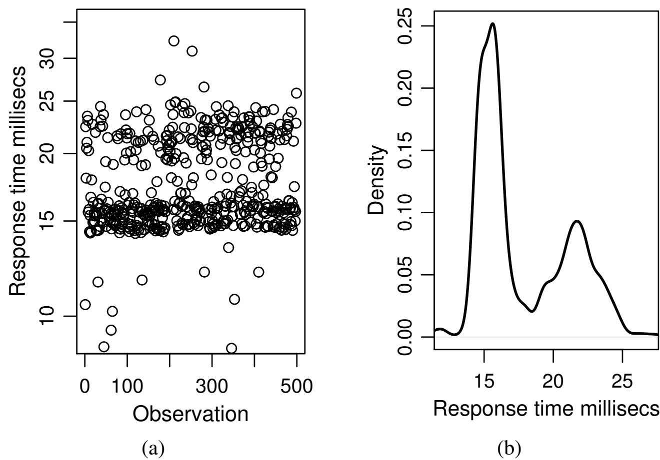


Fig. 1. "Clusters" and multi-modality of software response time distributions. (a) Response time scatter plot of an operation (`writeStringOrNull`) of the Dacapo Eclipse Benchmark [5], (b) shows its probability density estimate.

is denoted its response time [12]. Software performance models typically describe operation response times by response time sample values, probability distributions, or parametrized standard distribution families.

Response time distributions of operations in software systems, such as in Java EE applications, often show high variance and do not follow simple distribution families, such as exponential or normal distributions.

For instance, the response time distribution shown in Figure 1 of an operation measured in the Dacapo Eclipse Benchmark [5] cannot be accurately described by a single exponential or normal distribution.

Another example of multi-modal timing behavior distributions is provided by Bulej et al. [7]. These authors reported multi-modal response time distributions in different versions of CORBA middleware and use the term "cluster" for each group of similar response times. Bulej et al. [7] illustrate that clusters in timing behavior measurements reduce the potential to detect changes in the timing behavior of software. The authors experienced this problem in the context of performance regression benchmarking, which aims at detecting regressions in software performance between different versions of a software product.

High variance in response time distribution reduces the confidence in various statistical evaluations. An example for such an evaluation is the statistical hypothesis test that two response time observation sets belong to the same distribution. The confidence of this test usually decreases by increasing standard deviation, or more samples are required to reach the same confidence. Complex distributions, e.g. showing multi-modality, are not usable in many performance evaluation approaches because of mathematical tractability. Approximating complex distributions of response time measurements using simple distribution families is an option to satisfy requirements of performance evaluation approaches, but may lead to low quality results.

B. *Calling-context specific timing behavior*

Different timing behavior can correspond to multiple calling-contexts for the same software operation. Possible reasons are that the contexts correspond to particular software system states or operations show different timing behavior when they are used in different types of service requests. An example for the first is that a system provides different levels of personalization depending on the current workload intensity [3]; an example for the latter is that the response time of a service might heavily depend on the type of the request e.g., a watermarking service in an online media store might show different response time distributions for different media types that use individual watermarking techniques.

Calling-context is the set of circumstances or facts that surround an operation call. Software operation executions are embedded in sequences of interacting operation executions that participate in answering external service requests (from users or other systems). We consider three simplified models of the general calling-context that consider different parts of the execution sequence of an execution: caller-context, stack-context, and trace-context. These models will be described in more detail the next section.

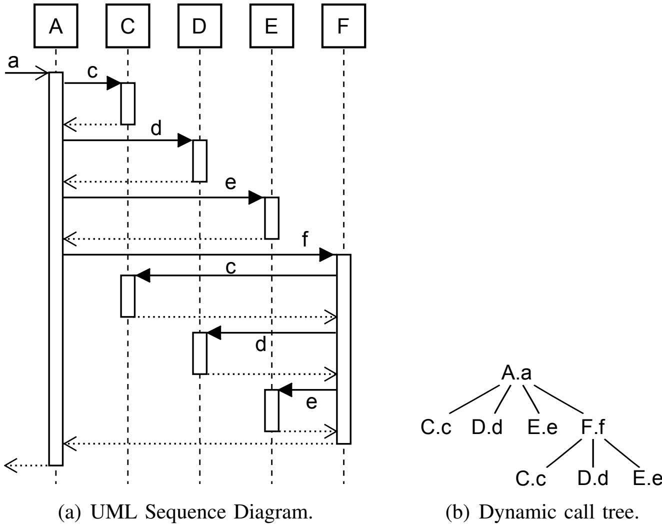
Many aspects of the context of an operation execution are relevant to performance analysis. A key activity of performance modeling is the selection of the relevant aspects to consider. Obviously, the more such aspects are included, the higher precision can be expected from performance analysis. Modeling all relevant aspects to timing behavior is usually not an option, since the overall modeling and analysis effort grows by increasing modeling detail. Additionally, in some cases such as performance modeling during the early design of a software system, relevant context information may be unknown and it has to be decided whether unknown relevant context information is estimated and included, or if it is excluded from the performance model.

The fact that the response time distribution of an operation is composed of response times made in different calling-contexts is often responsible for a significant part of the distribution variance or multi-modality. The use of relevant calling-context information can improve timing behavior evaluations that are sensible to high variance or multi-modality in response time distributions, such as many anomaly detection approaches.

III. Approach to Calling-context Sensitive Timing Behavior Modeling

In this section, we describe how calling-context information can be used in timing behavior modeling. We present and compare three different levels of calling-context information: caller-context, stack-context, and trace-context. Caller-context and stack-context information have been used in performance evaluation before (see Ammons et al. [2] and Graham et al. [11]).

Correlating the timing behavior of operation executions to the complete context given by the trace, i.e., the sequence of monitored execution events that are connected by synchronous communication, has not been published before. It is our



(a) UML Sequence Diagram. (b) Dynamic call tree.

Fig. 2. Two representations of the same trace.

conjecture that the complete "trace-shape" covers much more information that can be used to explain timing behavior than the caller- or stack-context information. In particular, trace-context analysis includes information on sub-calls made by an operation. Sub-call information is omitted by stack- and caller-context analysis. Of course, for response time evaluation, it is highly relevant whether and which sub-calls were made if more than one scenario is possible.

A. Software Behavior and its Monitoring

We assume that software systems are composed of components. The components provide *operations* that might be requested by other components, external users, or systems.

Primary artifacts of runtime behavior are *executions* of the operations. We define a monitored execution as tuple (o, i, r, st) of an operation o, its response time r, a start time st, and an identifier i to distinguish executions of the same operation. The *response time* of an execution is the number of time units (e.g., milliseconds) between the start and the end of an execution, i.e., it includes the time spent waiting for returns from other executions that are invoked by the callee.

A *trace* is a finite sequence of operation executions that results from a user request or request of an external system. We limit the scope to synchronous communication between executions (cp. UML 2.1.1 [14]): the caller of an operation is blocked and has to wait until the callee returns a result before it continues its own execution. As an example, Figure 2 illustrates two traces as UML Sequence Diagram.

A trace can be represented by a *dynamic call tree* [2]. Each node of such an *ordered* tree represents an operation execution by its operation name. An edge from one node to another, i.e. their parent-child relation, corresponds to the caller/callee relation within the trace. Figure 2(b) shows the dynamic call tree that corresponds to the trace shown in Figure 2(a).

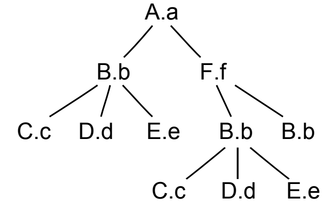


Fig. 3. Dynamic call tree.

B. Types of Calling-context Equivalence Classes

It is our goal to partition operation response times that are within equivalent calling-contexts. In the following we specify three equivalence relations:

- *Caller-context equivalence:* Two executions of the same operation are caller-context equivalent if they are called from operations with the same name.
- *Stack-context equivalence:* Two executions of the same operation are stack-context equivalent if the paths from the corresponding nodes to its root are equal.
- *Trace-context equivalence:* Two executions of the same operation are trace-context equivalent if the corresponding trees are equal and the both executions correspond to dynamic call tree nodes with the same position within the tree.

Trace-context equivalence implies stack-context equivalence and stack-context equivalence implies caller-context equivalence. If the monitoring data for a system consists only of traces conforming to the trace representations illustrated in Figure 3. Calling-context analysis would discover 7 caller-contexts, 10 stack-contexts (the 2nd and 3rd B.b are stack-context equivalent), and 11 trace-contexts.

Each of the three equivalence relations specifies a partitioning of the monitored executions and its response times into equivalence classes. In the following, we use the terms *caller-*, *stack-*, and *trace-context* to refer to an equivalence class of executions that are caller-, stack-, and trace-context equivalent respectively. The term calling-context refers to any of those three equivalence classes.

In general, the number of distinct calling-contexts tends to grow by the number of monitoring points. Adding a new monitoring point to an existing instrumentation also increases the number of trace-contexts and stack-contexts. Adding a monitoring point usually also increases the number of caller-contexts, but in some cases it may reduce it. An example from this is provided by Figure 2(b) and 3: Both trees are from the same system, but operation B.b was not instrumented during the monitoring for Figure 2(b). Trace analysis would identify 8 caller-contexts for the system corresponding to Figure 2(b), while 7 caller-contexts are identified when B.b is instrumented.

The fact that an instrumentation with n monitoring points has m calling-contexts does not imply that a second instrumentation with $n' > n$ monitoring points has more than m calling-contexts (in the same software system and for the same workload), since different monitoring points can increase the same numbers of calling-contexts differently.

IV. CASE STUDY

The case study evaluates how many calling-contexts exist in a typical software application, and how much variance of the operation response time distributions results from not distinguishing calling-contexts. Additionally, it is explored how the selection of monitoring points relates to the number of calling-contexts. The software system analyzed in the case study is the iBATIS JPetStore[1], which is a demo Java Web-application implementing an online store scenario. The instrumentation to monitor response times of the internal operations of the JPetStore is given by the software instrumentation package Kieker [15].

A. Evaluation Goals

The goals of the case study are (1) to explore the relation between the number of monitoring points and the number of resulting calling-contexts, (2) to support the hypothesis that multi-modality can arise from operations with different calling-context specific timing behavior, and (3) to compare the calling-context specific response time distributions to the response time distributions without calling-context analysis.

B. Setting

The software application analyzed in the case study is the iBATIS JPetStore 5 Web-application. It is deployed in the Apache Tomcat Servlet container (version 5.5.23) running on a desktop computer equipped with an Intel Pentium 4 3.00 GHz hyper-threaded CPU and 1 GB physical memory and Linux 2.6.17.13. The application server software employs Sun Java SE 1.6.0_03. JPetStore uses a database management system (MySQL 5.0.18) for storing business data running on a GNU/Linux 2.6.15 system with two Intel Xeon 3.00 GHz CPUs and 2 GB of physical memory. The application server and the database backend are connected via 100 Mbit Ethernet. A workload generator runs on a separate desktop computer being identically equipped and configured as the application server node above.

The workload for the JPetStore is generated by the workload driver Apache JMeter 2.2 extended by our probabilistic workload driver Markov4JMeter[2]. Markov4JMeter allows to emulate users based on an application model and a mix of corresponding probabilistic user behavior models. The think time between user requests is configured to be normally distributed. The experiment runs last 20 minutes; the first 3 minutes are considered the warm-up period and are ignored in the evaluation. The number of concurrent users is set to 10, which can be handled without any problems by the system under monitoring.

Three instrumentation scenarios are used:

- E1 Partial instrumentation: 18 manually selected monitoring points from a previous case study [16] on scalability (response times vs. workload intensity)
- E2 Full instrumentation: All operations and application entry points are monitored resulting in 199 different instrumented operations
- E3 Random instrumentation: 1500 random instrumentations having 2 to 198 monitoring points. The traces for these 1500 instrumentations are generated from the monitoring run of the full instrumentation by ignoring random subsets of monitoring points.

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

[2] http://markov4jmeter.sourceforge.net/

TABLE I
SUMMARY OF THE INSTRUMENTATION SCENARIOS AND NUMBERS OF DISTINCT CALLING-CONTEXTS.

Instrumentation	Partial (E1)	Full (E2)	Random (E3)
# Instrumented operations	18	199	2–198
# Monitored executions	121.323	2.032.573	2–2.032.572
# Traces	36.190	36.036	1–36.036
# Caller-contexts	20	290	2–312
# Stack-contexts	21	368	2–368
# Trace-contexts	31	7021	2–7021

C. Results

a) Total number of calling-contexts in relation to the number of monitoring points: As shown in Table I, 31 trace-contexts exist for the partial instrumentation with 18 monitoring points. 7021 trace-contexts exist in total for the full instrumentation scenario using 199 monitoring points.

Table I indicates that the number of trace-contexts, caller-contexts and stack-contexts grows by the number of monitoring points, and that the number of trace-contexts increases faster with the number of monitoring points than both other calling-contexts.

Figure 4 shows in more detail that the number of trace-contexts can be relatively high for a significant part of the operations: 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.

Instrumentation scenario E3 allows to study the relation between the number of monitoring points and calling-contexts for the software system in general. The number of resulting calling-contexts from 1500 random instrumentation scenarios is illustrated in Figure 5 in dependence to the number of monitoring points. In the sample application, the number of stack-contexts and caller-contexts both grow linearly with a similar rate by the number of monitoring points, as shown in Figure 5(a). For 81.7% of the evaluated instrumentation scenarios, the number of stack-contexts was larger than the number of caller-contexts for the same instrumentation.

In contrast to the number of distinct stack and trace-contexts, the number of caller-contexts is not at its maximum for full instrumentation. This demonstrates that adding monitoring points can reduce the number of caller-contexts.

Figure 5(b) visualizes the numbers of trace-contexts resulting from the 1500 random instrumentation scenarios. The

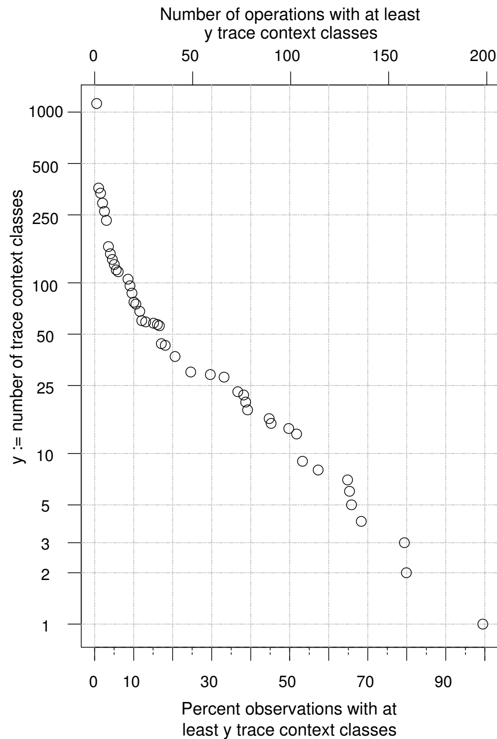


Fig. 4. Full instrumentation scenario: A significant number of observations each has a high number of trace-contexts. The diagram shows how many operations have more than a particular number of trace-contexts y. For example, 50% of the operations each has at least 13 trace-contexts.

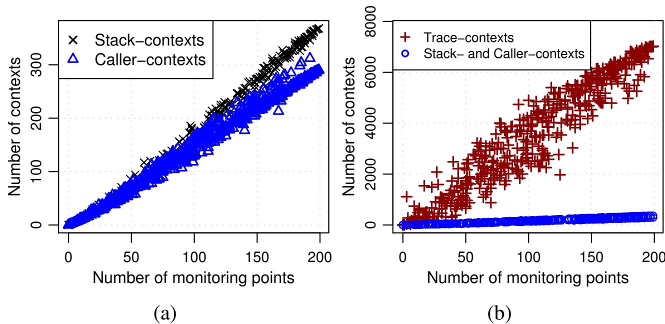


Fig. 5. The number of monitoring points in relation to the number of contexts.

number of trace-contexts increases much faster than the number of stack-contexts and caller-contexts does.

b) Number of calling-contexts per operation: For instrumentation E1, most operations (12 of 18) had only one trace-context, six had two trace-contexts, and one operation had seven trace-contexts. This results in an average number of 1.6 trace-contexts per operation. For instrumentation E2, the number of trace-contexts per observation varies between one trace-context (39 of 199 operations) and 1123 trace-contexts for one operation.

c) Number of response times per calling-context: 1344 of the 7200 trace-contexts 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. Statistical analyzes, such as the determination of standard deviation and the approximation of density distributions, may not produce meaningful results for calling-contexts with an unsufficient number of observations. . A possible solution to this problem is to characterize the timing behavior of such a calling-context by all response times of the corresponding operation. This means that in this case calling-context information is omitted.

d) Multi-modal distributions in calling-context analysis: For the instrumentation scenario E1, the operation `newOrder()` follows the multi-modal response time distribution shown in Figure 6(a). Trace analysis reveals that two trace-context, one stack-context, and one caller-context exist for this operation.

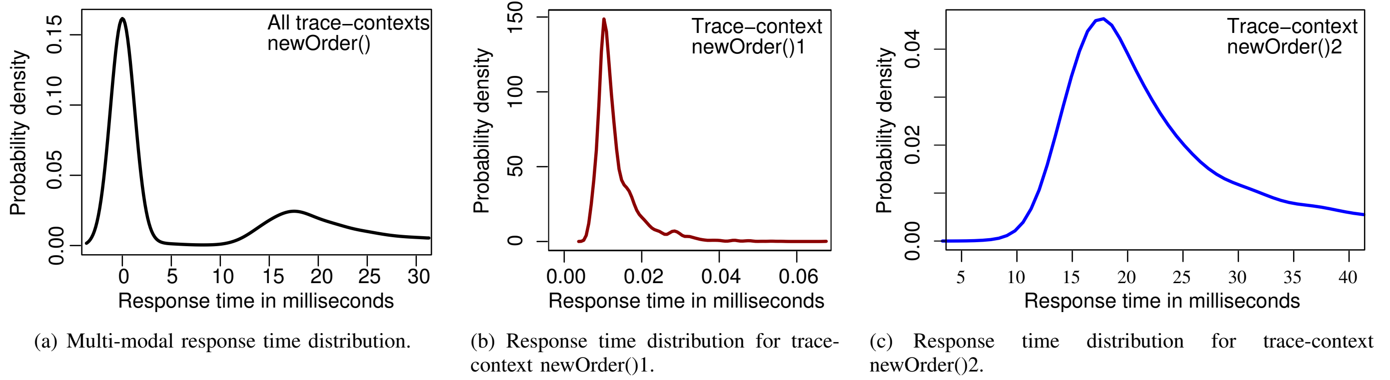
Trace-context analysis distinguishes two cases: In the first case, `newOrder(..)` does not make any monitored sub-calls to other operations, as shown in the Sequence Diagram in Figure 6(b). In the second case, there is a sub-call `insertOrder(..)`, as illustrated in Figure 6(c).

Only trace-context analysis allows one to differentiate the response times of this operation into two equivalence classes. The response times for each trace-context are displayed in Figure 7(a) and its corresponding distributions are shown in Figure 6(b) and (c). In this example, caller-context equivalence and stack-context equivalence are unable to distinguish both calling-contexts, since caller- and stack-context information do ignore sub-calls.

e) Response time distribution variance related to calling-context information: We use the standard deviation as the metric to study the impact of calling-context analysis to the response time distributions. The standard deviation is a common metric to characterize the dispersion of data and to quantify the (root mean square) error in the context of prediction or estimation. The sample variance is the square-root of the statistical variance; therefore its values are in the scale of the measurements (i.e., milliseconds in the case study). To compare the calling-context analyses to using no calling-context information, the standard deviation of all calling-contexts that correspond to an operation has to be aggregated. For this, we computed the average standard deviation of all calling-contexts for each operation, multiplied by their relative frequency in the monitoring data.

To quantify the benefit from calling-context analysis, the average standard deviation of using no calling-context information is compared to the average standard deviations from the calling-context sensitive response time distributions. Table II, shows the percentual decrease in average standard deviation from using calling-context analysis.

For E1, the decrease of standard deviation is only small in average for all calling-context types. This may be surprising, since the calling-context analysis was very effective for operation `newOrder()` (see Figure 6). In fact, the decrease of the standard deviation for this operation is at 38.73%, but this operation has a low relative call frequency (1,748 of 121,323). The boxplot in Figure 8(a) shows of 38.73% is only an exceptionally high decrease. Most operations show only low or no benefits from calling-context analysis for this

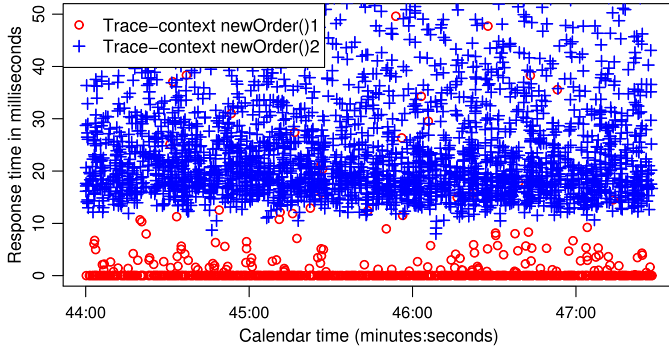


(a) Multi-modal response time distribution.

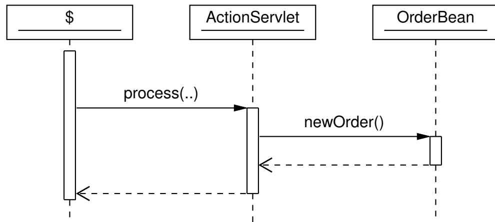
(b) Response time distribution for trace-context newOrder()1.

(c) Response time distribution for trace-context newOrder()2.

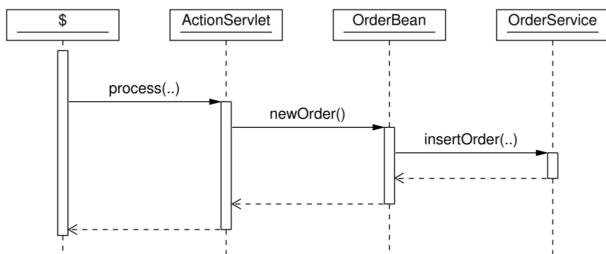
Fig. 6. Density distribution for (a) all response times of operation `newOrder()`, and (b,c) for each of the two trace-contexts.



(a)



(b) Trace corresponding to trace-context newOrder()1

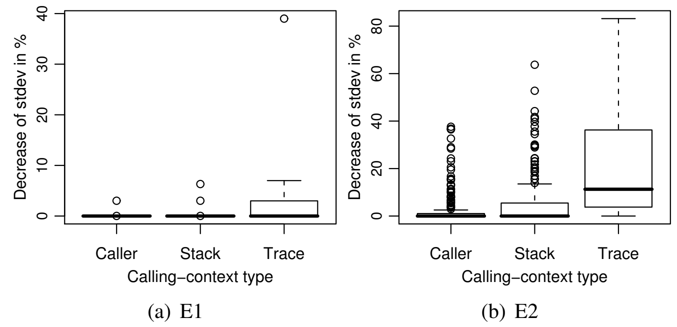


(c) Trace corresponding to trace-context newOrder()2

Fig. 7. Trace-context dependent timing behavior; instrumentation scenario E1: Response times for operation `newOrder()` show different distributions depending on whether corresponding to the trace-context(b) newOrder()1 or (c) newOrder()2.

TABLE II
STANDARD DEVIATION RELATED TO CALLING-CONTEXT ANALYSIS.

	Average st.dev. decrease in %	
	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



(a) E1

(b) E2

Fig. 8. Average decrease in standard deviation from using calling-context information compared to standard deviation using no calling-context information. Standard deviation decreases are boxplotted per operation.

instrumentation.

The data for instrumentation scenario E2, shows a much different picture than the data of E1: Trace-context analysis leads to 42.2% lower average standard deviation in the trace-context sensitive response time distributions than using no calling-context information. Trace-context analysis also clearly shows better results than stack-context analysis and caller-context analysis. The boxplot in Figure 8(b) indicates that for most instrumentations, trace-context analysis results in reduced standard deviation for more than 75% of the operations. In some cases the reduction was even above 80%.

V. Discussion and Limitations

A. Monitoring overhead

As discussed in the introduction, typical application scenarios are runtime performance management and failure diagnosis based on anomaly detection, such as the work by Duzbayev and Poernomo [9], Diaconescu et al. [8], Agarwal et al. [1]. This requires continuous monitoring during regular operation

of the software system. Therefore, the monitoring overhead should be reasonably low, for instance by imposing less than 15% overhead on response times and throughput.

A detailed discussion on monitoring overhead is beyond the scope of this paper. However, in the case studies we use our instrumentation tools Kieker[3]. The overhead of Kieker was estimated to be below 20% overhead on response times in the case study with full instrumentation. An overhead of about 10% was reported for the monitoring framework InfraRED [10], which uses similar aspect-oriented programming technology as Kieker does.

B. Distributed systems

Our monitoring infrastructure does not allow us to track execution traces through multiple execution environments. Therefore, a single execution path may be split into multiple execution traces, which results in a performance model that cannot address correlations between timing behavior and the execution traces characteristics that are not within the same environment. This limitation can be overcome by using an alternative monitoring approach, such as Briand et al. [6], that allows to monitor remote communication.

C. Representativeness and completeness of monitoring data

In our approach, timing behavior distributions and calling-contexts result from monitoring data. This results in the two major risks that the monitoring data is not representative for normal behavior and that not all calling-contexts are detected. For instance, calling-contexts are missing if possible execution sequences were not activated during the monitoring period, which depends on the system usage. These risks can be minimized by using a sufficient amount of monitoring data from real system usage. For instance, for a typical online store, we consider a few weeks of monitoring data to be sufficient for timing behavior anomaly detection. For the identification of calling-contexts, static (source code) analysis provides an alternative to monitoring data analysis, since it doesn't depend on system workload. Since the performance behavior of a software system changes over time (e.g., improving of algorithms, changes in user behavior, changes in hardware), it may be required to update software performance models regularly.

VI. Related Work

Related work comes from the domains of profiling, performance evaluation, online failure diagnosis, and performance prediction.

Some literature in the domain of software profiling addresses to connect response time behavior to method calls and context information. Graham et al. [11] introduced the profiler *gprof*. Gprof provides caller-context information (i.e., it makes caller-callee relations explicit). The trace-context concept, introduced in this paper, is an extension of the concept of caller-contexts. Most modern profiling tools, such as Intel's VTune Performance Analyzer follow gprof by providing caller-context information (see Xie and Notkin [17]). The work of Ammons et al. [2] goes beyond the caller-callee relationship and introduces what we denoted stack-context equivalence. We extend the stack-context by using the complete sequence of operations as criteria for equivalence. These authors do not discuss the timing behavior distributions resulting from calling-context analysis, which is a major focus of our paper.

In the timing behavior evaluation of Bulej et al. [7], clusters in timing behavior of a software system have been observed and analyzed. The authors report timing behavior clusters for a CORBA implementation in the context of comparing timing behavior of different software versions. The k-means clustering approach is used to identify clusters in timing behavior measurements. In contrast to our approach, this does not require trace data, therefore, the requirements on the monitoring infrastructure are lower than in our approach. However, our approach uses additional relevant information, which allows the precise distinction of timing behavior classes, while the k-means clustering approach is a heuristic that only performs well if the correct number of clusters is known in advance and the values within the clusters are well separated.

Various approaches have been presented to use timing behavior monitoring data of software systems in order to implement preemptive quality of service management. For instance, in the Magpie project [4] it has been motivated to correlate monitored events for specific requests to timing behavior measurements to identify anomalies and perform failure diagnosis. The Magpie approach shares the general idea of correlating monitored events within a request to timing behavior with our approach, but details on the correlation or empirical data have not been presented, so far.

The performance modeling approach of Koziolek et al. [13] considers parameter values as part of usage profiles in order to increase performance prediction precision. Parameter values can also be considered calling-context information. However, the three calling-context types described in this paper are not part of Koziolek et al.'s software performance model. A difference of this and other performance prediction approaches is the focus on the early design period, while our focus is on the evaluation of detailed monitoring data that is usually only available for deployed software systems.

VII. Conclusions

A. Summary

In this paper, we presented our approach to analyzing operation response time measurements in dependence to their calling-contexts. We introduced trace-context equivalence, which extends the concepts of caller-contexts and stack-contexts.

The case study using a Java Web-application, evaluated how much variance in software response time distributions can be associated to calling-context information. Furthermore, we evaluated how the number of different calling-contexts relates to the number of monitoring points.

The evaluation results show that using calling-context information produces response time distributions with significantly

[3]http://kieker.sourceforge.org

lower variance. The response time distributions based on trace-context analysis had much lower variance in average than those from stack-context analysis or caller-context analysis. Some operations in the evaluation system show a large number of trace-context dependent response time distributions.

Additionally, it was demonstrated that it was possible to replace a multi-modal response time distribution by two uni-modal response time distributions using trace-context information. Both high variance and multi-modality are threats to many timing behavior analysis approaches.

B. Future work

Currently, the approach requires the complete trace to be recorded and evaluated before the corresponding trace-context specific response time distribution can be identified. This "postdiction" can be used in failure diagnosis approaches, such as anomaly detection. However, for other performance evaluations it can be desirable to predict the expected response time for an operation before its execution. In that case, only a part of the full trace, e.g., the prefix, could be used. This information could be valuable to coordinate execution environment elements, such as the scheduler or the garbage collection.

A major design decision for monitoring is the selection of monitoring points. For field systems, a trade-off between monitoring overhead, and monitoring granularity has to be satisfied. The full instrumentation of all operations is usually computationally too expensive. Thus, monitoring points have to be selected. Trace-context analysis could be used to determine additional monitoring points for manually-defined set of monitoring points. From data of a fully instrumented execution period, it is possible to estimate, which additional monitoring point would be optimal for trace-context analysis.

Other next steps are to evaluate trace-context analysis in large industry systems, and to evaluate its efficiency. The efficiency evaluation would relate the benefits of trace-context analysis to the costs imposed by monitoring.

References

[1] Manoj K. Agarwal, Karen Appleby, Manish Gupta, Gautam Kar, Anindya Neogi, and Anca Sailer. Problem determination using dependency graphs and run-time behavior models. In *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, 2004. doi: 10.1007/b102082.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the Conference on Programming language design and implementation (PLDI'97)*, pages 85–96. ACM, 1997. doi: 10.1145/258915.258924.

[3] Martin F. Arlitt, Diwakar Krishnamurthy, and Jerry Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology*, 1(1):44–69, 2001. doi: 10.1145/383034.383036.

[4] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HOTOS'03)*, pages 15–15. USENIX Association, 2003.

[5] 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, October 2006.

[6] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, September 2006. doi: 10.1109/TSE.2006.96.

[7] Lubomír Bulej, Tomáš Kalibera, and Petr Tůma. Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 60(1-4):345–358, 2005. doi: 10.1016/j.peva.2004.10.013.

[8] Ada Diaconescu, Adrian Mos, and John Murphy. Automatic performance management in component based software systems. In *Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, pages 214–221. IEEE, 2004. doi: 10.1109/ICAC.2004.15.

[9] Nurzhan Duzbayev and Iman Poernomo. Runtime prediction of queued behaviour. In Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Proceedings of the 2nd International Conference on the Quality of Software Architectures (QoSA'06)*, volume 4214 of *Lecture Notes in Computer Science*, pages 78–94. Springer, 2006. doi: 10.1007/11921998_10.

[10] Kamal Govindraj, Srinivasa Narayanan, Binil Thomas, Prashant Nair, and Subin P. On using AOP for Application Performance Management. In Matt Chapman, Alexandre Vasseur, and Günter Kniesel, editors, *AOSD 2006 - Industry Track Proceedings (Technical Report IAI-TR-2006-3, University of Bonn)*, pages 18–30, March 2006.

[11] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Notes*, volume 17, pages 120–126. ACM, 1982. doi: 10.1145/872726.806987.

[12] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, first edition, April 1991.

[13] Heiko Koziolek, Steffen Becker, and Jens Happe. Predicting the Performance of Component-based Software Architectures with different Usage Profiles. In *Proc. 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.

[14] Object Management Group (OMG). Unified Modeling Language: Superstructure Version 2.1.1, February 2007.

[15] Matthias Rohr, André van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke, and Wilhelm 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, February 2008.

[16] André van Hoorn. Workload sensitive timing behavior anomaly detection in large software systems. Master's thesis, (Diplomarbeit) University of Oldenburg, Software Engineering Group, Department of Computing Science, 2007.

[17] Tao Xie and David 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, December 2002.