Workload-Intensity-Sensitive Timing Behavior Analysis for Distributed Multi-User Software Systems*

Matthias Rohr[1,2], André van Hoorn[2], Wilhelm Hasselbring[2,3],
Marco Lübcke[4], Sergej Alekseev[5,6]
[1] BTC Business Technology Consulting AG, Germany
[2] Graduate School TrustSoft, University of Oldenburg, Germany
[3] Software Engineering Group, University of Kiel, Germany
[4] CeWe Color AG & Co. OHG, Oldenburg, Germany
[5] Nokia Siemens Networks GmbH, Berlin, Germany
[6] Hochschule Mittweida, University of Applied Sciences, Mittweida, Germany

ABSTRACT

In many multi-user software systems, such as online shopping systems, varying workload intensity causes high statistical variance in timing behavior distributions. However, this major impact on timing behavior is often ignored.

This paper introduces our approach WITiBA (Workload-Intensity-Sensitive Timing Behavior Analysis) to consider inter-dependencies between concurrent executions of software operations within a distributed system to reduce the standard deviation for succeeding analysis steps. This can be beneficial for analysis methods or simulation methods in terms of tighter confidence intervals, or shorter simulations.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Performance, Measurement

Keywords

software performance, profiling, workload intensity, scalability, monitoring, concurrency, response time distribution

1. INTRODUCTION

Response time distributions of operations in enterprise software systems often show high variance (see e.g., [7]). High variance can make it more difficult to draw statistical conclusions [6].

Our hypothesis is that varying workload intensity (e.g., from concurrent user activity) is a major reason for high variance in distributed multi-user systems, such as online stores, and online banking sites, and that typical evaluations of monitoring data can be significantly improved by explicitly considering workload intensity. This paper introduces the novel statistical method WITiBA (Workload-Intensity-Sensitive Timing Behavior Analysis) for evaluating software timing behavior in the context of the workload intensity that is present at the time of the execution.

WITiBA provides a workload-intensity-specific timing behavior model with significantly lower variance by defining classes based on workload intensity. This follows the general idea of Menasce and Almeida [6], with the difference that they grouped similar requests in terms of performance resource usage to reduce variability.

Many evaluation and simulation techniques can benefit from our variance reduction approach in terms of requiring less observations, providing tighter confidence intervals, or requiring less or shorter simulation runs [4]. Empirical results from field and lab studies demonstrate the benefits of our approach and quantify the variation related to varying workload intensity.

This paper is structured as follows. Sections 2 and 3 introduce the system model and our approach to timing behavior analysis. The case studies are presented in Section 4, before related work and the conclusions follow in Sections 5 and 6.

2. BACKGROUND

This section describes the system model underlying our approach, foundations on timing behavior, and variance reduction.

2.1 System Model

It is assumed that software systems are (or can be) hierarchically structured into entities of the following types: *operations*, *components*, and *execution environments*. Components provide operations that might be requested by other components, external users, or systems. Components are deployed to a number of possibly distributed, inter-connected execution environments. An execution environment comprises the physical hardware platform including the computational resources, the operating system, as well as the component container, e.g., a Java EE application server.

Primary artifacts of system runtime behavior are *executions* of software operations. A *trace* represents the internal control-flow among executions originating from an external

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

WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.
Copyright 2009 ACM 978-1-60558-563-5/10/01 ...$10.00.

service request. We limit the scope to synchronous communication, i.e., a trace is the result of a sequence of executions serviced by a single thread of control. This implies that execution sequences that are connected by asynchronous communication are represented as separate traces.

2.2 Timing Behavior of Software Systems

The execution-related timing metrics used in our approach are *response time* and *execution time*. The response time of an execution is defined as the time period between the start and end of an execution including the response time of nested executions. The execution time excludes the response of nested executions. e with $op(e) = o$ represents an execution of operation o. The start time, response time, and execution time of e are denoted $st(e)$, $rt(e)$, and $et(t)$.

Due to a finite number of shared hardware and software resources, the timing behavior of a software system is influenced by the workload it is exposed to. Workload can be divided into *workload intensity* and *individual request characteristics* (cp. [6]). Typical workload intensity metrics are arrival rates and the number of jobs in a system. Individual request characteristics are the properties of individual requests, e.g., in terms of request types and parameter values.

Response times tend to increase non-linearly by workload intensity [4]: Up to a first workload intensity level the response time does not increase significantly. Starting with a second level, the response time increases approximately linearly, before it tends to increase rapidly by increasing workload intensity due to resource contention.

2.3 Standard Deviation Reduction

The benefit of our approach will be quantified by the reduction of standard deviation (in percent) in relation to the original dataset. First, it is computed for each operation how much the standard deviation for all observations is reduced compared to the average (weighted by the class size) standard deviation for each workload-intensity-specific sub-class (short: *standard deviation reduction*). In a second step, the relative standard deviation values of all instrumented operations of the software are aggregated (weighted by number of observation per operation) into a single value for each variant of our approach. To achieve statistically robust evaluation results, operations with less that 600 observations were accounted with a standard deviation reduction of 0%.

3. OUR APPROACH TO TIMING BEHAVIOR ANALYSIS

This section presents our new approach, called WITiBA (Workload-Intensity-Sensitive Timing Behavior Analysis), to consider workload intensity in software timing behavior analysis. WITiBA's idea is to group timing behavior corresponding to different workload intensity scenarios. Each class represents the timing behavior observations for a workload intensity interval. Figure 1 illustrates this for the three classes of low, medium, and high workload intensity. As indicated by the three probability density functions, considering workload intensity can reveal workload-intensity-specific timing behavior. An approach splitting the observation set into classes, such as ours, requires a suitably high number of observations.

The key element of our approach is a workload intensity metric, denoted *pwi* (Platform Workload Intensity). This paper introduces four alternative 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, pwi_2 uses execution times instead of response times, pwi_3 is extended for distributed systems, and pwi_4 uses weights in order to take into account that different software operation in a distributed system can have different timing behavior influences to each other. pwi_4 is the average weighted sum of all concurrently executing operations over a time period within the same execution environment. Table 1 summarizes the characteristics of the metrics in terms of whether they use response times or execution times; whether the structure of distributed systems is considered; and whether concurrent executions of other operations are all equally weighted.

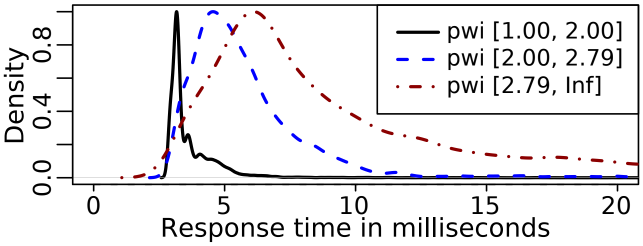


Figure 1: Example: Probability distributions for low, medium, and high workload intensity of operation `getItemListByProduct(..)` of case study CS-1.

Table 1: *pwi* metrics overview.

Metric	Time metric	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

All metrics use basic control-flow information. Therefore, the monitoring of the metrics can be implemented efficiently, which makes them suitable for continuous operation in real world systems.

3.1 Platform Workload Intensity *pwi₁*

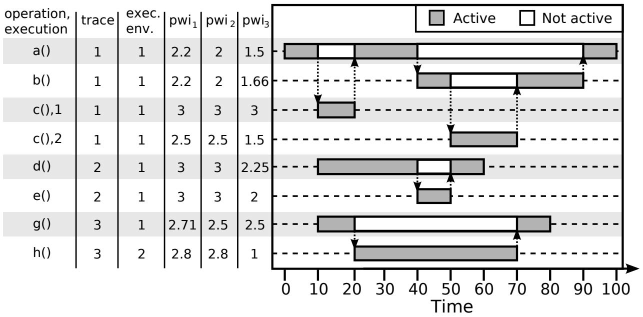
The pwi_1 metric is defined as the average number of all concurrent traces during the time period between the start (call action) and the end of an operation execution. More precisely, for an execution e with $st(e), rt(e) \in \mathbb{N}$, the platform workload intensity function $pwi_1 : e \rightarrow [1, \infty] \subset \mathbb{R}$ is defined as follows:

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

where $|AT(t)|$ is the cardinality of a set $AT(t)$. Time is assumed to be discrete, therefore, $t \in \mathbb{N}$. $AT(t)$ is defined

$$AT(t) := \{tr \mid \exists e' \in tr : t \in [st(e'), st(e') + rt(e')]\}. \quad (2)$$

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 at time t. Hence, pwi_1 provides the average number of traces executing during the execution time period of e. The values of pwi_1 start at 1 since an



operation, execution	trace	exec. env.	pwi$_1$	pwi$_2$	pwi$_3$
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 2: *pwi* computation examples.

execution has always its own trace in the set AT. Examples for the computation of pwi_1 can be found in Figure 2. The gray shading indicates which executions are active.

3.2 Platform Workload Intensity pwi_2

pwi_2 differs to pwi_1 by considering the execution time period instead of the response time period. Timing behavior analysis using the execution time period is usually more precise, i.e., only the time period is considered in which an operation execution is active and does not wait for the result of sub-calls.

pwi_2 uses the execution time $et(e)$, based on the assumption that an operation execution that is waiting for a sub-call does not compete for resources during the waiting period. The time periods in which the execution e to be evaluated is waiting for the results of sub-calls are ignored. This extends Equation 1 to

$$pwi_2(e) := \frac{1}{et(e)} \sum_{t=st(e)}^{st(e)+rt(e)} |AT(t)| \cdot Active(e,t) \qquad (3)$$

with $Active \rightarrow \{0,1\}$ and $Active(e,t) = 0$ if the execution e waits at time t for a sub-call to complete and 1 else.

Examples for the computation of pwi_2 are in Figure 2.

3.3 Platform Workload Intensity pwi_3

The following pwi metrics consider the structure of a distributed system. pwi_3 and pwi_4 assume that the active executions within the same execution environment compete for the same resources. For instance, a CPU is only typically provided to local operation executions by the local execution environment. However, this assumption may not hold for resources that are shared over the network.

A trace can only be active in one of the execution environments a time, although it may contain executions spanning more than one execution environment. Therefore, computing an execution's pwi should include only the activity within its corresponding execution environment, while activities in other execution environments must be ignored since these do not directly compete for the same resources.

Mathematically, this extends Equations 3 and 2 to

$$pwi_3(e) := \frac{1}{et(e)} \sum_{t=st(e)}^{st(e)+rt(e)} |AT(t,e)| \cdot Active(e,t) \qquad (4)$$

with

$$\begin{aligned} AT(t,e) := \{tr \mid \exists e' \in tr : Active(e',t) = 1 \\ \wedge \;\; env(e') = env(e)\} \end{aligned} \qquad (5)$$

where $env(e)$ provides the execution environment in which an execution e executes. In words, $AT(t,e)$ is the number of traces having an execution e' that executes in the execution environment $env(e)$ at time t without waiting for sub-calls. Examples for the computation of pwi_3 can be found in Figure 2.

3.4 Platform Workload Intensity pwi_4

pwi_1–pwi_3 model each operation execution with equal resource demands. pwi_4 uses weights to consider different resource demands for each operation. For this, W is defined the weight matrix, where $w_{o,p} \in \mathbb{R}$ is the weight to be used within an execution of operation o for considering concurrent executions of operation p. A relatively high value of $w_{o,p}$ indicates that executions of p have a strong influence (e.g., because of resource sharing) to an execution of o.

The pwi_4 is computed by aggregating and weighting operation-specific pwi values as defined by Equations 6 and 7. Let e be an execution of operation o with its execution time et, then $pwi_4(e)$ is defined as follows:

$$pwi_4(e) := \frac{1}{et(e)} \cdot \sum_{p=1}^{m} w_{o,p} \cdot pwi_4(e,p) \qquad (6)$$

$$pwi_4(e,p) := \sum_{t=st(e)}^{st(e)+rt(e)} |AT(t,e,p)| \cdot Active(e,t). \qquad (7)$$

$pwi_4(e,p)$, defined in Equation 7, extends Equation 4 by a reference to the operation p. $AT(t,e,p)$ is the set of traces having an execution of operation p that is active at time t in the execution environment $env(e)$:

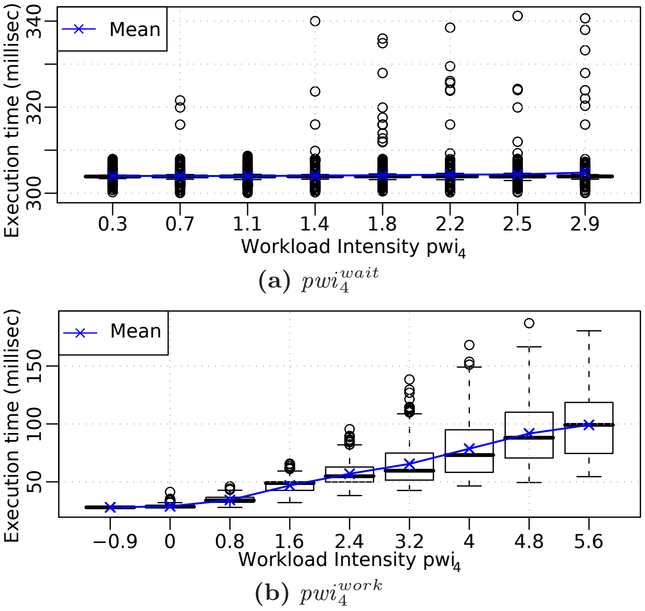
$$\begin{aligned} AT(t,e,p) := \{tr \mid \exists e' \in tr : Active(e',t) = 1 \;\wedge \\ env(e') = env(e) \;\wedge \\ op(e') = p\}. \end{aligned} \qquad (8)$$

In order to use pwi_4, the weight matrix W has to be provided. The determination of the weight matrix is an n dimensional optimization problem, with n as the number of operations in the system, but the relevant search space is usually much smaller, since only the number of operations in the same execution environment are considered according to Equation 8.

W can be determined from historical monitoring data by either using machine learning algorithms or by determining a solution analytically. The computational costs can be high for systems in which a large number of operations is instrumented within the same execution environment and a suitably large amount of observations is required. In the case studies, a machine learning technique (gradient descent) iteratively refines the weights randomly to determine those that provide the highest variance reduction in average. Heuristics, such as the method used, do usually not provide an optimal solution, but are often more efficient in providing a reasonably good solution than analytical methods. A first a priori estimate for the weight $w_{o,p}$ is given by the correlation coefficient between the series of response times

Table 2: Example: Weight vectors (columns).

	W^{work}	W^{wait}
work	2.01	1.03
wait	-0.05	0.05



(a) pwi_4^{wait}

(b) pwi_4^{work}

Figure 3: Example boxplots: Relation between pwi_4 and execution times.

for operation o and the series of corresponding $pwi_4(e, p)$ values.

When the pwi_4 weights are determined by machine learning techniques, there is the danger of so-called overfitting. Overfitting denotes the case when a model fits well in the context of the training data but fails for data sets that were not used in training. Overfitting was prevented with so-called early stopping, which stops the training of a model parameter when it would reduce the performance in the context of a second data set. Standard deviation reduction benefits were only quantified with an independent third dataset.

Example pwi_4. Let a program consist of the two operations *wait* and *work*. Operation *wait* waits for some time without requiring any resources during that period and *work* performs CPU intensive computations. It is to expect 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*.

A straight forward Java implementation of this program was executed 120,000 times under various workloads, with 1-20 concurrent executions. The pwi_4 computation is parameterized with the weight vectors of Table 2, which have been learned from training data.

Figure 3(b) shows a strong relation between execution times of *work* and pwi_4: both execution time mean and variance grow by increasing pwi_4 values. Most monitored execution times of operation *wait* are relatively independent from the corresponding pwi_4 values, as displayed in Figure 3(a). Table 3 provides the resulting standard deviation reduction (see Section 2.3), showing that pwi_4 performs best.

Table 3: Example: Standard dev. reduction (%).

	$pwi_1 = pwi_2 = pwi_3$	pwi_4^{work}	pwi_4^{wait}	pwi_4
work	22.1 ± 2	72.0 ± 2	7.6 ± 1	72.5 ± 2
wait	16.3 ± 8	18.3 ± 9	11.7 ± 6	18.8 ± 9

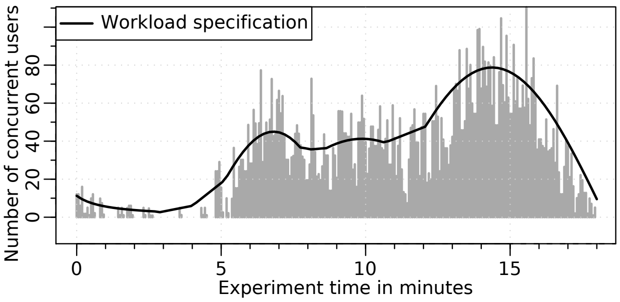


Figure 4: CS-1: Workload intensity specification.

4. CASE STUDIES

In this section, empirical results from lab studies and a field study are presented. As in the previous section, the evaluation metric is the average relative *standard deviation reduction* (see Section 2.3). Table 4 summarizes the case studies. CS-0 denotes the example of the previous section. Based on experience in first experiments, the number of overlapping bins is set to 10.

4.1 Distributed Web Shop (CS-1)

The software application analyzed in the case study is a re-engineered distributed version of the iBATIS JPetStore 5 which represents an online shopping store. We divided the JPetStore into four software components, each deployed to a dedicated machine and a databaseon a fifth machine.

The workload is generated by the workload driver Apache JMeter extended by Markov4JMeter [10][1]. The workload has two major characteristics: 1) the user behavior model is probabilistic, specified by a Markov model, and 2) the varying workload intensity is specified based on a curve based on 24 hour monitoring data of a real system (Figure 4). The curve is scaled to a maximum of 78 concurrent users to stay below 80% utilization of the maximal capacity.

The experiment scenario was executed 5 times with system executions of 18 minutes each. Each run provided about 740,000 executions from 34 probes in the system. Figure 5 shows the standard deviation reduction results. All four methods strongly reduce standard deviation – in average from 35% for pwi_1 up to 56% for pwi_4. Log-transforming the pwi values, before defining bins additionally improves standard deviation reduction by 29% in average. For pwi_4, this results in a standard deviation reduction of 65%.

4.2 Telecommunication System (CS-2)

CS-2 analyzes monitoring data from a telecommunication signaling system 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.

Figure 6 displays the workload specification (BHCA = Busy Hour Call Attempts). This is a low workload intensity scenario, since the CPU utilization does not exceed 20%,

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

Case study	Study type	Workload		% CPU utilization	# operations instrumented	# execution environments monitored	System type
		Intensity	User behavior				
0	Example	Linearly incr.	Constant	10 – 80%	2	1	Two method example
1	Lab	Adapted from real system	Markov Model	0 – 80%	34	5	Distributed Web shop
2	Lab	Test scenarios	Non-probabilistic scenarios	0 – 20%	8	2	Telecommunication signaling system
3	Field	Real	Real	0 – 15%	161	1	Photo shopping and service portal

Table 4: Case Studies.

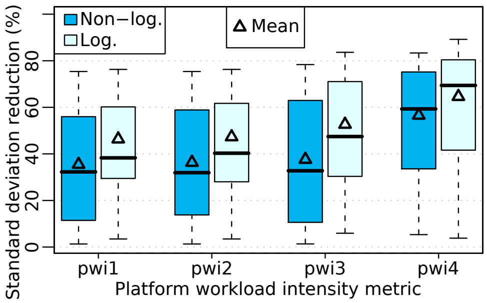


Figure 5: CS-1: Standard deviation reduction.

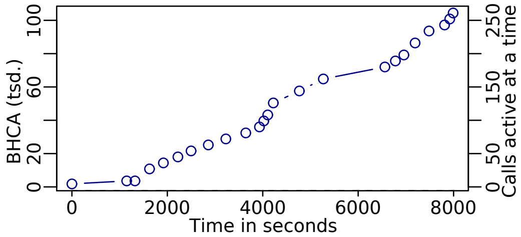


Figure 6: CS-2: Workload specification.

which provides less potential for benefits from considering workload intensity.

The monitoring data consists of 2.5 million executions for eight software operations. Two load-balanced identical execution environments are monitored.

pwi_2 and pwi_3 are equal, since the traces monitored in CS-2 never span over multiple execution environments.

Figure 8(a) shows the standard deviation reduction results for CS-2. As in CS-1, pwi_4 performs best in the comparison of the four alternative methods. The average standard deviation is additionally improved by more than 40% if the logarithm of the pwi values are used for defining timing behavior classes. Prior log-transformation of the pwi_4 results in 32.34% standard deviation reduction.

4.3 Photo Shopping and Service Portal (CS-3)

The monitoring data in CS-3, is from a portal of CeWe Color AG, Europe's largest digital photo service provider. Customers use the portal to order photo prints and other photo products.

The monitoring data consists of about 1.5 million observations for each day. A large number of software operations (161) has been instrumented in a single execution environment of the load-balanced production environment. The workload (shown in Figure 7) is the real system usage monitored on a single of many load-balanced execution environments. The 100% line indicates the average workload during Friday's daytime. The complete monitoring data from the first three days were used in this paper.

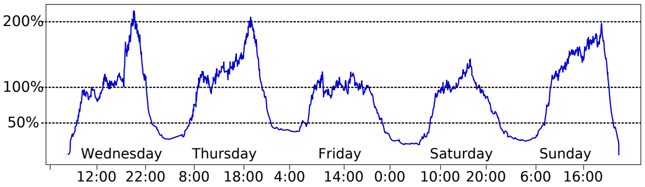
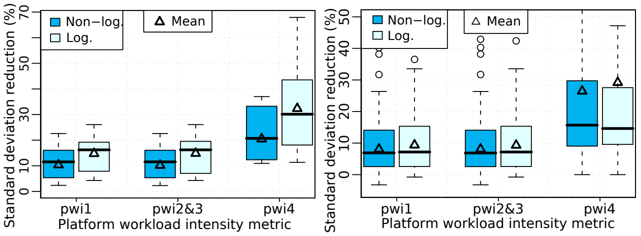


Figure 7: Workload curve of CS-3 (active sessions).



(a) CS-2

(b) CS-3

Figure 8: Standard deviation reduction.

The system was not under high load on the days of monitoring. The CPU utilization did not exceed 15% for executing the operation executions. Therefore, smaller benefits from considering workload intensity in performance analysis can be expected than in the case studies with higher workload intensity.

Figure 8(b) shows the standard deviation reduction results for CS-3. Again, pwi_4 performs best in the comparison of the four alternative methods (26.46%, 29.15% for log.). As only a single execution environment was monitored, pwi_2 equals to pwi_3. Several of the 161 instrumented operations were only executed less times than required for statistically robust computations. For these operations, a standard deviation reduction of 0% was assumed in computing the average relative standard deviation reduction.

4.4 Summary

The three case studies indicate 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 results in CS-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 the non-distributed case studies CS-2 and CS-3, pwi_3 is equal to pwi_2.

pwi_4 performs best in all case studies. This indicates that it is beneficial to model the timing behavior influence of concurrent executions for each software operation individually.

5. RELATED WORK

Our former work [8] also addressed the reduction of standard deviation in software timing behavior models. It showed that a large part of the standard deviation in software response time distributions can be related to the position of a software execution within its corresponding execution sequence. This paper's approach is orthogonal to our former work: instead of analyzing the position of a software execution in a trace, this paper analyzes the influence of concurrent software executions.

Menasce and Almeida [6] group similar requests in terms of performance resource usage to reduce variability. In this paper, classes are defined based on workload intensity instead of request types.

In software profiling, timing behavior is correlated to operation calls and to traces. Graham et al. [3] introduced the profiler *gprof*, which provides caller-context information (i.e., makes caller-callee relations explicit). Ammons et al. [1] extend this work to stack-context equivalence, i.e., two executions are considered equivalent if the execution stack contains the same sequence of operations. Both approaches do not consider workload intensity.

Bailey and Soucy [2] categorize requests into trivial, intermediate, and complex service requests. The timing behavior of requests are compared against request-type-specific response time objectives for failure diagnosis. Workload intensity in terms of concurrent usage is not considered.

Maxion [5] evaluated network characteristics in the context of workload intensity changes by considering the time of the day. The approach is based on the observation, that the system studied (a large university network) shows typical workload intensity patterns over the day. The approach also models week days and weekends separately. This approach is similar to ours in that the workload intensity is considered in evaluating system behavior. However, our approach addresses software timing behavior instead of network characteristics and does not model day time patterns.

6. CONCLUSIONS

We addressed the relation between varying workload intensity in multi-user software systems and the resulting statistical variance in timing behavior distributions obtained from monitoring data. This paper introduces our approach for the analysis of monitoring data, called WITiBA (Workload-Intensity-Sensitive Timing Behavior Analysis). WITiBA considers inter-dependencies between concurrent executions of software operations within a distributed system to reduce the standard deviation for succeeding analysis steps. This can be beneficial for many timing behavior analysis methods or simulation methods in terms of smaller confidence intervals, or shorter simulation time.

The case study results show that a significant amount of model variance can be reduced by using our workload metrics. The largest benefit can be achieved with the metric pwi_4, which considers concurrent executions of software operations in a distributed system. pwi_4 learns weights from monitoring data to address the observation that concurrent executions of software operations have operation-specific timing behavior inter-dependencies. These inter-dependencies result for instance from the competition for the same resources.

The implementation prototype reasonably scales: analyzing hundred thousand executions requires less than a minute on a standard desktop computer. The pwi_4 metric requires additional computational effort for learning the weight vectors, which depends on the learning technique used, the number of learning iterations, and the number of different monitoring points within the corresponding execution environments. In CS-3, weights for 161 software operations were computed within several minutes for each operation.

A constant monitoring overhead in the order of microseconds for each activated monitoring point and operation execution could be achieved with the Kieker framework[2] [9] for monitoring calling dependencies and response times. In our ongoing field studies, the monitoring overhead was reported to be small if only major platform services (e.g., ≤ 50) were instrumented.

Future work is to compare workload intensity with other timing behavior influences, such as parameter values, and request types.

References

[1] Ammons, G., Ball, T., and Larus, J. R. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. Conf. on Programming Language Design and Implementation (PLDI'97)* (1997), ACM, pp. 85–96.

[2] Bailey, R. M., and Soucy, R. C. Performance and availability measurement of the IBM information network. *IBM Systems Journal 22*, 4 (1983), 404–416.

[3] Graham, S. L., Kessler, P. B., and McKusick, M. K. gprof: A call graph execution profiler. *SIGPLAN Notes 17*, 6 (1982), 120–126.

[4] Jain, R. *The Art of Computer Systems Performance Analysis*, first ed. John Wiley & Sons, Apr. 1991.

[5] Maxion, R. A. Anomaly detection for diagnosis. In *Proc. Int'l Symp. on Fault-Tolerant Computing (FTCS '90)* (June 1990), IEEE, pp. 20–27.

[6] Menascé, D. A., and Almeida, V. A. *Capacity Planning for Web Services: Metrics, Models, and Methods.* Prentice Hall, Oct. 2001.

[7] Mielke, A. Elements for response-time statistics in ERP transaction systems. *Performance Evaluation 63*, 7 (July 2006), 635–653.

[8] Rohr, M., van Hoorn, A., Giesecke, S., Matevska, J., Hasselbring, W., and Alekseev, S. Trace-context sensitive performance profiling for enterprise software applications. In *Proc. SPEC Int'l Performance Evaluation Workshop (SIPEW '08)* (June 2008), vol. 5119 of *LNCS*, Springer, pp. 283–302.

[9] Rohr, M., van Hoorn, A., Matevska, J., Sommer, N., Stoever, L., Giesecke, S., and Hasselbring, W. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proc. IASTED Int'l Conf. on Software Engineering 2008* (Feb. 2008), ACTA Press, pp. 80–85.

[10] van Hoorn, A., Rohr, M., and Hasselbring, W. Generating probabilistic and intensity-varying workload for Web-based software systems. In *Proc. SPEC Int'l Performance Evaluation Workshop (SIPEW '08)* (June 2008), vol. 5119 of *LNCS*, Springer, pp. 124–143.

[2] `http://kieker.sourceforge.org`