Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems

Fabian Brosig
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
Email: fabian.brosig@kit.edu

Nikolaus Huber
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
Email: nikolaus.huber@kit.edu

Samuel Kounev
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
Email: kounev@kit.edu

***Abstract*—Modern enterprise applications have to satisfy increasingly stringent Quality-of-Service requirements. To ensure that a system meets its performance requirements, the ability to predict its performance under different configurations and workloads is essential. Architecture-level performance models describe performance-relevant aspects of software architectures and execution environments allowing to evaluate different usage profiles as well as system deployment and configuration options. However, building performance models manually requires a lot of time and effort. In this paper, we present a novel automated method for the extraction of architecture-level performance models of distributed component-based systems, based on monitoring data collected at run-time. The method is validated in a case study with the industry-standard SPECjEnterprise2010 Enterprise Java benchmark, a representative software system executed in a realistic environment. The obtained performance predictions match the measurements on the real system within an error margin of mostly 10-20 percent.**

I. Introduction

Modern enterprise applications have to satisfy increasingly stringent requirements for performance, scalability and efficiency. The ability to provide performance guarantees under certain resource efficiency constraints is gaining in importance. To avoid the pitfalls of inadequate Quality-of-Service (QoS), the expected performance characteristics of systems need to be evaluated during all phases of their life cycle. Estimating the level of performance a system can achieve is normally done by means of system performance models that are used to predict the performance and resource utilization of the system under specific workloads. During system development, such models can be exploited to evaluate the performance of early-stage prototypes. During system deployment, capacity planning can be conducted without the need for expensive load testing. During operation, a performance model can be used as a basis for continuous performance-aware resource management over time [1], [2]. In all of these contexts, typically the following questions arise:

- What will be the mean service response times and the utilization of system resources under a given workload?
- What will be the impact of a changing workload (e.g., transaction mix and/or workload intensity)?
- What are the maximum load levels that the system will be able to handle and which resources would become bottlenecks under load?
- How would the system performance change if resources are added or removed, e.g., nodes in an application server cluster are added or removed?
- How much resources are required to satisfy performance and availability requirements while ensuring efficient resource usage?

In order to answer such questions using a performance model, the model needs to be designed to reflect the abstract system structure and capture its performance-relevant aspects. We distinguish between *predictive performance models* and *descriptive architecture-level performance models*. The former capture the temporal system behavior and can be used directly for performance prediction by means of analytical or simulation techniques (e.g., queueing networks). The latter describe performance-relevant aspects of the system's software architecture, execution environment, and usage profile (e.g., UML models augmented with performance annotations). They can normally be transformed into predictive performance models by means of automatic model-to-model transformations [3]. By modeling the system's performance-influencing factors explicitly, architecture-level models allow to predict the impact of changes in the software components, resource allocations and/or system workloads.

Over the past decade, a number of architecture-level performance meta-models have been developed by the performance engineering community. Proposed meta-models are, e.g., the UML SPT and MARTE profiles [4], CSM [5], PCM [6] and KLAPER [7]. However, building models that accurately capture the different aspects of system behavior is a challenging task and requires a lot of time when applied manually to large and complex real-world systems [8], [9], [10]. Given the costs of building performance models, techniques for automated model extraction based on observation of the system at run-time are highly desirable. Current performance analysis tools used in industry mostly focus on profiling and monitoring transaction response times and resource consumption. They often provide large amounts of low-level data while important

978-1-4577-1639-3/11/$26.00 ASE 2011, Lawrence, KS, USA

information about the end-to-end performance behavior is missing (e.g., service control flow and resource demands).

In this paper, we propose a novel method for automated extraction of architecture-level performance models for distributed component-based systems. The method uses run-time monitoring data to extract the model structure based on tracing information and to extract model parameters based on easily obtainable performance metrics such as throughput, resource utilization and response times. The method supports point-to-point asynchronous communication and includes an approach for modeling probabilistic parameter dependencies. We implemented the proposed extraction method and evaluated its effectiveness by applying it to a system of a representative size and complexity - a deployment of the industry-standard SPECjEnterprise2010 benchmark[1]. The benchmark is deployed in a complex execution environment consisting of a cluster of application servers and a back-end database server for persistence. We extracted a performance model of the system and evaluated its predictive power in scenarios of increasing complexity.

The presented method builds on our approach sketched in [11] where we described a preliminary implementation of some parts of our extraction method suffering from many limitations and restrictions. We studied a small part of a pre-release version of SPECjEnterprise2010 deployed in a small testing environment which was not representative of real-life deployments. While the results were encouraging, it was not clear if our approach was applicable to realistic applications and what accuracy and scalability it provides. Given the many limitations of our original method, a larger part of the SPECjEnterprise2010 application (e.g., Java servlets, Java Server Pages, Java Message Service, Message-Driven Beans) could not be considered in the extraction. The work presented here addresses the above challenges allowing to extract a model of the entire SPECjEnterprise2010 application. In addition, it includes a detailed validation and evaluation of our approach.

In summary, the contributions of this paper are: i) a method for automated extraction of architecture-level performance models of component-based systems, including an approach to characterize parametric dependencies probabilistically, ii) a comprehensive evaluation of our approach in the context of a case study of a representative system deployed in a realistic environment. To the best of our knowledge, no similar studies exist in the literature considering systems of the size, representativeness and complexity of the one considered here. We demonstrate that the automated extraction of architecture-level performance models using measurement data obtained with common monitoring tools is feasible and provides a solid basis for capacity planning. The latter can be exploited for continuous performance-aware resource management improving the system's efficiency and thus lowering its total-cost-of-ownership.

[1]SPECjEnterprise2010 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjEnterprise2010 is located at http://www.spec.org/jEnterprise2010.

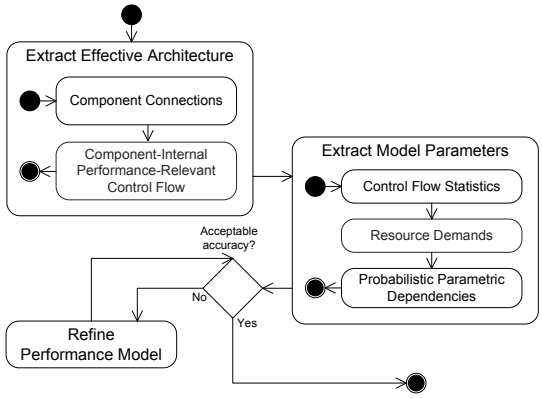

Fig. 1. Model extraction process.

The remainder of this paper is organized as follows. In Section II, we describe the method for the automated extraction of architecture-level performance models. We then describe a proof-of-concept implementation in Section III followed by a case study in Section IV. We review related work in Section V and conclude the paper in Section VI.

II. Automated Performance Model Extraction

Our goal is to automate the process of building performance models by observing the system behavior at run-time. Based on monitoring data, performance-relevant abstractions and parameters should be automatically extracted. The process we employ to extract architecture-level performance models includes three main steps depicted in Figure 1. First, we extract the system's effective architecture [12], i.e., the set of components and connections between components that are effectively used at run-time. Second, we extract performance model parameters characterizing the system's control flow and resource demands. Third, the resulting model is iteratively refined until it provides the required accuracy.

A. Extraction of Effective Architecture

Given a component-based software system, extracting its architecture requires identifying its building blocks and the connections between them. In the context of performance model extraction, also the components' performance-relevant internal structure needs to be extracted. Note that, as mentioned above, we consider only those parts of the architecture that are effectively used at run-time.

1) Component Connections: Componentization is the process of breaking down the considered software system into components. It is part of the mature research field of software architecture reconstruction [13]. Component boundaries can be obtained in different ways, e.g., specified manually by the system architect or extracted automatically through static code analysis (e.g., [14]). The granularity of the identified components determines the granularity of the work units whose performance needs to be characterized, and hence the

granularity of the resulting performance model. In our context, a component boundary is specified as a set of software building blocks considered as a single entity from the point of view of the system's architect. For instance, this can be a set of classes or a set of Enterprise Java Beans and Servlets.

Once the component boundaries have been determined, the connections between the components can be automatically identified based on monitoring data. We determine the control flow between the identified components using *call path tracing*. We analyze monitoring data consisting of event records obtained through instrumentation. An event record represents an entry or exit of a software building block. In order to trace individual system requests, the list of gathered event records has to be grouped and ordered. The set of groups is equal the set of equivalence classes $[a]_{\mathcal{R}}$ according to the following equivalence relation $\mathcal{R}$: Let a, b be event records obtained through instrumentation. Then a relates to b ($a \sim_{\mathcal{R}} b$) if and only if a and b were triggered by the same system request. This is well-defined because an event record is triggered by exactly one system request. In the following, equivalence classes are denoted as *call path event record sets*. Ordering the elements of a call path event record set in chronological order results in a *call path event record list*. From this list a call path can be obtained. We refer to [15], [16], [12] where call path event record lists are transformed to UML sequence diagrams. Given a list of call paths and the knowledge about component boundaries, the list of effectively used components, as well as their actual entries (provided services) and exits (required services) can be determined. Furthermore, for each component's provided service one can determine the list of external services that are called.

When obtaining monitoring data, obviously only those paths that are exercised can be captured. Thus, a representative usage profile is a prerequisite for a dynamic control flow analysis.

2) Component-Internal Performance-Relev. Control Flow: After extracting the components and the connections between them, the performance-relevant control flow inside the components is examined. In order to characterize the performance of a component's provided service, we need to know about its internal resource demanding behavior, on the one hand, and how it makes use of external service calls, on the other hand. Obviously, it makes a difference if an external service is called once or, e.g., ten times within a loop. Furthermore, the ordering of external service calls and internal computations of the provided service may have an influence on the component's performance. The performance-relevant control flow we aim to extract is an abstraction of the actual control flow. Performance-relevant actions are internal computation tasks and external service calls, hence also loops and branches where external services are called.

The set of call paths derived in the previous substep provides information on how a provided component service relates to external service calls of the component's required services. Formally, let $X^C = x_1^C, \ldots, x_{n^C}^C$ be the set of provided services of component C and $Y^C = y_1^C, \ldots, y_{m^C}^C$ the set of its required services. For compactness, we omit the index C from now on. Then, the call paths constitute a relation $\mathcal{G} = \{(x, S_x) | x \in X, S_x \subseteq S\}$ with $S = \{(l_1, \ldots, l_k) | k \in \mathbb{N}, l_i \in Y (\forall i : 1 \leq i \leq k)\}$, where S_x represents the set of sequences of observed external service calls for provided service x.

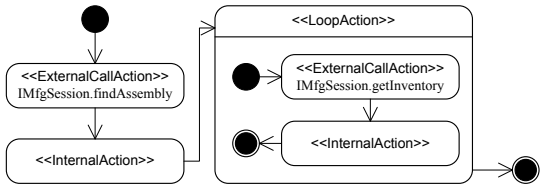

Fig. 2. Example: Performance-relevant component service control flow.

For instance, if there is a provided service x so that $\forall S_x : (x, S_x) \in \mathcal{G} : S_x = y_1 \vee S_x = y_2$ holds, then one could assume that service x has a control flow where either y_1 or y_2 is called, i.e., that there is a branch. Multiple approaches exist for determining the performance-relevant control flow constructs, e.g., [17], [14]. By instrumenting the performance-relevant control flow constructs inside the component, the *effective* component-internal control flow can be directly extracted from the obtained call paths. An example of a performance-relevant service control flow is depicted in Figure 2.

B. Extraction of Model Parameters

Having extracted the performance-relevant control flow structure in the previous step, the model then needs to be parameterized with statistical information about the control flow including resource demand quantifications.

1) Control Flow Statistics: As control flow statistics, we extract branch probabilities and loop iteration numbers. Both parameters can be extracted from the call paths obtained in the previous step. While it is sufficient to characterize a branch probability with a mean value, for a loop iteration number, the call paths allow deriving a Probability Mass Function (PMF). For instance, the call paths might reveal that a loop is executed either two times or ten times, each with a probability of 50%. In that case, the average value would not be a proper approximation as a loop iteration number.

2) Resource Demands: To characterize the resource demanding behavior of a component's service, the resource demands of its internal computations need to be quantified. Determining resource demands involves identification of the resources used by the service and quantification of the amount of time spent using these resources. The resource demand of a service is its total processing time at the considered resource not including any time spent waiting for the resource to be made available. Referring to Figure 2, the internal actions need to be annotated with demands for the used resources (e.g., CPU, HDD). Resource demand estimation is an established research area. Typically, resource demands are estimated based on measured response times or resource utilization and throughput data [18], [19]. Most approaches focus on CPUs and I/O resources, i.e., memory accesses are normally not considered explicitly.

3) Probabilistic Parametric Dependencies: The estimates of the model parameters described in the previous two sections are averaged over the observed call paths. In case the behavior of the considered service depends on one or more input parameters passed when invoking the service, such parametric dependencies can be captured in the extracted model. For instance, a branch probability might heavily depend on the value of an input parameter and in such cases it is desirable to be able to quantify such dependencies. By monitoring service input parameters and co-relating observed parameter values with the observed service control flow, we can derive probabilistic models of parametric dependencies.

However, if no a priori information about the existence of parametric dependencies is available, their automatic discovery based on monitoring data alone is a challenging problem that requires the use of complex correlation analysis or machine learning techniques [20]. An automatic detection method has to cope with high degrees of freedom: Each control flow construct might depend on multiple input parameters and the possible dependencies are not restricted to parameter values, other parameter characterizations such as the length of an array might also have an influence. For these reasons, we consider the automatic discovery of parametric dependencies at run-time based on monitoring data alone to be impractical. We assume that information about the existence of potential performance-relevant parametric dependencies is available a priori, e.g., from the system developer or from an automatic detection method based on static code analysis. With the knowledge of which parametric dependencies to observe, we then use monitoring data to quantify the dependencies probabilistically.

We now discuss our approach to quantifying parametric dependencies based on monitoring data. For compactness of the presentation, we assume that each control flow action may depend on at most one input parameter. For each service execution and for each parameter dependent action, a tuple can be obtained that consists of the observed parameter characterization (e.g., the parameter value or its size) and information about which control flow path was taken:

- For a branch action with $bt_1, \ldots, bt_m$ branch transitions, the monitored control flow path is characterized by the index $i, 1 \leq i \leq m$ of the observed transition.
- For a loop action, the monitored control flow path is characterized by the observed loop iteration number $i, i \geq 0$.

Let v be a monitored parameter characterization, then $t_a = (v, i)$ is such an observed tuple for the considered parameter-dependent action a. Considering multiple service executions, for each parameter-dependent action a, a set of observed tuples $T_a = (t_{a,1}, \ldots, t_{a,n})$ can be extracted. In the following, we fix the action a so that we can omit the index, i.e., $T_a = T = (t_1, \ldots, t_n)$. Considering the taken control flow path for a given parameter characterization v as a discrete random variable, we can approximate its probability mass function (PMF) based on the observed tuple list. Let $V = \{v_k | t_k = (v_k, i_k) \in T\}$ be the set of observed parameter characterizations. Let $C = \{i_k | t_k = (v_k, i_k) \in T\}$ be the set of observed control flow paths taken. Then for each $v \in V$, the PMF of the taken control flow path can be approximated as $f_v : C \to [0, 1]$, where

$$f_v(i) = \frac{\#\{t_k | t_k = (v_k, i_k) \in T \wedge v_k = v \wedge i_k = i\}}{\#\{t_k | t_k = (v_k, i_k) \in T \wedge v_k = v\}}$$

If the set $V = \{v_1, \ldots, v_n\}$ is large, distinguishing the branch transition probabilities, loop iteration numbers or resource demands for each individual v_i is impractical. In this case, we partition the set V and work with partitions instead of individual values. The partitioning can be done using, e.g., equidistant or equiprobable bins. The distribution of the branch transition probabilities, loop iteration numbers and resource demands is then approximated partition-wise. The PMF representing one partition can be computed by aggregation: Let $P_r = \{v_{r,1}, \ldots, v_{r,l}\} \subseteq V$ be such a partition. Then the PMF $f_r : C \to [0, 1]$ representing the partition P_r is defined as the normalized weighted sum of the PMFs $f_{v_{r,1}}, \ldots, f_{v_{r,l}}$:

$$f_r : C \to [0, 1], \quad f_r(i) = \frac{\sum_{j=1}^{l} p(v_{r,j}) f_{v_{r,j}}(i)}{\sum_{j=1}^{l} p(v_{r,j})}$$

where

$$p : C \to [0, 1], \quad p(v) = \frac{\#\{t_k | t_k = (v_k, i_k) \in T \wedge v_k = v\}}{\#V}$$

C. Performance Model Calibration

Calibrating a performance model means comparing model predictions with measurements on the real system with the aim to adjust the model improving its accuracy. Given an observed deviation between predictions and measurements, in general, there are two ways of adjusting the performance model: (i) one possibility is to increase the model's granularity by adding details on the modeled system behavior, i.e., refining the model. The other possibility (ii) is to adjust the model parameters (e.g., branch probabilities or resource demands). In that case, the granularity of the model does not change.

In our approach, we propose two ways of calibrations: The resource demands are calibrated by a factor that represents the measurement overhead during resource demand extraction. Furthermore, given that component-based systems typically run on a complex middleware stack, system requests may get delayed until they enter the system boundaries where the measurement sensors are injected. These delays should be accounted for.

D. Overhead Control

When applying the model extraction, the monitoring overhead has to be kept low. Particularly, when extracting resource demand parameters as mentioned in Section II-B2, the overhead has to be low since otherwise the parameter extraction then may be biased.

To reduce the overhead of monitoring system requests, in general there exist two orthogonal concepts: (i) *quantitative throttling*: throttling the number of requests that are actually

monitored, (ii) *qualitative throttling*: throttling the level of detail requests are monitored at. Existing work on (i) is presented, e.g., in [21]. The authors propose an adaptive time slot scheduling for the monitoring process. The monitoring frequency depends on the load of the system. In phases of high load, the monitoring frequency is throttled. Concerning (ii), the monitoring approach presented in [22] allows an adaptive monitoring of requests, i.e., monitoring probes can be enabled or disabled depending on what information about the requests should be monitored.

III. Proof-of-Concept

In this section, we describe an implementation of our approach presented in the previous section in the context of a representative platform for distributed component-based applications. In Section IV, we then present a case study of a representative application demonstrating the effectiveness and practicality of our approach.

A. Context

We apply our automated performance model extraction approach in the context of the Java Platform, Enterprise Edition (Java EE), one of today's major technology platforms for building enterprise systems. Besides providing a framework for building distributed web applications, it includes a server-side framework for component-based applications, namely the Enterprise Java Beans (EJB) Architecture.

We use the Palladio Component Model (PCM) as an architecture-level performance modeling formalism to describe extracted models of running Java EE applications. PCM is a domain-specific modeling language for describing performance-relevant aspects of component-based software architectures [6]. It provides modeling constructs to capture the influence of the following four factors on the system performance: i) the implementation of software components, ii) the external services used by components, iii) the component execution environment and iv) the system usage profile. These performance-influencing factors are reflected in an explicit context model and a parameterized component specification. Recent surveys [23], [24], [25] show that the clear separation of these factors is one of the key benefits of PCM compared to other architecture-level performance models such as the UML SPT and MARTE profiles [4], CSM [5], or KLAPER [7].

PCM models are divided into five sub-models: The *repository model* consists of interface and component specifications. A component specification defines which interfaces the component provides and requires. For each provided service, the component specification contains a high-level description of the service's internal behavior. The description is provided in the form of a so-called *Resource Demanding Service Effect Specification (RDSEFF)*. The *system model* describes how component instances from the repository are assembled to build a specific system. The *resource environment model* specifies the execution environment in which a system is deployed. The *allocation model* describes the mapping of components from the system model to resources defined in the resource environment model. The *usage model* describes the user behavior. It captures the services that are called at runtime, the frequency (workload intensity) and order in which they are invoked, and the input parameters passed to them.

As an application server implementing the Java EE specifications, we employ Oracle WebLogic Server (WLS). WLS comes with an integrated monitoring platform, namely the WebLogic Diagnostics Framework (WLDF). WLDF is a monitoring and diagnostics framework that enables collecting and analyzing diagnostic data for a running WLS instance. The two main WLDF features that we make use of are the data harvester and the instrumentation engine.

The data harvester can be configured to collect detailed diagnostic information about a running WLS and the applications deployed thereon. The instrumentation engine allows injecting diagnostic actions in the server or application code at defined locations. In short, a location can be the beginning or end of a method, or before or after a method call. Depending on the configured diagnostic actions, each time a specific location is reached during processing, an event record is generated. Besides information about, e.g., the time when or the location where the event occurred, an event record contains a *diagnostic context id* which uniquely identifies the request that generated the event and allows to trace individual requests as they traverse the system.

B. Implementation

The implementation of the extraction process is based on our preliminary work in [11] which included a very basic and limited implementation of a subset of the extraction algorithm. While in [11] we only considered EJBs, our new implementation supports also Servlets and Java Server Pages (JSPs) as well as Message-Driven Beans (MDBs) for asynchronous point-to-point communication using the Java Message Service (JMS). In the following, we briefly describe the implementation of the extraction process.

For the extraction of the component connections according to Section II-A1, the component boundaries can be specified as groups of EJBs, Servlets and JSPs. Thus, WLDF is configured to monitor entries/exits of EJB business methods, Servlet services (including also JSPs) and JMS send/receive methods. As depicted in Figure 3, the WLDF diagnostic context id uniquely identifies a request, but forked threads receive the same context id. Hence, to separate the different call paths from each other, the context id is not sufficient. In those cases, we make use of the transaction id. The ordering of the event records is done via the event record id. Based on the set of observed call paths, the effective connections among components can be determined, i.e., required interfaces of components can be bound to components providing the respective services.

For the extraction of the component-internal performance-relevant control flow according to II-A2, we follow the approach described in [11]. Performance-relevant actions are made explicit by method refactorings. This is because of the lack of tool support for in-method instrumentation. Current

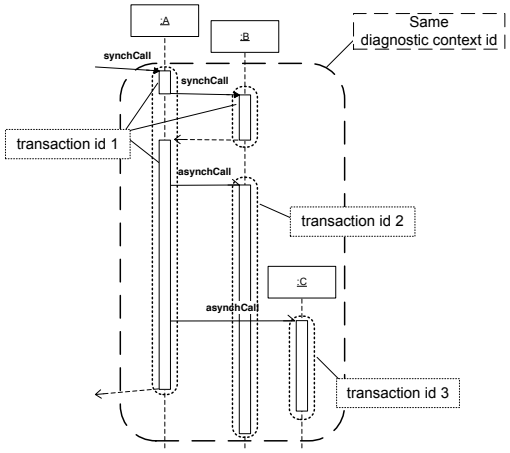

Fig. 3. Diagnostic context id and transaction ids during asynchronous messaging.

instrumentation tools including WLDF support only method-level instrumentation. They do not support instrumentation at custom locations other than method entries and exits.

The sending of asynchronous JMS messages is modeled as fork action. Control flow statistics are collected in parallel to the extraction of the abstract component-internal control flow.

For the resource demand estimation of individual internal actions, we apply two approaches: i) in phases of low load we approximate resource demands with measured response times, ii) in phases of medium to high load we estimate resource demands based on measured utilization and throughput data with weighted response time ratios [11].

Concerning the extraction of probabilistic parametric dependencies, the current WLDF version limits the type of parameters that can be examined. While WLDF allows injecting monitors providing information about method input parameters, it provides only String representations of the monitored parameters. For complex types, only the object name is provided. Thus, complex parameter types currently cannot be monitored appropriately.

An example of an extracted PCM RDSEFF is shown in Figure 4. In includes internal actions, external call actions and a loop action. The loop action is annotated with a loop iteration number, specified as a PMF. With a probability of 30%, the loop body is executed 11 times, with a probability of 50%, it is executed 10 times and with a probability of 20% it is executed 9 times. Notice that both internal actions of the RDSEFF are enriched with a CPU demand annotation. While PCM supports generic work units, in the context of this paper, a demand of 1 CPU unit is understood as a demand of 1 millisecond CPU time.

IV. Case Study: Modeling SPECjEnterprise2010

To evaluate the implemented model extraction method, we applied it to a case study of a representative Java EE application. The application we consider is the SPECjEnterprise2010

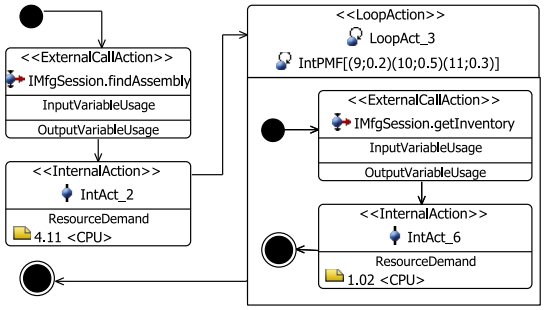

Fig. 4. Example RDSEFF for the provided service `scheduleWorkOrder`.

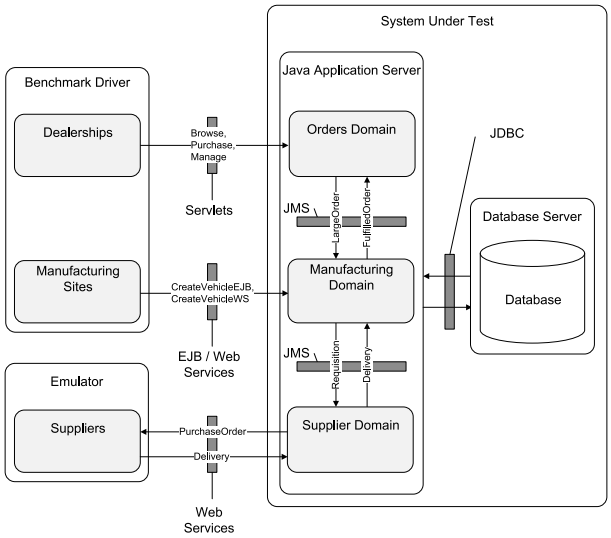

Fig. 5. SPECjEnterprise2010 architecture [26].

benchmark. We start with an overview of the benchmark followed by a description of our experimental environment.

A. SPECjEnterprise2010

SPECjEnterprise2010 is a Java EE benchmark developed by SPEC's Java subcommittee for measuring the performance and scalability of Java EE-based application servers. The benchmark workload is generated by an application that is modeled after an automobile manufacturer. As business scenarios, the application comprises customer relationship management (CRM), manufacturing and supply chain management (SCM). The business logic is divided into three domains: orders domain, manufacturing domain and supplier domain. Figure 5 depicts the architecture of the benchmark as described in the benchmark documentation. The application logic in the three domains is implemented using EJBs which are deployed on the considered Java EE application server. The domains interact with a database server via Java Database Connectivity (JDBC) using the Java Persistence API (JPA). The communication between the domains is asynchronous and implemented using point-to-point messaging provided by the Java Message Service (JMS). The workload of the orders domain is triggered by dealerships whereas the workload of the manufacturing domain is triggered by manufacturing sites.

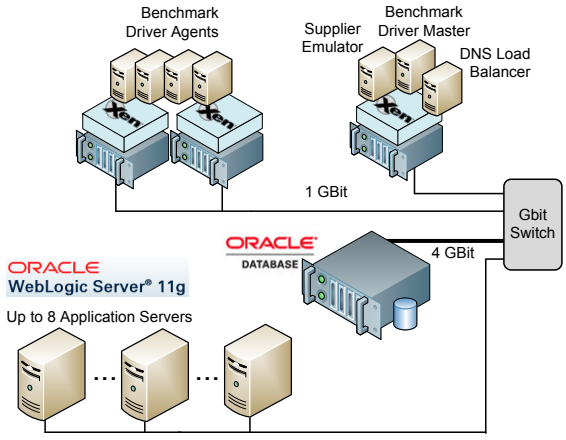

Fig. 6. Experimental environment.

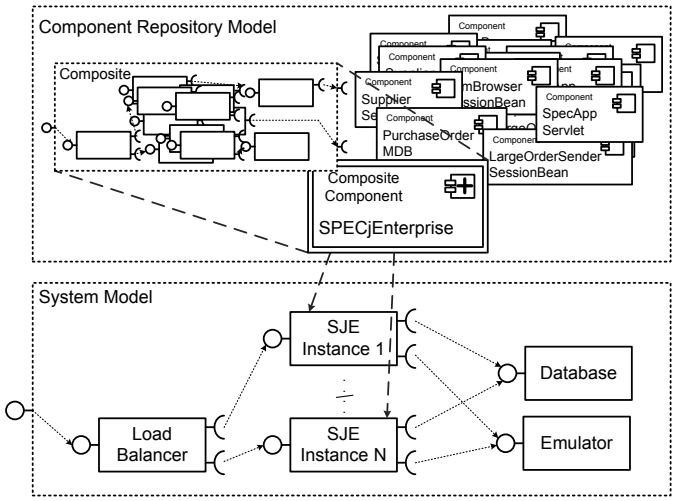

Fig. 7. SPECjEnterprise2010 PCM model structure.

Both, dealerships and manufacturing sites are emulated by the benchmark driver, a separate supplier emulator is used to emulate external suppliers. The communication with the suppliers is implemented using Web Services. While the orders domain is accessed through Java Servlets, the manufacturing domain can be accessed either through Web Services or EJB calls, i.e., Remote Method Invocation (RMI). The benchmark driver executes five benchmark operations. A dealer may *browse* through the catalog of cars, *purchase* cars, or *manage* his dealership inventory, i.e., sell cars or cancel orders. In the manufacturing domain, work orders for manufacturing vehicles are placed, triggered either per WebService or RMI calls (*createVehicleWS* or *createVehicleEJB*).

B. Experimental Environment

We installed the benchmark in the system environment depicted in Figure 6. The benchmark application is deployed in an Oracle WebLogic Server (WLS) 10.3.3 cluster of up to eight physical nodes. Each WLS instance runs on a 2-core Intel CPU with OpenSuse 11.1. As a database server (DBS), we used Oracle Database 11g, running on a Dell PowerEdge R904 with four 6-core AMD CPUs, 128 GB of main memory, and 8x146 GB SAS disk drives. The benchmark driver master, multiple driver agents, the supplier emulator and the DNS load balancer were running on separate virtualized blade servers. The machines are connected by a 1 GBit LAN, the DBS is connected with 4 x 1 GBit ports.

C. Evaluation

1) Performance Model: While in [11], the focus was on modeling the manufacturing domain of a pre-version of the benchmark, the study presented here considers the entire benchmark application, i.e., including supplier domain, dealer domain, the web tier and the asynchronous communication between the three domains. We had to deal with EJBs including MDBs for asynchronous point-to-point communication, web services, Servlets and JSPs. Figure 7 presents a high-level overview of the basic structure of the extracted PCM performance model. The system model configuration shows a load balancer which distributes incoming requests to replicas of the SPECjEnterprise2010 benchmark application which themselves need an emulator instance and a database instance. A benchmark application instance refers to a composite component which is located in the component repository. The composite component in turn consists of component instances, e.g., a `SpecAppServlet` component or a `PurchaseOrderMDB` component. These components reside in the repository as well. The performance model of the benchmark application consists of 28 components whose services are described by 63 RDSEFFs. In total, 51 internal actions, 41 branch actions and four loop actions have been modeled.

The resources we considered were the CPUs of the WLS instances (WLS CPU) and the CPUs of the database server (DBS CPU). The network and hard disk I/O load could be neglected. Note that we configured load-balancing via DNS.

We extracted a PCM model whose resource demands were estimated based on utilization and throughput data. For the apportioning of resource demands among WLS CPU and DBS CPU, see [11]. To keep the overhead low, we separated the extraction step in which call paths are monitored from the extraction step in which resource demands are estimated. Both steps were conducted with one single WLS instance. The resource demands were extracted during a benchmark run with a steady state time of 900 sec and a WLS CPU utilization of about 60%. The same benchmark run was then executed without any instrumentation in order to quantify the instrumentation overhead factor and calibrate the estimates of the WLS CPU resource demands. Furthermore, we measured the delay for establishing a connection to the WLS instance which is dependent on the system load. With the knowledge of the number of connections the individual benchmark operations trigger, the load-dependent delay is estimated and taken into account in the predicted response times.

For the solution of the extracted PCM model, we used the queueing-network-based simulator SimuCom [6].

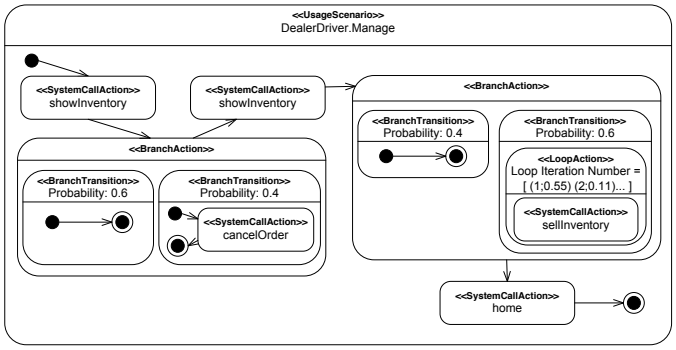

Fig. 8. Usage scenario for operation *Manage*.

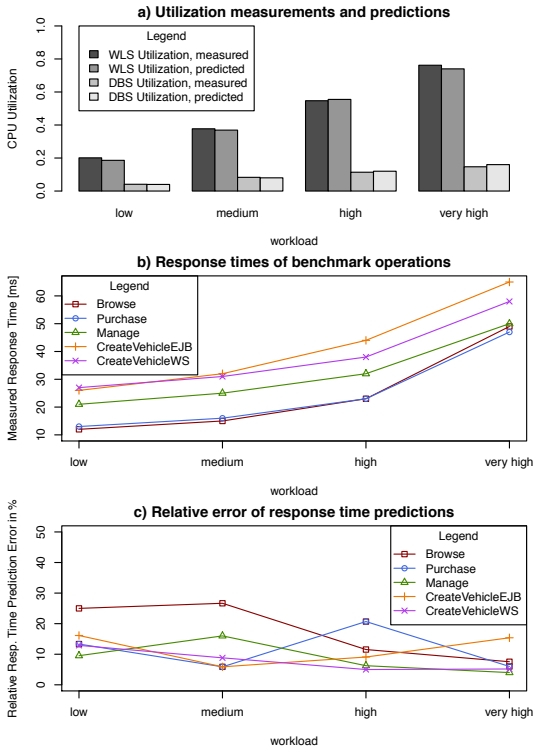

Fig. 9. Scenario 1: Measurements, prediction results and prediction errors.

2) Evaluation Results: The extracted PCM models are validated by comparing the model predictions with measurements on the real system. The usage model representing the benchmark workload has been provided manually. The five benchmark operations are modeled as individual usage scenarios. Figure 8 shows the usage scenario of benchmark operation *Manage* in a notation similar to UML activity diagrams. It consists of several system calls, two branches with corresponding transition probabilities and a loop action. The loop iteration number is given as a PMF that was derived from monitoring data. The usage scenarios of the remaining four benchmark operations are of similar complexity.

In the investigated scenarios we vary the throughputs of the dealer and manufacturing driver as well as the deployment configuration. Note that we extracted the performance model on a single WLS instance whereas for validation, WLS clusters of different sizes were configured. As performance metrics, we considered the average response times of the five benchmark operations as well as the average utilization of the WLS CPU and the DBS CPU. In clustered scenarios where several WLS instances are involved, we considered the average utilization of all WLS CPUs. Note that the response times of the benchmark operations are measured at the driver agents, i.e., the WLS instances run without any instrumentation. For each scenario, we first predicted the performance metrics for low load conditions ($\approx 20\%$ WLS CPU utilization), medium load conditions ($\approx 40\%$), high load conditions ($\approx 60\%$) and very high load conditions ($\approx 80\%$) and then compared them with steady-state measurements on the real system.

Scenario 1: Cluster of two application server instances. For the first validation scenario, we configured an application server cluster consisting of two WLS instances and injected different workloads. The measured and predicted server utilization for the different load levels are depicted in Figure 9 a). The utilization varies from 20% to 80%. The utilization predictions fit the measurements very well, both for the WLS instances as well as for the DBS. Figure 9 b) shows the response times of the benchmark operations for the four load levels. The response times vary from 10ms to 70ms. As expected, the higher the load, the faster the response times grow. Compared to the other benchmark operations, *Browse* and *Purchase* have lower response times, while *CreateVehicleEJB* and *CreateVehicleWS* take most time. In Figure 9 c), the relative error of the response time predictions is shown. The error is mostly below 20%, only *Browse* has a higher error but still lower than 30%. The prediction accuracy of the latter increases with the load. This is because *Browse* has a rather small resource demand but includes a high number of round trips to the server translating in connection delays (15 on average).

Scenario 2: Cluster of four application server instances. In Scenario 2, we considered a four node WLS cluster again at four different load levels. Figure 10 a) shows the measurements and predictions of the server utilization. Again, the predictions are accurate. However, one can identify a small deviation of the DBS CPU utilization that is growing with the load.

Figure 10 b) depicts the relative response time prediction error for Scenario 2. Again, the relative error is mostly below 20%. For the same reasons already mentioned in Scenario 1, operation *Browse* stands out a little. However, its prediction error is still below 30%.

Scenario 3: Cluster of eight application server instances. For Scenario 3, we deployed the benchmark in a cluster of eight WLS instances. As shown in Figure 11, the server utilization predictions are very accurate. While the DBS utilization prediction exhibits an error that grows with the injected load, it does not exceed 15%.

In cases of low to medium load, the accuracy of the predicted response times is comparable to Scenarios 1 and 2

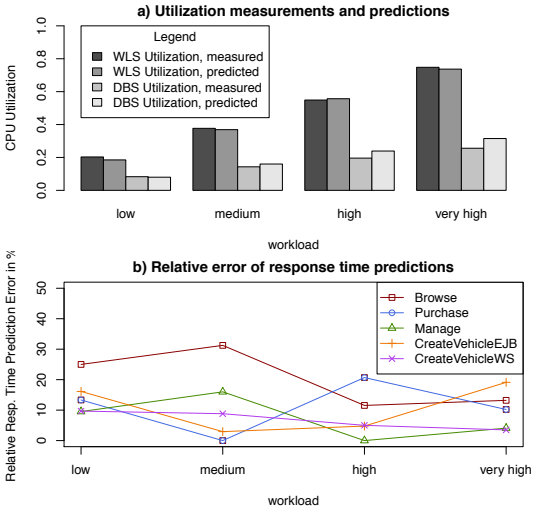

Fig. 10. Scenario 2: Utilization measurements and relative errors.

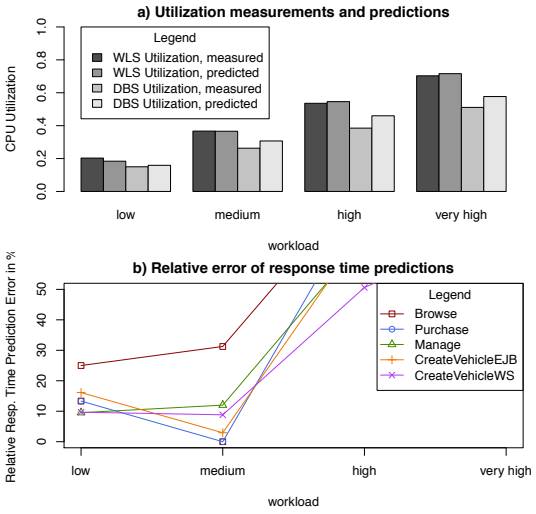

Fig. 11. Scenario 3: Utilization measurements and predictions and relative error of response time predictions.

(Figure 11 b). However, in cases of high load, the prediction error grows by an order of magnitude. This is because under this load level some of the WLS cluster nodes were overloaded, i.e., the cluster was not load-balanced anymore. The overloaded WLS instances then lead to biased response time averages. We assume that the cluster is unbalanced due to DNS caching effects that are not reflected in the performance model.

3) Summary: We validated the performance model that was extracted with the automated method as described in Sections II and III in various realistically-sized deployment environments under different workload mixes and load intensity. Concerning the CPU utilization, the observed prediction error for the WLS application server was below 5%. For the database server, the CPU utilization prediction error was mostly below 10%. The response time predictions of the benchmark operations mostly had an error of 10% to 20%. In the case of the eight node application server cluster, the response time predictions were inaccurate for higher loads. This is because the cluster was not load-balanced anymore.

V. Related Work

Each part of the approach presented in this paper is related to multiple extraction approaches: a) trace-based approaches, b) resource demand estimation, c) run-time monitoring, d) benchmark-based approaches and e) extraction of PCM instances.

Trace-based approaches. Call path tracing is a form of dynamic analysis which a number of approaches apply to gain reliable data on the actual execution of an application. Hrischuk et al. [16] use such an approach to extract performance models in the scope of "Trace-Based Load Characterization". Israr et al. [12] use pattern matching on trace data to differentiate between asynchronous, blocking synchronous, and forwarding communication. Both approaches use Layered Queueing Networks (LQNs) as the target performance model and are hence limited to LQN structures. Thus, features such as stochastic characterizations of loop iteration numbers or branch probabilities are not supported. Briand et al. [15] extract UML sequence diagrams from trace data which is obtained by aspect-based instrumentation.

Resource demand estimation. Approaches to resource demand estimation can be found in [18], [19], [27]. While in [18] applies the Service Demand Law directly for single workload classes, linear regression approaches to partition resource demand among multiple workload classes can be found in [19], [28]. In [27], utilization and throughput data is used to build a Kalman filter estimator.

Run-time monitoring. Approaches such as [29], [30] use systematic measurements to build mathematical models or models obtained with genetic optimization. However, the models serve as interpolation of the measurements and a representation of the system architecture is not extracted. Tools for automatic and adaptive monitoring are presented in [31], [32], [33], all focused on monitoring and collecting performance data of Java applications, but they are not explicitly focused on component-based architectures.

Benchmark-based approaches. Denaro et al. [34] and Chen et al. [35] identify the middleware as a key factor for performance problems in component-based applications. To allow performance prediction, Denaro develops application-specific performance test cases to be executed on available middleware platforms. Chen et al. propose a simple benchmark that processes typical transaction operations to extract a performance profile of the underlying component-based middleware and to construct a generic performance model.

PCM extraction. Several approaches are concerned with the extraction of PCM instances from code. The tool Java2PCM [17] extracts component-level control flow from Java code. Krogmann et al. [20] extract behavior models via static and dynamic analysis but do not focus on extracting timing values during system operation and their approach relies on manual instrumentations.

VI. Concluding Remarks

In this paper, we presented a method to automate the extraction of architecture-level performance models of distributed component-based systems using run-time monitoring data. We combined existing techniques such as call path tracing and resource demand estimation to an end-to-end model extraction method. As a proof-of-concept, we implemented the method with state-of-the-art, component-of-the-shelf monitoring tools. We validated it with the representative, industry-standard SPECjEnterprise2010 benchmark application in realistically scaled deployment environments. We extracted a performance model and derived performance predictions for various validation scenarios. Resource utilizations are predicted with a relative error of mostly 5%. Response times, which are typically much harder to predict accurately, are predicted with a relative error of about 10 to 20%.

We could show that an automated extraction of architecture-level performance models using measurement data obtained with common monitoring tools is a feasible way to extract performance models for distributed component-based systems. The obstacle of building a performance model manually can be overcome. Performance engineers can use the automated extraction process to obtain performance models allowing to answer sizing, resource efficiency and performance questions in an effective way.

However, further investigations concerning the extraction of internal service behavior and the characterization of probabilistic parametric dependencies are of concern. In addition, we are interested in studies with performance predictions for highly configurable, dynamic virtualized environments. The latter will reveal new challenges how appropriate architecture-level performance model representations should look like in the context of virtualized environments.

References

[1] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic qos management and optimisation in service-based systems," *IEEE Trans. on Softw. Eng.*, vol. 99, no. PrePrints, 2010.

[2] S. Kounev, F. Brosig, N. Huber, and R. Reussner, "Towards self-aware performance and resource management in modern service-oriented systems," in *IEEE SCC 2010*.

[3] P. Meier, S. Kounev, and H. Koziolek, "Automated Transformation of Palladio Component Models to Queueing Petri Nets," in *MASCOTS 2011*.

[4] Object Management Group (OMG), "UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)," May 2006.

[5] D. Petriu and M. Woodside, "An intermediate metamodel with scenarios and resources for generating performance models from UML designs," *Softw. and Syst. Modeling*, 2007.

[6] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, 2009.

[7] V. Grassi, R. Mirandola, and A. Sabetta, "Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach," *J. Syst. Softw.*, 2007.

[8] J. L. Hellerstein, "Engineering autonomic systems," in *ICAC 2009*.

[9] V. Cortellessa, A. Di Marco, and P. Inverardi, "Transformations of software models into performance models," in *ICSE 2005*.

[10] S. Kounev, "Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets," *IEEE Trans. on Softw. Eng.*, 2006.

[11] F. Brosig, S. Kounev, and K. Krogmann, "Automated Extraction of Palladio Component Models from Running Enterprise Java Applications," in *ROSSA*, 2009.

[12] T. Israr, M. Woodside, and G. Franks, "Interaction tree algorithms to extract effective architecture and layered performance models from traces," *J. Syst. Softw.*, 2007.

[13] P. Tonella, M. Torchiano, B. D. Bois, and T. Systä, "Empirical studies in reverse engineering: state of the art and future trends," *Empirical Software Engineering*, 2007.

[14] L. Chouambe, B. Klatt, and K. Krogmann, "Reverse Engineering Software-Models of Component-Based Systems," in *12th European Conference on Software Maintenance and Reengineering*, 2008.

[15] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Trans. on Softw. Eng.*, 2006.

[16] C. E. Hrischuk, M. Woodside, J. A. Rolia, and R. Iversen, "Trace-Based Load Characterization for Generating Performance Software Models," *IEEE Trans. on Softw. Eng.*, 1999.

[17] T. Kappler, H. Koziolek, K. Krogmann, and R. Reussner, "Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering," in *Software Engineering*, 2008.

[18] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy, *Capacity planning and performance modeling: from mainframes to client-server systems*. Prentice-Hall, Inc., 1994.

[19] J. Rolia and V. Vetland, "Parameter estimation for performance models of distributed application systems," in *CASCON*, 1995.

[20] K. Krogmann, M. Kuperberg, and R. Reussner, "Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction," *IEEE Trans. on Softw. Eng.*, 2010.

[21] K. Gilly, S. Alcaraz, C. Juiz, and R. Puigjaner, "Analysis of burstiness monitoring and detection in an adaptive web system," *Computer Networks*, 2009.

[22] J. Ehlers and W. Hasselbring, "Self-adaptive software performance monitoring," in *Software Engineering*, 2011.

[23] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Trans. on Softw. Eng.*, 2004.

[24] S. Becker, L. Grunske, R. Mirandola, and S. Overhage, "Performance prediction of component-based systems - a survey from an engineering perspective," in *Architecting Sys. with Trustworthy Components*, 2004.

[25] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, 2010.

[26] SPECjEnterprise2010 Design Document, http://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html. 2011.

[27] T. Zheng, M. Woodside, and M. Litoiu, "Performance Model Estimation and Tracking Using Optimal Filters," *IEEE Trans. on Softw. Eng.*, 2008.

[28] G. Pacifici, W. Segmuller, M. Spreitzer, and A. N. Tantawi, "Cpu demand for web serving: Measurement analysis and dynamic estimation," *Performance Evaluation*, 2008.

[29] D. Westermann and J. Happe, "Towards performance prediction of large enterprise applications based on systematic measurements," in *WCOP*, 2010.

[30] M. Courtois and M. Woodside, "Using regression splines for software performance analysis," in *WOSP*, 2000.

[31] D. Carrera, J. Guitart, J. Torres, E. Ayguade, and J. Labarta, "Complete instrumentation requirements for performance analysis of Web based technologies," in *ISPASS*, 2003.

[32] A. Mos and J. Murphy, "A Framework for Performance Monitoring, Modelling and Prediction of Component Oriented Distributed Systems," in *WOSP*, 2002.

[33] M. Rohr, A. van Hoorn, S. Giesecke, J. Matevska, W. Hasselbring, and S. Alekseev, "Trace-context sensitive performance profiling for enterprise software applications," in *SIPEW*, 2008.

[34] G. Denaro, A. Polini, and W. Emmerich, "Early Performance Testing of Distributed Software Applications," in *WOSP*, 2004.

[35] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance prediction of Component-Based Applications," *J. Syst. Softw.*, 2005.