Automated Extraction of Palladio Component Models from Running Enterprise Java Applications

Fabian Brosig, Samuel Kounev, Klaus Krogmann
Software Design and Quality Group
Universität Karlsruhe (TH), Germany
{brosig, skounev, krogmann}@ipd.uni-karlsruhe.de

ABSTRACT

Nowadays, software systems have to fulfill increasingly stringent requirements for performance and scalability. To ensure that a system meets its performance requirements during operation, the ability to predict its performance under different configurations and workloads is essential. Most performance analysis tools currently used in industry focus on monitoring the current system state. They provide low-level monitoring data without any performance prediction capabilities. For performance prediction, performance models are normally required. However, building predictive performance models manually requires a lot of time and effort. In this paper, we present a method for automated extraction of performance models of Java EE applications, based on monitoring data collected during operation. We extract instances of the Palladio Component Model (PCM) - a performance meta-model targeted at component-based systems. We evaluate the model extraction method in the context of a case study with a real-world enterprise application. Even though the extraction requires some manual intervention, the case study demonstrates that the existing gap between low-level monitoring data and high-level performance models can be closed.

1. INTRODUCTION

For effective performance management during system operation, performance models are highly beneficial. They enable a proactive and flexible run-time resource management by predicting the system performance (e.g., service response times and resource utilization) for anticipated usage and configuration scenarios. For example, if one observes a growing customer workload and assumes a steady workload growth rate, a performance model can help to determine when the system would reach its saturation point. This way, system operators can react to the changing workload before the system has failed to meet its performance objectives, thus avoiding a violation of service level agreements (SLAs). Furthermore, performance models can be exploited to implement autonomic performance and resource management, e.g., automatic system reconfiguration at run-time to ensure that SLAs are continuously satisfied and system resources are utilized efficiently.

Unfortunately, building a predictive performance model manually requires a lot of time and effort [9]. The model must be designed to reflect the abstract system structure and capture its performance-relevant aspects. In addition, model parameters like service resource demands or system configuration parameters have to be determined. Current performance analysis tools used in industry mostly focus on profiling and monitoring transaction response times and resource consumption. The tools often provide large amounts of low-level data while important information needed for building performance models is missing, e.g., service resource demands. Given the costs of building performance models, techniques for automatic extraction of models based on observation of the system at run-time are highly desirable. During system development, such models can be exploited to evaluate the performance of system prototypes. During operation, an automatically extracted performance model can be applied for efficient and performance-aware resource management.

In this paper, we present a method for automated extraction of performance models of enterprise Java applications during operation. The target platform we consider is the Oracle WebLogic Server (WLS) which is currently one of the leading Java EE application servers available on the market [19]. Our model extraction method is based on monitoring data collected during operation using state-of-the-art, off-the-shelf monitoring tools available for the considered platform. In particular, we use the WebLogic Diagnostics Framework (WLDF) shipped as part of WLS. We implemented the proposed extraction method in a tool prototype and evaluated its effectiveness in the context of a case study with a beta-version of the successor of the SPECjAppServer2004 benchmark which we will refer to as *SPECjAppServer2004_Next* [1].

As a performance model, we selected the Palladio Component Model (PCM, [2]). PCM is a high-level UML-like design-oriented modeling approach that captures the perfor-

ROSSA 2009, October 19, 2009 - Pisa, Italy

[1]SPECjAppServer2004 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2004_Next 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 SPECjAppServer2004 is located at `http://www.spec.org/osg/jAppServer2004`.

mance-relevant aspects of component-based software architectures. For performance prediction, PCM supports low-level mathematical performance models like, e.g., queueing networks. Mathematical models are automatically derived from the high-level model, which makes the PCM easy to use for software developers. We consider component-level performance models such as PCM since they allow per-component or per-service performance predictions based on detailed usage behavior specifications. This is required for the performance management of modern enterprise systems where SLAs need to be managed in dynamic architectures where, e.g., services are newly composed and deployed on-the-fly.

The rest of this paper is organized as follows. In Section 2, we provide some background on the technologies and tools we use. We then describe the model extraction method in Section 3 followed by a presentation of the case study in Section 4. Finally, we review related work in Section 5 and conclude the paper in Section 6.

2. BACKGROUND

We start by describing the technology platform, performance models and monitoring tools our model extraction method is based on.

2.1 Java Platform, Enterprise Edition

The Java Platform, Enterprise Edition (Java EE) is one of the most popular middleware architectures for enterprise applications. A significant part of Java EE is the Enterprise JavaBean (EJB) Architecture. It is a server-side framework for component-based enterprise Java applications. The EJB 3.0 specification supports two types of so-called beans: i) *session beans* that encapsulate business logic in the form of services executed through synchronous invocation and ii) *message-driven beans* (MDBs) that encapsulate business logic executed to process asynchronous messages sent through a message-oriented middleware. Session beans can be *stateful* or *stateless*, depending on whether they maintain a conversational state, i.e., a conversational context between client and bean, or not. For persisting data, the current version of Java EE proposes the Java Persistence API (JPA).

2.2 Palladio Component Model

The Palladio Component Model (PCM) is a domain-specific modeling language for describing performance-relevant aspects of component-based software architectures [2]. It has been used successfully to model a number of different systems of variable size and complexity, e.g., [2, 13, 15]. The PCM comes with a tool set facilitating the creation and analysis of PCM performance models [1].

Given that PCM is targeted at component-based architectures, its focus is on modeling the performance-influencing factors of software components. In order to capture the time behavior and resource consumption of a component, four factors have to be taken into account. Obviously, the component's implementation affects its performance. Additionally, the component may depend on external services whose performance has to be considered. Furthermore, both the way the component is used, i.e., the usage profile, and the execution environment in which the component is running have to be taken into consideration. Since these performance-influencing factors are explicitly modeled in PCM, the impact of changing them can be predicted.

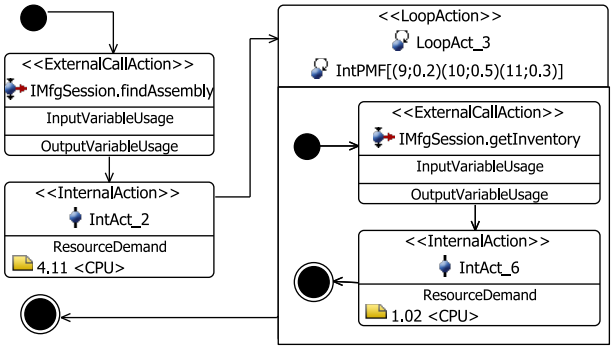


Figure 1: Example RDSEFF for the provided service `scheduleWorkOrder`.

To support modeling large applications and concurrently involve multiple developers, the PCM's model is split 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 as a so-called *Resource Demanding Service Effect Specification (RDSEFF)*, which we describe in more detail below. 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. PCM allows modeling processing resources like, e.g., CPUs and disk drives. 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 run-time, the frequency (workload intensity), the order and the input parameters passed to them.

In the following, we provide a brief overview of the modeling capabilities of RDSEFFs. An RDSEFF describes the performance-relevant internal behavior of a provided component service in an abstract fashion. The goal is to capture the control flow and resource consumption of the service depending on the input parameters passed to it.

The service behavior is characterized by a sequence of performance-relevant actions. Calls to required services are modeled using so-called `ExternalCallActions`, whereas internal computations within the component are modeled using `InternalActions`. Control flow actions like `LoopAction` or `BranchAction` are used only when they affect calls to required services, e.g., if a required service is called within a loop; otherwise, the loop is captured as part of an `InternalAction`. `LoopActions` and `BranchActions` are characterized with loop iteration numbers and branch probabilities, respectively. `InternalActions`, on the other hand, are characterized with resource demand specifications.

As an example, consider a component `WorkOrderSession` that provides the interface `IWorkOrderSession` and requires the interface `IMfgSession`. The RDSEFF in Figure 1 describes the service `IWorkOrderSession#scheduleWorkOrder` implemented by the component. Starting with an `ExternalCallAction` to service `IMfgSession#findAssembly`, the service executes an `InternalAction` requiring CPU resources followed by a `LoopAction` whose number of iterations is spec-

ified as a Probability Mass Function (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. The loop body contains an `ExternalCallAction` to service `IMfgSession#getInventory` and a further `InternalAction` that is enriched by a CPU demand annotation. While PCM supports generic CPU units, in the context of this paper, a demand of 1 CPU unit is understood as a demand of 1 millisecond CPU time.

Beyond the example, PCM RDSEFFs allow to model dependencies between input parameters (parameters passed upon service invocation and parameters returned from external service calls) and control flow (number of loop iterations and branch selection) and resource demands [12]. For example, a `LoopAction` can be executed for each element of a list provided as input parameter and an `InternalAction` can have a resource demand depending on the list's length.

2.3 WebLogic Diagnostics Framework

WebLogic Diagnostics Framework (WLDF) is a monitoring and diagnostics framework that enables collecting and analyzing diagnostic data for a running Oracle WebLogic Server (WLS). The diagnostic data provides insight into the run-time performance of the server and allows isolating, diagnosing and resolving potential issues with the server operating environment [21]. 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. For instance, the amount of free Java heap memory or the number of pooled bean instances of a specific stateless session bean can be monitored.

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 so-called *diagnostic context id*. The diagnostic context id uniquely identifies the request that generated the event allowing to trace individual requests as they traverse the system.

3. MODEL EXTRACTION METHOD

We now present our model extraction method and discuss the way we have implemented it in our tool prototype. For further details on the method and its implementation we refer to [4].

Figure 2 illustrates the model extraction process. The three main steps of the process are: i) extraction of the application architecture, ii) extraction of performance-relevant control flow and iii) extraction of resource demands. The extraction is based on monitoring data collected during operation.

Note that we focus on the EJB 3.0 Component Model and do not consider the web tier (Servlets, Java Server Faces, etc). We concentrate on session beans and do not consider asynchronous messaging respectively MDBs. We assume that the Java Persistence API (JPA) is used for persisting entity data. Finally, we are currently considering a single

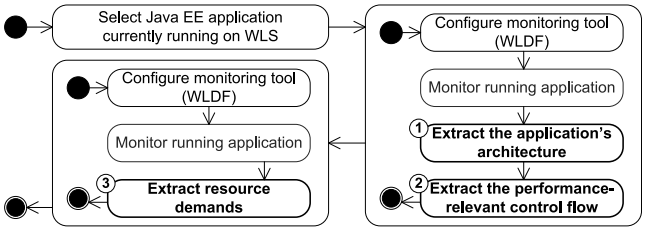


Figure 2: Steps of the model extraction method.

WLS instance, however, the approach we propose can be extended to multiple instances.

3.1 Extracting the Application's Architecture

In the first step of the extraction process, the effective application architecture is extracted. The latter refers to the set of components and connections between components that are effectively used during operation. Besides considering each EJB as an individual component according to the EJB 3.0 Component Model, the method also supports grouping multiple EJBs to higher-level components, e.g., considering EJBs contained in the same Java package as a single component.

The components and connections are identified on the basis of trace data reflecting the observed call paths during execution. With the help of the *diagnostic context id* provided by the WLDF instrumentation engine, individual requests can be traced as they traverse the system. For instance, if WLDF is configured to monitor entries and exits of EJB business methods, each call to a business method triggers the generation of an event record at the beginning and end of the method. Grouping the event records by the diagnostic context id and sorting the groups by the *event record id* leads to traces of individual requests [3]. 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.

3.2 Extracting Performance-Relevant Control Flow

In PCM, the performance-relevant control flow of a component service is modeled as an RDSEFF. Given a component service, in order to extract an RDSEFF, one first has to identify the performance-relevant actions of the service. We assume these performance-relevant actions to be known, given that they can be identified using existing approaches [6, 9].

We concentrate on monitoring the effective control flow, extract probabilities of different call paths in contrast to extracting explicit parametric dependencies. RDSEFFs distinguish internal component computations (`InternalActions`), calls to external services (`ExternalCallActions`) and control flow constructs between external service calls (`LoopActions` or `BranchActions`). In order to make it possible to monitor the performance-relevant actions of a service, we assume that such actions are moved to separate methods. Each method representing a performance-relevant action is named according to a naming scheme that makes the control flow of the respective component service explicit. That requirement arises from the lack of tool support for in-method

```
class A {
  @EJB private BLocal session;
  ...
  public int methodA(int a) {
    /* compute something */
    int result = 0;
    for (int i = 0; i < a; i++) {
      /* compute something */
      /* call external service */
      result += session.methodB();
    }
    return result;
  }
}
```

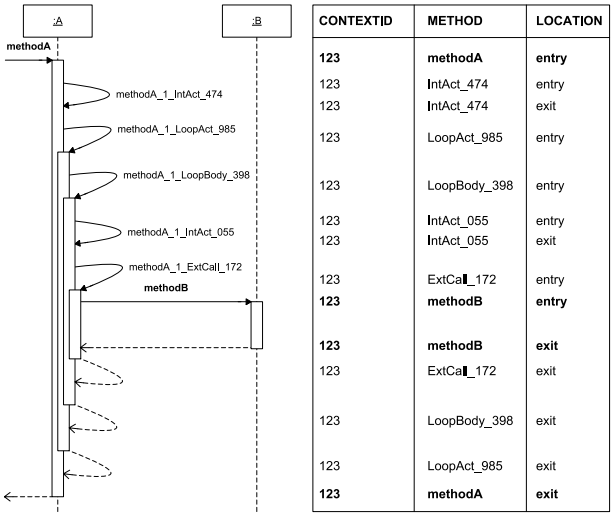
⇓

```
class A {
  @EJB private BLocal session;
  ...
  public int methodA(int a) {
    /* extracted internal action */
    methodA_1_IntAct_474(a);
    /* extracted loop action */
    int result = 0;
    methodA_1_LoopAct_985(result, a);
    return result;
  }
  public void methodA_1_IntAct_474(int a){
    /* compute something */
  }
  public int methodA_1_LoopAct_985
                          (int result, int a) {
    for (int i = 0; i < a; i++) {
      /* extracted loop body */
      result += methodA_1_LoopBody_398();
    }
    return result;
  }
  public int methodA_1_LoopBody_398() {
    /* extracted internal action */
    methodA_1_IntAct_055();
    /* extracted external call action */
    return methodA_1_ExtCall_172();
  }
  public void methodA_1_IntAct_055() {
    /* compute something */
  }
  public int methodA_1_ExtCall_172() {
    /* call external service */
    return session.methodB();
  }
}
```

Listing 1: Example: Code refactoring.

instrumentation. Current instrumentation tools including WLDF support only method-level instrumentation. They do not support instrumentation at custom locations other than method entries and exits. As part of our future work, we intend to relax the above assumption. Given that performance-relevant actions are moved to separate methods, WLDF can be configured to monitor the performance-relevant control flow. Based on the generated trace data, RDSEFFs can be extracted. For `LoopActions`, the number of loop iterations is extracted as PMF. For `BranchActions`, branch transition probabilities are extracted.

Listing 1 shows an example that illustrates the required refactoring. The code fragment on the top shows the implementation of a component service `A#methodA`. It consists of some internal computations and a call to an external service in another component within a loop. The code fragment on the bottom shows the refactored implementation. The `InternalActions`, the `LoopAction` and the `ExternalCallAction` have been moved into separated methods. The newly introduced methods are named according to the naming scheme mentioned above.



CONTEXTID	METHOD	LOCATION
123	**methodA**	**entry**
123	IntAct_474	entry
123	IntAct_474	exit
123	LoopAct_985	entry
123	LoopBody_398	entry
123	IntAct_055	entry
123	IntAct_055	exit
123	ExtCal|_172	entry
123	**methodB**	**entry**
123	**methodB**	**exit**
123	ExtCal|_172	exit
123	LoopBody_398	exit
123	LoopAct_985	exit
123	**methodA**	**exit**

Figure 3: Call path tracing with WLDF.

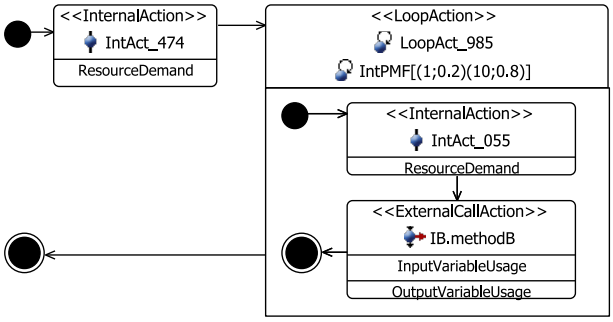


Figure 4: Extracted RDSEFF of `A#methodA`.

Figure 3 illustrates a call to `A#methodA` as an UML sequence diagram. The rows of the table on the right represent WLDF event records that are generated when the method is executed. The event record list is ordered by the event record id (which is not shown in the table). Given that the event records belong to one request, the diagnostic context id is the same for all records. Each event record contains information about the method name and the location where the record generation was triggered. By means of this information, the control flow can be extracted [3].

Figure 4 shows the RDSEFF extracted from the trace data generated by WLDF. The RDSEFF consists of an `InternalAction` followed by a `LoopAction` containing a further `InternalAction` and an `ExternalCallAction`. The loop iteration number of `LoopAction` `LoopAct_985` is extracted as a PMF. The PMF states that the loop iterates one time with a probability of 20% and ten times with a probability of 80%. Notice that the shown RDSEFF is not annotated with resource demands. The resource demand specifications are added in the next step.

3.3 Extracting Resource Demands

In PCM, the resource demands are specified as annotations attached to the `InternalActions` of RDSEFFs. Determining resource demands involves identification of the resources used by services 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. The challenge is to extract the resource demands from a running system that is not available for testing in a controlled environment.

For the resource demand extraction we do not use method instrumentation where single measurements result in single event records. Instead, we use method instrumentation where single measurements are aggregated online to obtain statistics on individual methods like, e.g., method invocation counts or average method response times. Given that single measurements are not required to be stored, the latter technique's overhead is an order of magnitude lower than the event record triggering approach. The downside of the data aggregation is the loss of information about individual measurements. Thus, resource demand estimation approaches that require individual measurement observations like, e.g., statistical regression are not applicable.

We investigated two approaches for estimating the resource demands of individual `InternalActions`: i) approximate resource demands with measured response times, ii) estimate resource demands based on measured utilization and throughput data. In the first approach, we measure the response time of an `InternalAction` and use the average response time as an estimate of the resource demand. This approach is simple, however, it only works if the considered resource dominates the overall response time of the `InternalAction`. Furthermore, the times spent queueing for the resource have to be insignificant compared to the actual processing time at the resource. Thus, the first approach is only applicable during phases of low resource utilization, i.e., typically 20%.

For cases of higher resource utilization, we propose the second approach. It uses measured utilization and throughput data to estimate resource demands. During an observation period of fixed length T, we obtain the average utilization ($\mathcal{U}_{i,r}$) of resource r due to executing `InternalAction` i and the total number of times $\mathcal{C}_i$ that `InternalAction` i is executed. For each `InternalAction` i, its resource demand ($\mathcal{D}_{i,r}$) for resource r can be estimated as the quotient of utilization and throughput:

$$\mathcal{D}_{i,r} = \frac{\mathcal{U}_{i,r}}{\mathcal{C}_i/T} = \frac{\mathcal{U}_{i,r} \cdot T}{\mathcal{C}_i},$$

which is often referred to as *Service Demand Law* [17]. While the $\mathcal{C}_i$'s can be easily measured, determining $\mathcal{U}_{i,r}$ is a challenge. The total utilization of a resource, e.g., the overall CPU utilization, can be measured using operating system tools. However, what needs to be determined is the fraction of the total resource utilization that is caused by a distinct action (`InternalAction` i). This is a complex task, given that we observe the system during operation where usually many actions are running at the same time.

We do not use profilers for that purpose, since profiling imposes too much overhead and is normally too intrusive. Instead, we partition the measured total resource utilization with response time ratios that are weighted with throughput data. Concerning one individual resource r, let $i_1, \ldots, i_n$ be the list of `InternalActions` that stress the considered resource during the observation period, $\mathcal{R}_{i_1}, \ldots, \mathcal{R}_{i_n}$ their measured average response times, $\mathcal{C}_{i_1}, \ldots, \mathcal{C}_{i_n}$ their invocation counts and finally $\mathcal{U}_r$ the total resource utilization of resource r. We estimate the average utilization $\mathcal{U}_{i_k,r}$ ($1 \leq k \leq n$) of resource r due to executing `InternalAction` i_k as:

$$\mathcal{U}_{i_k,r} = \mathcal{U}_r \cdot \frac{\mathcal{R}_{i_k} \cdot \mathcal{C}_{i_k}}{\sum_{j=1}^{n} \mathcal{R}_{i_j} \cdot \mathcal{C}_{i_j}}.$$

Thus, we partition the total resource utilization $\mathcal{U}_r$ using weighted response time ratios. This partitioning is based on the assumption that the measured response times of the `InternalActions` are proportional (at least approximately) to their resource demands which again implies that the considered resource dominates the response times of the `InternalActions`. However, in contrast to the first approach, the resource demands can be estimated during an observation period under medium to high load, i.e., resource utilization between 50%-80%.

When there are `InternalActions` that are not dominated by a single resource, i.e., if there is more than one resource having a significant impact on the `InternalAction` processing times, the second approach can be generalized regarding the granularity of measured times. If one can measure processing times of individual execution fragments, so that the measured times of these fragments are dominated by single resources, again the second approach can be applied.

We adopt the strategy of the second approach in order to apportion CPU resource demands among the database server (DBS) and the application server (in our case, WLS). We apply the following approximation: Looking at an EJB transaction, the transaction consists of a working phase and a commit phase. Processing times in the working phase are apportioned to the WLS CPU. Processing times in the commit phase are apportioned to the DBS CPU. The assumption is that the length of the working phase is proportional to the WLS CPU resource demand whereas the length of the commit phase is proportional to the DBS CPU resource demand. The approximation implies that, when estimating resource demands for the DBS CPU, we overestimate database writes and ignore database reads. However, given that JPA implements internal entity caches that reside on the WLS instance, database reads are not expected to dominate the overall application performance. Note that the adequacy of this approximation depends on the type of application considered. While processing times of working phases normally can be directly measured, in general, processing times of transaction commit phases are not accessible. For instance, in EJB 3.0, there are transactions that are managed by the EJB container. In order to measure such a middleware service, we compute the difference between method response times *inside* the method and response times *outside* the method. For the former case, we introduce time sensors after method entry and before method exit. For the latter case, the time sensors are placed around the method call at the callee. In this way, we obtain approximated processing times of transaction commit phases .

In Section 4, we evaluate and compare the proposed approaches for estimating the resource demands.

4. EVALUATION

To evaluate the model extraction method, we implemented it as part of a tool prototype and applied it to a case study on a representative Java EE application. The application we consider is the SPECjAppServer2004_Next benchmark application. We start with a brief overview of the benchmark.

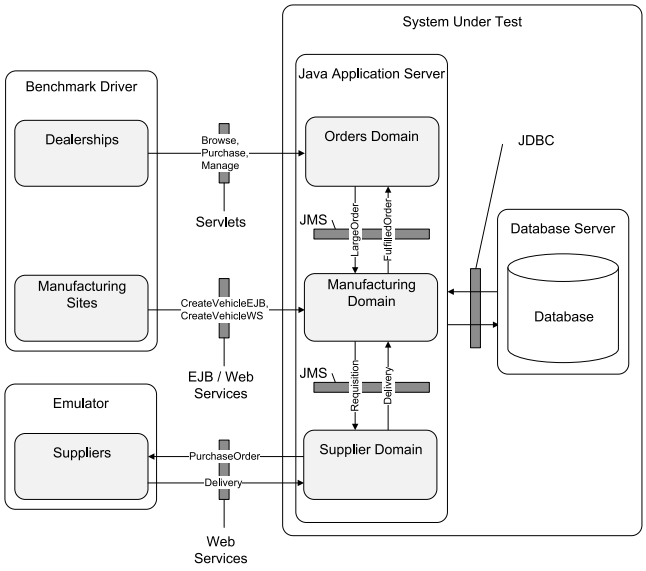


Figure 5: SPECjAppServer2004_Next architecture.

4.1 SPECjAppServer2004_Next Benchmark

SPECjAppServer2004_Next is a beta version of the successor of the SPECjAppServer2004 benchmark. It 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.

To give an example of the business logic implemented by the benchmark, consider a car dealer that places a large order with the automobile manufacturer. The large order is sent to the manufacturing domain which schedules a *work order* to manufacture the ordered vehicles. In case some parts needed for the production of the vehicles are depleted, a request to order new parts is sent to the supplier domain. The supplier domain selects a supplier and places a purchase order. When the ordered parts are delivered, the supplier domain contacts the manufacturing domain and the inventory is updated. Finally, upon completion of the work order, the orders domain is notified.

Figure 5 depicts the architecture of the benchmark as described in the benchmark documentation. The benchmark application is divided into three domains: orders domain, manufacturing domain and supplier domain. 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. 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). As shown on the diagram, the system under test spans both the Java application server and the database server. The emulator and the benchmark driver have to run outside the system under test so that they do not affect the benchmark results.

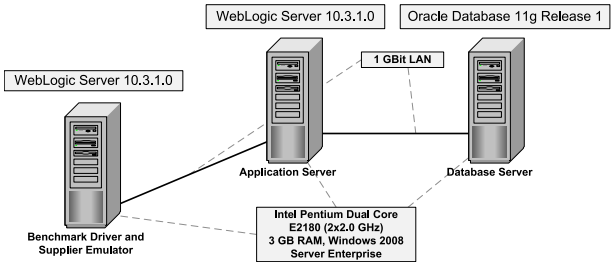


Figure 6: Experimental environment.

The benchmark driver executes four 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. A manufacturer may place *work orders* for manufacturing vehicles.

We selected the SPECjAppServer2004_Next benchmark application as a basis for our case study since it models a representative, state-of-the-art system. Previous versions of the benchmark have already been successfully applied for research purposes [10, 11].

4.2 Experimental Environment

We installed the benchmark in the system environment depicted in Figure 6. The benchmark application was deployed in Oracle WebLogic Server (WLS). WLS was configured to use a file-based store to persist event records generated by WLDF. As a database server (DBS) for persisting JPA entity data, Oracle Database 11g was used. The benchmark driver and the supplier emulator were running on a separate machine. The three machines all have Intel Pentium Dual Core CPUs, 3 GB of RAM and are connected using a 1 GBit Ethernet.

4.3 Case Study

We conducted a case study with the goal to evaluate the accuracy of PCM models of the SPECjAppServer2004_Next application extracted by means of our tool prototype. We focused on the main part of the benchmark application, the manufacturing domain. The EJBs are considered as individual components. The resources we considered were the CPUs of the system under test, i.e., the CPU of the WLS instance (WLS CPU) and the CPU of the database server (DBS CPU). We considered several different scenarios varying, on the one hand, the type of workload under which the PCM models are extracted, on the other hand, the type of workload for which performance predictions are made. By workload here we mean the operation mix and operation throughput. Thus, the operation throughput determines the workload intensity. Note that we considered only open workloads. For each scenario, two PCM models were considered:

- A PCM model in which resource demands were approximated with the measured response times, denoted as *Model A*.
- A PCM model in which resource demands were estimated based on the utilization and throughput data, denoted as *Model B*.

The extracted PCM models are validated by comparing the model predictions with measurements on the real system. While the model extraction requires instrumentation, the measurements used for the validation were made without any instrumentation. For the PCM model predictions we used a queueing-network based simulation [2]. As performance metrics, we considered the average response times of business operations as well as the average utilization of the WLS CPU and the DBS CPU. For each scenario and the two corresponding PCM models, we first predicted the performance metrics for low load conditions ($\approx$ 20% WLS CPU utilization), medium load conditions ($\approx$ 40% and $\approx$ 60%) and high load conditions ($\approx$ 80%) and then compared them with measurements. Note that all predictions of a scenario are derived from one Model A and one Model B. We considered the following specific scenarios:

- Scenario 1: The workload consisted of the business operation `ScheduleWorkOrder`. The resource demands for Model A were extracted during light system load. The resource demands for Model B were extracted during high system load. The goal of this scenario is to evaluate the accuracy of the model predictions for a fixed operation mix and varying workload intensity. The model predictions were compared against measurements under low, medium and high load conditions.
- Scenario 2a: This scenario is similar to the above with the exception that it used the business operation `CreateVehicle`. `CreateVehicle` is a larger operation consisting of several sub-operations one of them being the `ScheduleWorkOrder` operation considered in Scenario 1.
- Scenario 2b: We used the PCM models extracted in Scenario 2a to conduct performance predictions for the workload considered in Scenario 1. This is possible because the workload considered in Scenario 1 is a subset of the workload considered in Scenario 2a. The goal is to evaluate the applicability of our approach, when performance predictions are made based on a model that was extracted during a period when the respective workload for which predictions are made was run concurrently with other workloads.

4.3.1 Scenario 1

In this scenario, the resource demands for Model A were extracted during a steady state time of 1020 sec with an average WLS CPU utilization of 12%. The resource demands for Model B were extracted during a steady state time of 1020 sec and a WLS CPU utilization of 81%. For both models, WLS CPU and DBS CPU resource demands were extracted for four internal actions. To validate the model for different throughput levels, we compare the predicted server utilization and operation response times against measurements on the real system which are made during a steady state time of 1020 sec. We used the operation `ScheduleWorkOrder` for both the workload during model extraction and to validate the extracted models.

Figure 7 shows the results of the model validation. Predictions based on Model B are slightly better than predictions based on Model A. For the highest considered throughput level, both models deliver no performance predictions. This is because the system as represented by the models is not able to sustain the injected load since the WLS CPU utilization is overestimated to be 100%. Both models overestimate the WLS CPU utilization while underestimating the DBS CPU utilization. The modeling prediction error for CPU utilization is mostly about 20%. The modeling prediction error for response times increases with the throughput level. The higher the CPU utilization, the bigger the impact of the overestimated WLS CPU demands on the predicted response times. We assume that the overestimation of the WLS CPU demands is due to the instrumentation overhead (about 15%) during resource demand extraction.

4.3.2 Scenario 2a

In this scenario, the resource demands for Model A were extracted during a steady state time of 1140 sec with an average WLS CPU utilization of 9%. The resource demands for Model B were extracted during a steady state time of 1140 sec and a WLS CPU utilization of 75%. For both models, nine internal actions were annotated with WLS CPU demand estimations, DBS CPU demands were estimated for four internal actions. Measurements on the real system were made during a steady state time of 1140 sec. We used the operation `CreateVehicle` for both the workload during model extraction and to validate the extracted models.

Figure 8 shows the results for Scenario 2a. For both Model A and Model B, the prediction error is mostly about 20%. In the experiment with the highest throughput, it is the overestimated WLS CPU utilization that causes a response time deviation of 47%. For all throughput levels, the DBS CPU utilization is overestimated by Model A. Thus, in this scenario, the approximation of DBS CPU demands with the measured response times of transaction commit phases does not match as well as in Scenario 1. Apart from that, the results in this scenario are similar to the results in Scenario 1.

4.3.3 Scenario 2b

In this scenario, the PCM models extracted in Scenario 2a are validated with a different workload. The difference between the validations in Scenario 2a and Scenario 2b is in the usage profile we used as input for the performance prediction and the measurements we compared the predictions against. We used the operation `CreateVehicle` for the workload during model extraction and the operation `ScheduleWorkOrder` to validate the extracted models.

Figure 9 shows the results for Scenario 2b. Note that for the same reasons as in Scenario 1, both models deliver no performance predictions for the highest considered throughput level. As expected, this scenario's Model A performs similar to Model A of Scenario 1. Concerning Model B, while the performance predictions for the WLS CPU utilization are about 5% higher than the predictions of Scenario 1, the DBS CPU utilization is underestimated with an error of about 50%. Thus, partitioning DBS CPU utilization via response time ratios of transaction commit phases requires

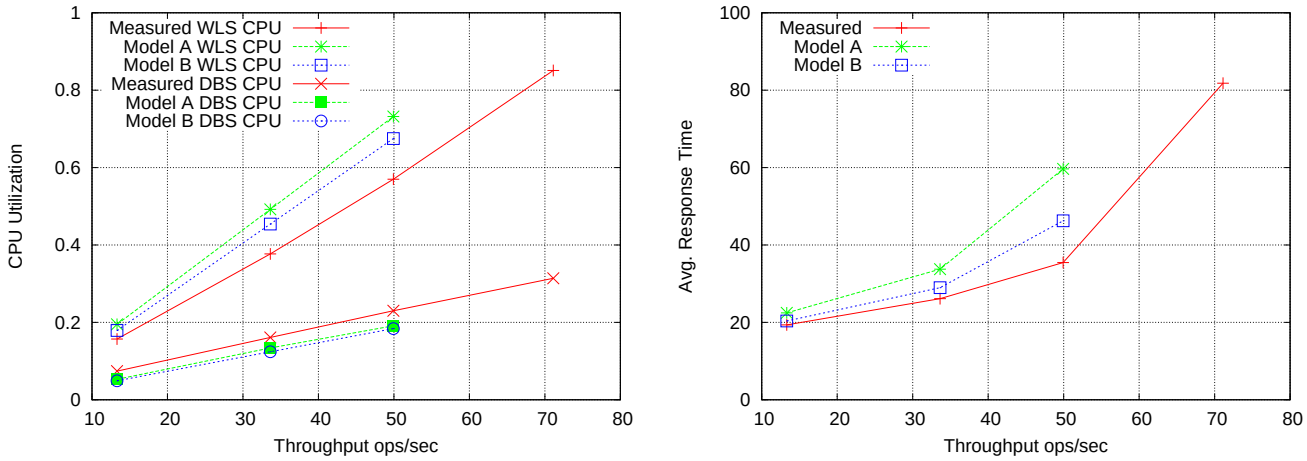


Figure 7: Scenario 1: Validation of the `ScheduleWorkOrder` performance model.

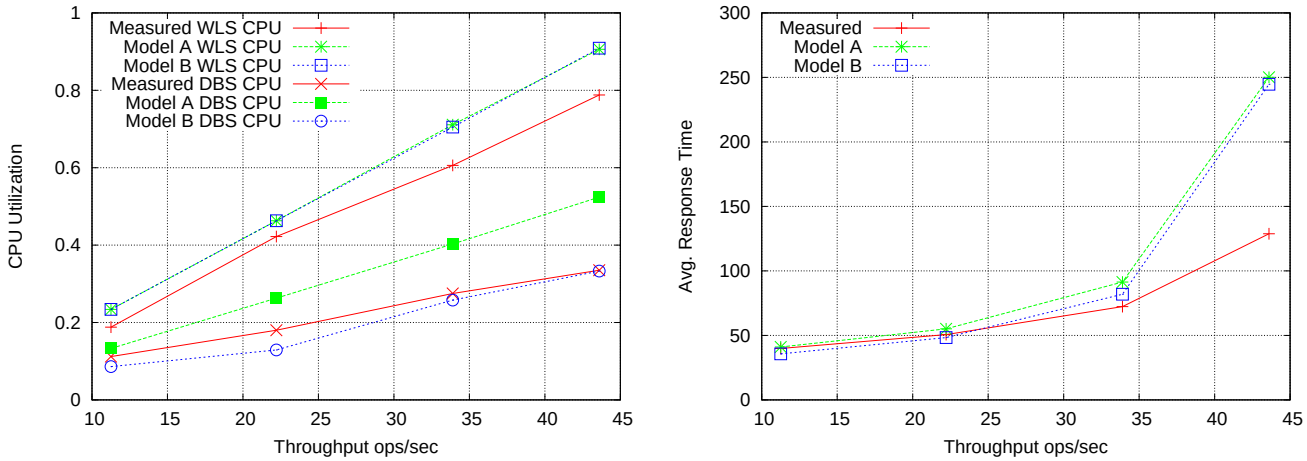


Figure 8: Scenario2a: Validation of the `CreateVehicle` performance model.

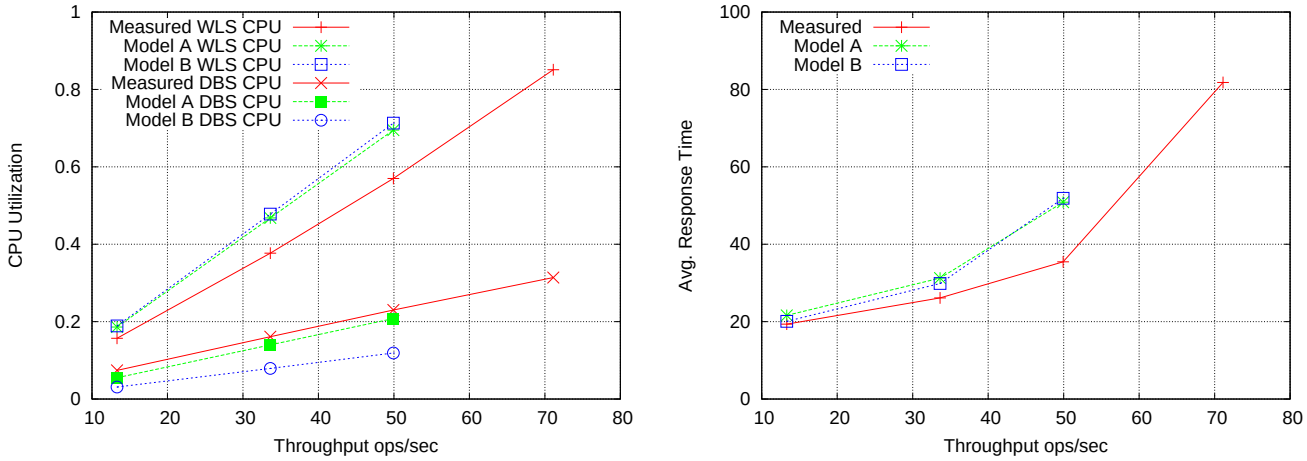


Figure 9: Scenario2b: Validation of the `CreateVehicle` performance model.

further investigation while partitioning WLS CPU utilization via response time ratios of internal actions appears to be appropriate. Concerning the estimation of DBS CPU demands, we expect that the partitioning will be improved if we take the I/O demands of the commit phases into account.

4.3.4 Summary

With the automatically extracted PCM models, we were able to predict CPU utilization and operation response times with an error of mostly about 20% – 30%. The lower the concurrency level in the considered usage profile, the better the resource demand estimations were. In cases of high concurrency and high system load, we partition the measured resource utilization among different actions which introduces a loss of precision. In Scenario 2b, this issue was observed especially for estimated DBS CPU demands.

However, further calibrations are possible. For instance, the estimated WLS CPU demands can be calibrated by taking the instrumentation overhead into account. The estimated DBS CPU demands can be calibrated by considering also the I/O demands of the transactions. To observe I/O demands we can use WLDF to monitor SQL queries sent to the DBS. However, this comes at the expense of a significantly higher overhead.

5. RELATED WORK

The approach presented in this paper is related to multiple extraction approaches: a) trace-based approaches, b) runtime monitoring, and c) 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. [7] extract performance models in the scope of "Trace-Based Load Characterization (TLC)". The target model of TLC is not component-based as ours. The "effective" architecture of a software system is extracted by Israr et al. in [8] using pattern matching on trace data. Thereby, their approach can differentiate between asynchronous, blocking synchronous, and forwarding communication. Opposed to our approach, the components Israr et al. support have no explicit control flow. TLC and Israr et al. use Layered Queueing Networks (LQNs) as the target performance model and are limited to LQN structures. Thus, PCM features such as stochastic characterizations of loop iteration numbers or branch probabilities are not supported.

Dynatrace Diagnostics [20] is an industrial tool for performance management. It traces transactions for applications deployed in distributed, heterogeneous .NET and Java environments. Besides providing a call tree, it also monitors method argument values and provides information about the system's resource utilization. An explicit architecture model, components or behavior are not extracted.

Briand et al. [3] extract UML sequence diagrams from trace data which is obtained by aspect-based instrumentation.

Run-time monitoring. Carrera et al. [5] present an automatic monitoring framework covering the operation system, JVM, middleware and application level. After initially defining performance objectives, it automatically traces the execution. Performance data (including resource consumption) allow primarily hotspot and bottleneck detection.

The Compas tool [18] is an adaptive monitoring and performance management framework capable of extracting data in real-time from a running application which generates system behavior models. It addresses performance issues related to the EJB layer in Java EE applications. Compas instruments through a proxy layer which encapsulates each component. The instrumentation is transparent like in our approach and assumes synchronous invocations, i.e., it does not support MDBs. In contrast to PCM, Compas is not explicitly dealing with components or context information on bound component.

Zheng et al. [22] focus on run-time monitoring and online prediction of performance. Their models are reverse engineered by the use of Kalman filters, however, they are not component-based. Like in our approach, they are not required to directly monitor resource demands of interest but can estimate them based on known (or easily measurable) metrics such as response time and resource utilization.

PCM extraction. Several approaches are concerned with the extraction of PCM instances from code. The tool Java2PCM [9] extracts component-level control flow from Java code. The extracted behavior model matches the control flow structure results from our approach. Krogmann et al. [14, 16] extract behavior models via static and dynamic analysis but do not focus on extracting timing values during system operation, instead they abstract from concrete timing values (Java bytecode operations) to enable cross-platform prediction. Their approach relies on own instrumentations and extracts control and data flow by means of machine learning techniques.

The ArchiRec [6] approach for static code analysis complements the approach presented in this paper. ArchiRec provides component boundaries as input for identifying internal actions and component calls within our approach. In ArchiRec, components and their interfaces are identified within Java and EJB code through metrics calculation and subsequent hierarchical clustering.

6. CONCLUDING REMARKS

In this paper, we presented an approach for automatically extracting performance models on the basis of monitoring data collected during operation and available from off-the-shelf monitoring tools. The proposed extraction method was implemented in a tool prototype. The extracted performance models allow performance predictions of Java EE applications using the PCM.

We evaluated our approach in the context of a case study with a real-world enterprise application, a beta-version of the successor of the SPECjAppServer2004 benchmark, which we deployed in a realistic system environment. The extracted performance models were used to derive performance predictions that were compared to actual measurements. The prediction error was between 20 to 30 percent. However, further calibrations are possible.

Even though the performance model extraction requires some intervention, the prototype we implemented shows that the existing gap between low-level monitoring data and high-level performance models can be closed. Most available performance analysis tools monitor the current system state. We propose techniques that are capable to predict the system's performance for different configurations and workloads.

In the future we will extend our work on the model extraction into several directions. In the short term, we will extend the extraction method to consider point-to-point asyn-

chronous messaging, i.e., MDBs. We plan to also consider the web tier of Java EE, e.g., Servlets and Java Server Pages. With these enhancements, we will extend the case study to the entire benchmark application, then using the released version of the successor of SPECjAppServer2004. Furthermore, we will implement the improvements on the extraction of resource demands we proposed in Section 4.3.4 and investigate how to handle the issue of extracting resource demands for load-dependent resources like CPUs implementing dynamic frequency scaling technologies. To further automate the model extraction we will investigate how to overcome the current component code refactoring.

The work will be continued in the context of the *Descartes* project[2] where online performance models generated dynamically from the evolving system configuration are exploited for autonomic performance and resource management.

7. REFERENCES

[1] Palladio Component Model. Last visit: 2009-07-23. `http://www.palladio-approach.net/`.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[3] L. C. Briand, Y. Labiche, and J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Trans. Software Eng.*, 32(9):642–663, 2006.

[4] F. Brosig. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. Master's thesis, Universität Karlsruhe (TH), Germany, 2009.

[5] D. Carrera, J. Guitart, J. Torres, E. Ayguade, and J. Labarta. Complete instrumentation requirements for performance analysis of Web based technologies. *Proc. 2003 IEEE Int'l Symposium on Performance Analysis of Systems and Software*, pages 166–175, 2003.

[6] L. Chouambe, B. Klatt, and K. Krogmann. Reverse Engineering Software-Models of Component-Based Systems. *12th European Conf. on Software Maintenance and Reengineering*, pages 93–102, 2008. IEEE Computer Society.

[7] C. Hrischuk, C. Murray Woodside, and J. Rolia. Trace-based load characterization for generating performance software models. *IEEE Trans. Software Eng.*, 25(1):122–135, 1999.

[8] T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software, 5th Int'l Workshop on Software and Performance*, 80(4):474–492, 2007.

[2]The Descartes project is funded by the German Research Foundation within the Emmy Noether Programme for a period of five years. The project aims to develop a set of novel methods and techniques for building next generation autonomic and self-aware enterprise systems. The latter will be aware of their own performance and will automatically adapt as the environment evolves ensuring that system resources are utilized efficiently and performance requirements are continuously satisfied.

[9] T. Kappler, H. Koziolek, K. Krogmann, and R. H. Reussner. Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. *Software Engineering 2008*, volume 121 of *Lecture Notes in Informatics*, pages 140–154, 2008.

[10] S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Trans. on Software Eng.*, 32(7):486–502, 2006.

[11] S. Kounev and A. Buchmann. Performance Modeling and Evaluation of Large-Scale J2EE Applications. In *Proc. 29th Int'l Computer Measurement Group (CMG) Conf.*, 2003.

[12] H. Koziolek, S. Becker, and J. Happe. Predicting the Performance of Component-based Software Architectures with different Usage Profiles. *Proc. 3rd Int'l Conf. on the Quality of Software Architectures (QoSA'07)*, *Springer LNCS* 4880, pages 145–163, 2007.

[13] H. Koziolek, S. Becker, J. Happe, and R. Reussner. *Model-Driven Software Development: Integrating Quality Assurance*, Chapter Evaluating Performance of Software Architecture Models with the Palladio Component Model, pages 95–118. IDEA Group, 2008.

[14] K. Krogmann, M. Kuperberg, and R. Reussner. Reverse Engineering of Parametric Behavioural Service Performance Models from Black-Box Components. *MDD, SOA und IT-Management (MSI 2008)*, pages 57–71, 2008. GITO Verlag.

[15] K. Krogmann and R. H. Reussner. *The Common Component Modeling Example*, *Springer LNCS* 5153, Chapter Palladio: Prediction of Performance Properties, pages 297–326, 2008.

[16] M. Kuperberg, K. Krogmann, and R. Reussner. Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models. *Proc. 11th Int'l Symposium on Component Based Software Engineering (CBSE 2008)*, *Springer LNCS* 5282, pages 48–63, 2008.

[17] D. Menasce, V. Almeida, and L. Dowdy. *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*. Prentice-Hall, New Jersey, Mar. 1994.

[18] A. Mos and J. Murphy. A Framework for Performance Monitoring, Modelling and Prediction of Component Oriented Distributed Systems. *WOSP '02: Proc of the 3rd Int'l Workshop on Software and Performance*, pages 235–236, 2002. ACM.

[19] Y. Natis, M. Pezzini, K. Iijima, and R. Favata. Magic Quadrant for Enterprise Application Servers, 2Q08, 2008. Gartner RAS Core Research Note G00156200.

[20] F. Rometsch and H. Sauer. Dynatrace Diagnostics: Performance-Management und Fehlerdiagnose vereint. *iX*, 9/2008:72–75, 2008.

[21] R. Sly. Introduction to the WebLogic Diagnostics Framework (WLDF). 2006. Dev2Arch Article. Last visit: 2009-07-23. `http://www.oracle.com/technology/pub/articles/dev2arch/2006/06/wldf.html`.

[22] T. Zheng, C. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Trans. Software Eng.*, 34(3):391–406, 2008.