Model-based Self-Adaptive Resource Allocation in Virtualized Environments*

Nikolaus Huber
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
nikolaus.huber@kit.edu

Fabian Brosig
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
fabian.brosig@kit.edu

Samuel Kounev
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
samuel.kounev@kit.edu

ABSTRACT

The adoption of virtualization and Cloud Computing technologies promises a number of benefits such as increased flexibility, better energy efficiency and lower operating costs for IT systems. However, highly variable workloads make it challenging to provide quality-of-service guarantees while at the same time ensuring efficient resource utilization. To avoid violations of service-level agreements (SLAs) or inefficient resource usage, resource allocations have to be adapted continuously during operation to reflect changes in application workloads. In this paper, we present a novel approach to self-adaptive resource allocation in virtualized environments based on online architecture-level performance models. We present a detailed case study of a representative enterprise application, the new SPECjEnterprise2010 benchmark, deployed in a virtualized cluster environment. The case study serves as a proof-of-concept demonstrating the effectiveness and practical applicability of our approach.

Categories and Subject Descriptors

C.4 [**Modeling techniques**]; I.6.4 [**Model Validation and Analysis**]

General Terms

Performance, Management

Keywords

Self-adaptive, resource management, virtualization

1. INTRODUCTION

Recent trends like virtualization and Cloud Computing aim at decoupling applications and services from the underlying hardware infrastructures. This provides increased flexibility because resources (e.g., CPU, memory, network bandwidth) can be allocated on demand and adapted in response to changes in system workloads. For example, in Cloud Computing resource allocations can be scaled up and down in an elastic manner reflecting the load intensity and resource demands of running applications. Moreover, virtualization allows to reduce the number of physical servers in data centers by running multiple independent virtual machines (VMs) on the same physical hardware. By improving energy efficiency, this promises significant cost savings for both service providers and infrastructure providers.

However, the above benefits come at the cost of increased system complexity and dynamics, making it challenging to provide Quality-of-Service (QoS) guarantees (e.g, availability and performance) in the face of highly variable application workloads. Service providers are faced with the following issues during operation: How much resources should be allocated to a new service deployed in the virtualized infrastructure on-the-fly, in order to guarantee Service-Level Agreements (SLA) for both the new service and existing services? How should resource allocations of running applications and services be adapted in response to changes in their workloads? How much additional resources are required to sustain increasing load conditions due to growing customer workloads? How much resources can be released, without compromising SLAs, in order to avoid inefficient resource usage after a drop in the workload intensity? Answering such questions requires the ability to predict at *run-time* how the performance of running applications would be affected if service workloads change, as well as the ability to predict the effect of changing resource allocations to adapt the system accordingly. We refer to this as *online performance prediction*. The latter allows to *proactively* adapt the system to the new workload conditions avoiding SLA violations or inefficient resource usage.

Over the past decade, a number of performance prediction techniques based on *architecture-level* performance models have been developed by the performance engineering community as surveyed in [18]. However, these techniques are targeted for offline use at system design and deployment time, and are normally employed for evaluating alternative system designs and/or for sizing and capacity planning before putting the system into production. The advantage of such techniques, compared to techniques, e.g., [21, 19, 10], based on classical performance models (e.g., queueing networks), is that they could potentially allow to explicitly capture the performance influences of the software architec-

*This work was funded by the German Research Foundation (DFG) under grant No. KO 34456-1.

SEAMS '11, May 23-24, 2011, Waikiki, Honolulu, HI, USA

ture, the application usage profiles as well as the execution environment. While the software architecture typically does not change during operation, the application usage profiles and the resource allocations at the various levels of the execution environment may change frequently. Moreover, even though the software architecture does not change often, its performance-relevant behavior has to be taken into account. For example, the input parameters passed to a service may have direct impact on the set of software components involved in executing the service, as well as their internal behavior and resource demands. Therefore, a detailed performance model capturing the performance-relevant aspects of both the software architecture and the multi-layered execution environment, as well as the dependencies on the usage profile, is needed.

In this paper, we present a novel approach to self-adaptive resource allocation in virtualized environments based on online architecture-level performance models. We explore the use of such models as a means for online performance prediction allowing to predict the effects of changes in user workloads, as well as to predict the effects of respective reconfiguration actions, undertaken to avoid SLA violations or inefficient resource usage. We present a detailed case study with a representative application, the new SPECjEnterprise2010 benchmark[1], deployed in a virtualized cluster environment. The case study serves as a proof-of-concept showing the feasibility of using architecture-level performance models at run-time and the benefits they provide. The contributions of this paper are: i) a generic self-adaptive control loop and a respective resource allocation algorithm for virtualized environments based on online performance models, ii) an implementation of our approach in the context of a novel case study of a representative enterprise application, the new SPECjEnterprise2010 benchmark, of a realistic size and complexity, iii) an experimental evaluation of the developed framework demonstrating its effectiveness and practical applicability.

The rest of this paper is organized as follows: Section 2 provides some background on performance models, the foundation of this work. Section 3 describes our self-adaptive resource allocation approach. In Section 4, we describe the architecture of the SPECjEnterprise2010 benchmark, the resulting performance model, and the results of the evaluation of our approach. Finally, we review related work in Section 5 and wrap up with some concluding remarks in Section 6.

2. MODELING APPROACH

We distinguish between *descriptive* architecture-level performance models and *predictive* performance models. The former describe performance-relevant aspects of software architectures and execution environments (e.g., UML models augmented with performance annotations). The latter capture the temporal system behavior and can be used for performance prediction by means of analytical or simulation techniques (e.g., queueing networks or stochastic Petri nets). Over the past decade, a number of architecture-level performance meta-models for describing performance-relevant aspects of software architectures and execution environments have been developed by the performance engineering community, the most prominent examples being the UML SPT and MARTE profiles [23]. Other proposed meta-models include CSM, PCM and KLAPER [18]. The common goal of these efforts is to make it possible to predict the system performance by transforming architecture-level performance models into predictive performance models in an automatic or semi-automatic manner. They are normally used for evaluating alternative system designs or for sizing and capacity planning before putting the system into production. We explore the use of architecture-level performance models as a means for online performance prediction during system operation [16]. Such models allow for modeling the architectural layers and their configurations as well as the behavior of provided services explicitly. That way, the relative performance of different execution platforms or software stacks can be captured. Particularly when modeling virtualized environments, where virtualization layers may change during operation, an explicit modeling of the performance influence is valuable.

In recent years, with the increasing adoption of component-based software engineering, the performance evaluation community has focused on adapting and extending conventional performance engineering techniques to support component-based systems which are typically used as foundation for building modern enterprise applications. One of the most advanced component-based performance modeling languages, in terms of parametrization and tool support, is the Palladio Component Model (PCM) [18]. In this paper, we use the PCM as architecture-level performance model since it allows to explicitly model different usage profiles and resource allocations. To derive predictions from the PCM models, we use the Simucom framework [1] that implements a queueing-network based simulation.

In order to capture the time behavior and resource consumption of a component, PCM takes four factors into account [1]. 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 including service input parameters, and the execution environment in which the component is running have to be taken into consideration.

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/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)*. The goal of a RDSEFF is to capture the control flow and resource consumption of the service depending on the input parameters passed to it. The dependencies, e.g., loop iteration numbers or branch conditions depending on parameters passed upon service invocation, can be expressed probabilistically or deterministically. 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

[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`.

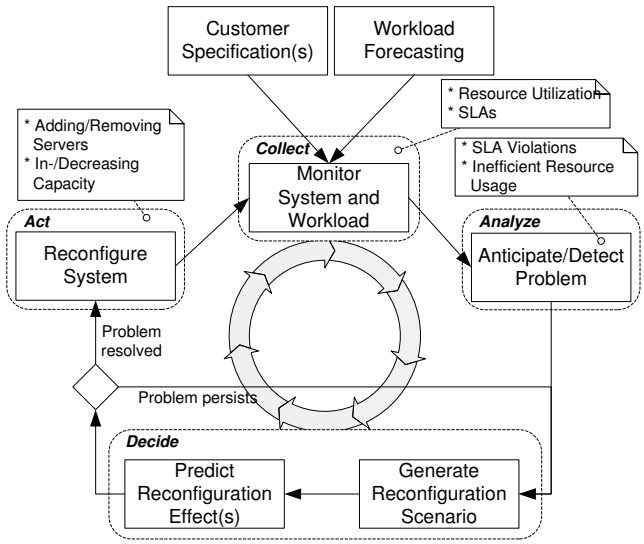

Figure 1: The system reconfiguration process [16].

allows modeling processing resources like, e.g., CPUs and disk drives. The *allocation model* describes the mapping of component instances 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 request frequency (workload intensity), the order in which services are called and the input parameters passed to them.

3. SELF-ADAPTIVE RESOURCE MANAGEMENT

This section presents our self-adaptive resource management algorithm which is based on a control loop model.

3.1 Adaptation Control Loop

The adaptation of the control loop described in [6] to our scenario is depicted in Figure 1. Currently, for the *collect* phase, we assume that changes of the workload are either announced by the customers (e.g., for an upcoming sales promotion) or by techniques like workload forecasting [3]. In the *analyze* phase, we use the described software performance models to predict the effect of changes and to *decide* which actions to take. The *act* phase implements the reconfiguration options considered in the analysis and described in the following.

Modern virtualization and middleware technologies provide multiple possibilities for dynamic resource (re)allocation and system reconfiguration. For example, virtualization allows to add/remove virtual CPU cores to virtual machines (VMs) or to change the hypervisor' scheduling parameters, e.g., increasing/decreasing the cap parameter. Application servers typically provide means to create application server clusters and dynamically add/remove cluster nodes. Furthermore, virtualization allows to migrate VMs from one physical server to another.

The mentioned dynamic reconfiguration options have their specific advantages and drawbacks. Some of them can be used at run-time but require a special system setup or introduce high reconfiguration overhead (e.g., VM live migration). Hence, for our self-adaptive resource allocation algorithm based on online performance prediction, we focus on adding/removing virtual CPUs and adding/removing application servers to an application server cluster.

3.2 Resource Allocation Algorithm

In this section, we present a formal definition of our resource allocation algorithm. This algorithm consists of two phases: PUSH phase and PULL phase. The PUSH phase allocates additional resources until all client SLAs are satisfied. The PULL phase optimizes the resource efficiency by deallocating resources that are not utilized efficiently. We present the algorithm in generic terms, such that it can be applied to different types of resources and resource allocation options.

Formally, the environment can be represented as a 3-tuple $M = (T, S, C)$ where:

$\mathbf{T} = \{t_1, t_2, ..., t_m\}$ is the set of resource types (e.g., types of VMs executed in the environment),

$\mathbf{S} = \{s_1, s_2, ..., s_n\}$ is the set of services offered in the environment,

$\mathbf{C} = \{c_1, c_2, ..., c_l\}$ is the set of client workloads and respective SLAs. Each $c_i \in C$ is a triple (s, λ, ρ) where $s \in S$ is the service used, λ is the workload intensity (expected request arrival rate), and ρ is the client requested average response time (SLA).

We define the following functions:

$V \in [S \rightarrow 2^T]$ specifies which resource types are required by service $s \in S$,

$F \in [S \times T \rightarrow I^{s,t}]$ referred to as *resource allocation function* assigns to each service $s \in S$ a set of instances $I^{s,t}$ of resource type $t \in T$ (e.g., VM instances). Each resource type instance is assumed to be allocated a given number of identical processing resources (e.g., CPUs, HDDs). Formally, the resource type instance $i \in I^{s,t}$ is represented as a triple $(\pi, \kappa, \overline{\kappa})$, where π is the processing rate of its processing resources, κ is the number of processing resources currently allocated (e.g., allocated virtual CPUs), and $\overline{\kappa}$ is the maximum number of processing resources that can be allocated (e.g., number of CPUs on a physical machine).

$D \in [S \rightarrow R^+]$ specifies the resource demand of service $s \in S$ in the same unit as the processing rate π of the resource type.

We define the following performance metrics:

$X(c)$ is the total number of requests of client workload $c \in C$ completed per unit of time (request throughput),

$R(c)$ is the average response time of a service request in client workload $c \in C$,

$U(t)$ is the average utilization of resource type $t \in T$ over all instances of the resource,

$\overline{U}(t)$ is the maximum allowed average utilization for resource type $t \in T$.

Finally, we define the following predicates:

$P_X(c)$ for $c \in C$ is defined as $(X(c) = c[\lambda])$,

$P_R(c)$ for $c \in C$ is defined as $(R(c) \leq c[\rho])$,

$P_U(t)$ for $t \in T$ is defined as $(U(t) \leq \overline{U}(t))$.

For a configuration represented by a resource allocation function F to be acceptable the following condition must hold $(\forall c \in C : P_X(c) \wedge P_R(c)) \wedge (\forall t \in T : P_U(t))$. This condition is checked by means of our online performance prediction mechanism.

Each time there is a change in the set of client workloads $C \longrightarrow \widetilde{C}$ (e.g., a new client workload $\widetilde{c} = (s, \lambda, \rho)$ is scheduled for execution or a change in the workload intensity λ of an existing workload is forecast), we use our online performance prediction mechanism to predict the effect of this change in the overall system workload. If an SLA violation is detected, the PUSH phase of our algorithm is executed which allocates additional resources until all client SLAs are satisfied. After the PUSH phase finishes, the PULL phase is executed to optimize the resource efficiency. If no SLAs are violated, the PULL phase starts directly. In the following, we describe the PUSH and PULL phases in more detail.

3.2.1 PUSH Phase

The following algorithm written in mathematical style pseudo code presents our basic heuristic for allocating resources to services such that client SLAs are satisfied.

```
while ∃c ∈ C̃ : ¬P_R(c) do
    for all t ∈ V(c[s]) : ¬P_U(t) do
        while cap(c,t) ≤ cap̄(c,t) do
            if ∃i ∈ F(c[s],t) : i[κ] < i[κ̄] then
                i[κ] ← i[κ] + 1
            else
                F(c[s],t) ← F(c[s],t) ∪ {î}
            end if
        end while
    end for
end while
```

Basically, while there exists a client response time SLA that is violated, the algorithm increases the amount of allocated resources for all resource types used by the service that currently exceed their maximum allowed utilization $\overline{U}(t)$. This is based on the assumption that violations are caused by at least one resource type used by the violated SLA has become a bottleneck. Increasing the number of allocated resources works as follows: If a there is an instance of the overutilized resource type t (e.g., a VM) which has some processing resources available (e.g., virtual CPUs) that are not allocated yet, additional resources are allocated. Otherwise, a new instance of the resource type $\widehat{i}$ is added (e.g., a new VM is started). In our algorithm, an additional resource instance increases the total capacity by one. Our algorithm assumes that there is an infinite amount of resources available and hence, this capacity increase is repeated until the amount of allocated resources reaches $\overline{cap}(c,t)$ defined as

$$\overline{cap}(c,t) = \left\lceil \frac{\sum_{c \in \widetilde{C}} c[\lambda] \cdot D(c[s])}{\sum_{c \in C} c[\lambda] \cdot D(c[s])} \right\rceil \cdot cap(c,t) \qquad (1)$$

The above is an estimated upper bound on the capacity of resource type t required to handle client workload c, based on a factor calculated by the specification changes and the previously assigned capacity $cap(c,t)$. It is calculated as the ratio of the newly specified arrival rates and the original arrival rates both multiplied with their respective resource demands. It is intended to reduce the number of scenarios for which online performance prediction has to be performed when searching for an acceptable configuration. Other methods on calculating this upper bound could be applied here in the future as well.

3.2.2 PULL Phase

The PULL phase aims to optimize the resource efficiency by trying to release resources that are not utilized much by the current client workloads.

```
for all c ∈ C do
    while ∃t ∈ V(c[s]) : Ū(t) − U(t) ≥ ε do
        if ∃i ∈ F(c[s],t) : i[κ] > 0 then
            i[κ] ← i[κ] − 1
            if ¬P_R(c) then
                i[κ] ← i[κ] + 1
            end if
            if i[κ] = 0 then
                F(c[s],t) ← F(c[s],t) \ {i}
            end if
        end if
    end while
end for
```

The optimization algorithm is applied to all client workloads $c \in C$. While there is a resource type t assigned to service s of the currently considered workload c whose delta between the maximum utilization $\overline{U}(t)$ and current utilization $U(t)$ is greater than a predefined constant ϵ, the amount of resources allocated to this service will be decreased, i.e., for a resource type instance i of t which currently has some resources allocated (e.g., virtual CPUs), the amount of allocated resources is decreased. If the client SLAs are predicted to be violated after this change, the change is reversed. In case after the change, the instance has no remaining allocated resources, the instance i can be removed from the set of resource type instances (e.g., VM can be shut down). Note that the set of a resource type instances can also become empty, e.g., if there is no service left using the respective resource type t.

4. CASE STUDY

In the following case study we evaluate our approach with the new SPECjEnterprise2010 benchmark. First, we explain the architecture of the benchmark and the corresponding performance model. Next, we describe the experimental setup the benchmark is deployed in. Finally, we present the results of our approach in different execution scenarios.

4.1 SPECjEnterprise2010

We selected the SPECjEnterprise2010 benchmark application as a basis for our case study since it models a representative, state-of-the-art system. In our case study, each service of the benchmark is considered as an independent service a client can invoke and specify SLAs for. Previous versions of the benchmark have already been successfully applied for research purposes [15, 17].

SPECjEnterprise2010 is a benchmark which is developed by SPEC's Java subcommittee to measure the end-to-end 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).

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

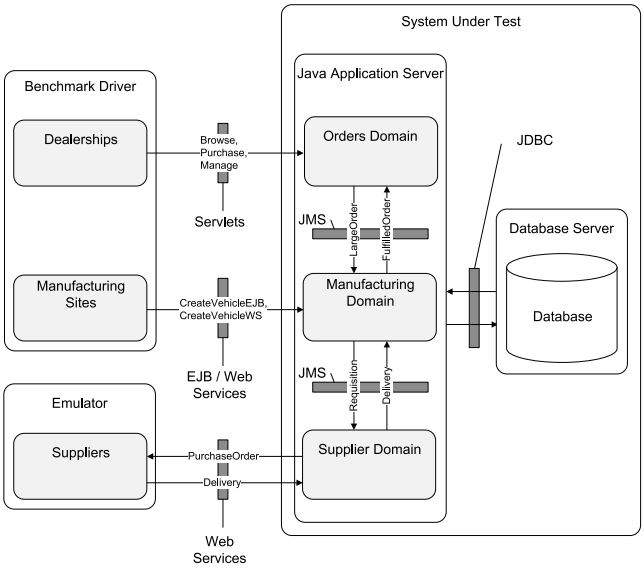

Figure 2: SPECjEnterprise2010 benchmark architecture.

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 2 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.

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. A manufacturer may place *work orders* for manufacturing vehicles, either triggered per WebService or RMI call.

The original benchmark driver distinguishes between the dealer and manufacturing domain, each having three and two benchmark operations, respectively. In the following, we refer to these benchmark operations as services. To control the request arrival rate of each service individually, we had to slightly modify the benchmark driver. We split up the two driver domains into five different domains, each invoking its own service. The resulting five independent services are called *Purchase*, *Manage*, *Browse*, *CreateVehicleEJB* and *CreateVehicleWS*. This separation enables us to control the workload intensity of each service independently of the others and to specify individual SLAs.

4.2 Performance Model

The predictions of the SPECjEnterprise2010 application are conducted using a PCM model as performance model. The PCM model is semi-automatically extracted from a running benchmark application instance. As extraction method, the method presented in [5] is used. While in [5], the focus was on modeling the manufacturing domain of a preversion of the benchmark, for this work, we extracted the entire benchmark application, i.e., including supplier domain, dealer domain, web tier and the asynchronous communication between the three domains.

The three main steps of the extraction process are: i) extraction of the application architecture, ii) extraction of the performance-relevant control flow and iii) extraction of resource demands. The extraction is based on monitoring data collected during operation with the Oracle WebLogic Diagnostics Framework (WLDF) running on Oracle WebLogic server instances.

In the first step, the effective application architecture is extracted. The latter refers to the set of components and connections between components that are effectively used during operation. The component boundaries, i.e., the question which servlets or Enterprise Java Beans (EJBs) form a component, can be provided manually. If not, the method considers each EJB, servlet and Java Server Page (JSP) as individual 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 [4]. 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.

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 [12]. We monitor the effective control flow, extract probabilities of different call paths in contrast to extracting

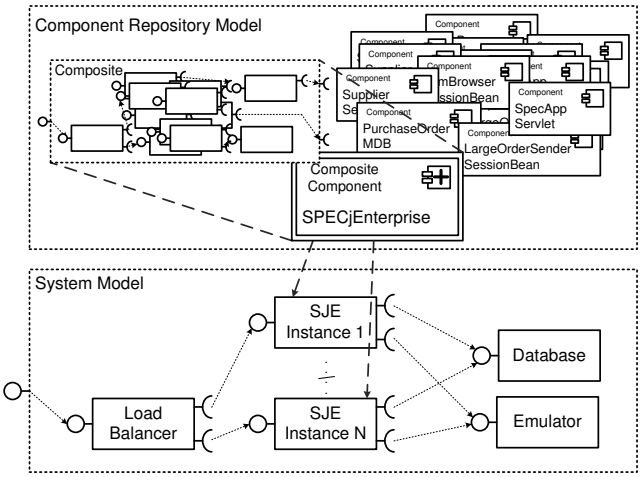

Figure 3: Modeling SPECjEnterprise Benchmark Application Structure.

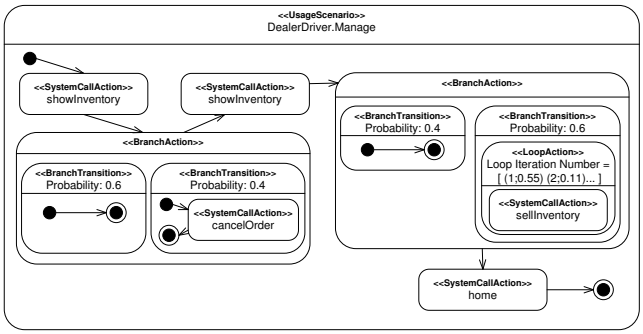

Figure 4: Usage scenario for operation *Manage*.

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. That way, 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 probability mass function (PMF). For `BranchActions`, branch transition probabilities are extracted.

After having the model structure extracted in the previous two steps, the resource demand annotations are estimated based on measured utilization and throughput data using the service demand law [21]. We partition the total resource utilization using weighted response time ratios as described in [5].

Figure 3 gives an overview of how the structure of the resulting PCM performance model looks like. 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

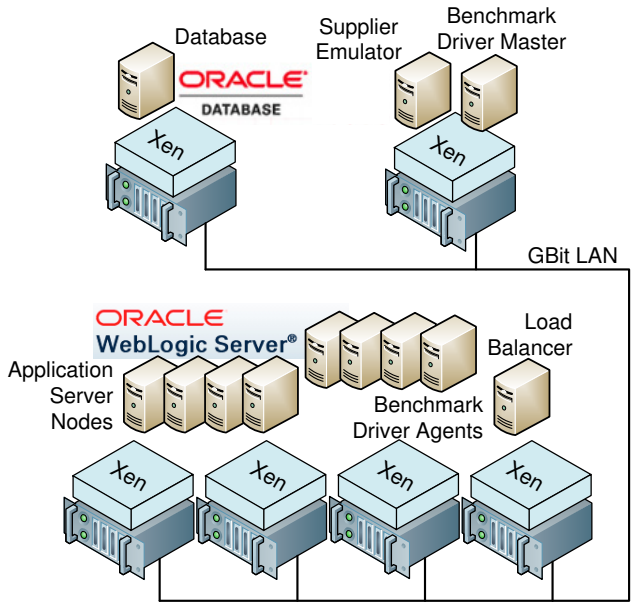

Figure 5: Experiment setup.

in the component repository. The composite component in turn consists of component instances, e.g., a `SpecAppServlet` component or a `PurchaseOrderMDB` component. Those 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 usage model representing the benchmark workload has been provided manually. The five benchmark operations are modeled as individual usage scenarios. Figure 4 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 probability mass function. For instance, in the depicted example, with a probability of 55% the loop body is executed only once, with a probability of 11% the loop iterates two times. The probabilities of the loop iteration numbers are derived from monitoring data. The remaining four benchmark operations are of similar complexity.

4.3 Experimental Setup

As hardware environment for the experiments, we use six blade servers from a cluster environment. Each server is equipped with two Intel Xeon E5430 4-core CPUs running at 2.66 GHz and 32 GB of main memory. The machines are connected by a 1 GBit LAN. Figure 5 shows the experiment environment. On top of each machine, we run Citrix XenServer 5.5 as the virtualization layer. Inside the XenServer's VMs, we run the benchmark components (application servers, driver agents, emulator, load balancer). Each component runs in its own VM, initially equipped with 2 virtual CPUs (VCPUs). As operating system, these VMs execute CentOS 5.3. As Java EE application server, we use the Oracle Weblogic Server (WLS) 10.3.3. The load balancer is

haproxy 1.4.8 using round-robin as load balancing strategy. The driver agents are provided by the Faban framework that is shipped with the benchmark. The database is an Oracle 11g database server instance deployed on a VM with eight VCPUs on a separate node on Windows Server 2008.

The SPECjEnterprise2010 benchmark application is deployed in a cluster of WLS nodes. For the evaluation, we considered reconfiguration options concerning the WLS cluster and the VCPUs the VMs are equipped with: WLS nodes are added to or removed from the WLS cluster, VCPUs are added to or removed from a VM. These reconfigurations are applicable at run-time, i.e., can be applied while the benchmark application is running. In our experiments, the VMs (WLS nodes) map to resource type instances in our reallocation algorithm and their VCPUs map to the capacity parameter. For the latter, we change the amount of VCPUs of a VM between four and two. These limits were chosen for technical reasons to ensure that all allocated resources are dedicated and not shared by several VMs.

4.4 Evaluation

We evaluate our approach in different scenarios presented in the following.

4.4.1 Adding a New Service

This first scenario is intended to evaluate the results of our approach when a new service is deployed in the environment on-the-fly. Assume that there are four services executed in our environment running on one node with two VCPUs (default configuration c_0). The SLAs of the four currently running services are as follows: (CreateVehicleEJB, 15, 54ms), (Purchase, 12.5, 80ms), (Manage, 12.5, 80ms), (Browse, 25, 80ms). This specification follows the same notation as the more general client workload specifications in Section 3.2.

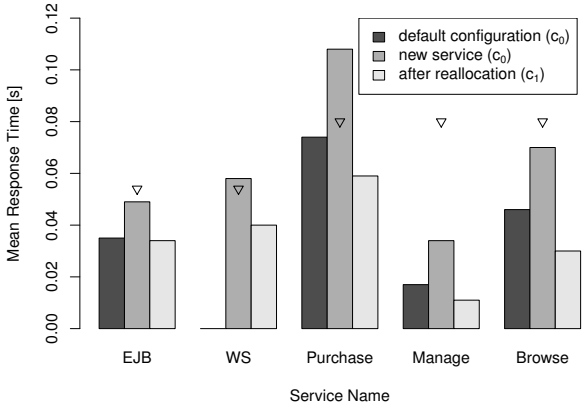

Figure 6: The response times of the five services and their respective SLAs (denoted by ▽) before and after reallocation when adding a new service.

Now a new service with the SLA (CreateVehicleWS, 15, 54ms) is added. To ensure that all SLAs are still maintained after deployment of the new service, our self-adaptive resource allocation mechanism is triggered. The results are depicted in Figure 6. After adding the new service to the model, the simulation predicts SLA violations for the services CreateVehicleWS and Purchase. Hence, the PUSH-Phase of the reconfiguration algorithm starts and suggests a capacity increase by one, hence adds an additional VCPU to the existing node (configuration c_1). After this change, the simulation indicates satisfied SLAs, hence the algorithm enters the PULL-Phase and tries to reduce the overall amount of used resources considering all workload classes, but fails because then the SLAs of CreateVehicleWS and Purchase are again violated. Therefore, the resulting configuration of our algorithms consists of one node with three VCPUs.

The above behavior was confirmed in our experiments depicted in Figure 6. The measurements show that with the default resource allocation the SLA for service CreateVehicleWS and Purchase cannot be sustained. However, after applying the resource allocation proposed by our algorithm, all SLAs are satisfied.

4.4.2 Workload Growth

In this scenario, we evaluate our approach when increasing the workload of all services deployed in our environment. We increase the load in two steps from 2x to 4x and 4x to 6x (see Figure 7). The standard workload (1x) is the workload as defined in the previous scenario for all five services.

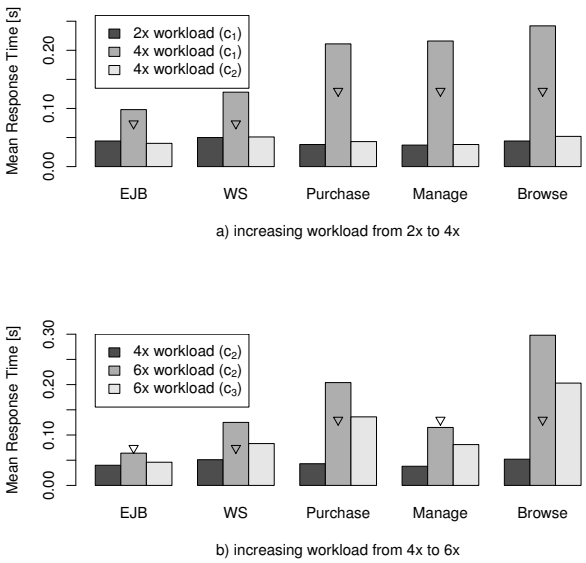

Figure 7: The response times when changing workload from 2x to 4x and 4x to 6x, respectively (SLAs denoted by ▽).

Our starting point is that five services are running on one node with three VCPUs (c_1) with twice the standard workload and the following SLAs (CreateVehicleEJB, 30, 74ms), (CreateVehicleWS, 30, 74ms), (Purchase, 25, 130ms), (Manage, 25, 130ms), (Browse, 50, 130ms) which are initially satisfied. Now, we increase the workload to 4x the standard load. For this new workload, the reallocation algorithm detects a violation of the SLAs and recommends to reallocate the system resources using two nodes, one with four VCPUs and one using three VCPUs (c_2). Applying this configuration to our benchmark, the SLAs are satisfied. For the

measurement results see Figure 7 a).

In the second step, we increase the workload to 6x the standard load, not changing the SLAs. Again, this leads to a violation of the SLAs in our simulation results. Therefore, we apply our algorithm, finding a new suitable configuration with three nodes, two with four VCPUs and one with three VCPUs (c_3). The experiment results are depicted Figure 7 b). However, the results show that after reallocation the SLA of the Browse service is still slightly violated. This is not due to inaccuracy of our model, but rather due to scalability problems of the database machine, which is not powerful enough to handle the new workload while satisfying the original SLAs. Hence, we are confident that given a more powerful database, the SLAs would be satisfied. The way this problem would be addressed in practice would be to either scale the database or renegotiate the SLAs. As both solutions can be handled with our online performance prediction mechanism, we plan to extend our approach with this solution in the future.

4.4.3 Workload Decrease

This scenario's purpose is to evaluate our approach in situations were the workload decreases. The intention is to release resources that are not utilized efficiently and hence increase the system efficiency.

Assume the situation that all services are executed with 6x the standard workload on three nodes - a total number of eleven VCPUs and all SLAs are satisfied (c_3). Now, we decrease the workload to 4x the standard load. For this change, our approach predicts that two nodes with a total of seven VCPUs (c_4) are sufficient to handle the decreased workload. The measurement results are depicted in Figure 8, demonstrating that the recommendation is correct. One can see that for configuration c_4 the average response time increases, but the SLAs are still satisfied.

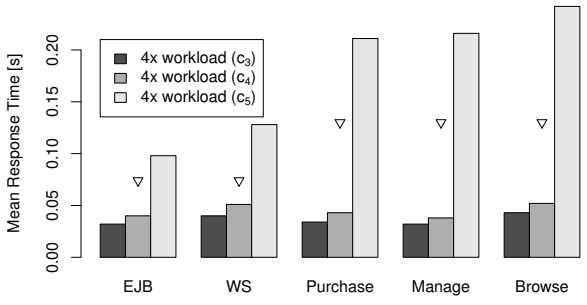

Figure 8: The response times when executing 4x workload before and after reconfiguration and with less resources (SLAs denoted by ▽).

To verify that resource allocations cannot be further reduced while satisfying the SLAs, we further reduced the allocated resources manually to one node with four VCPUs (c_5). The results for this configuration show that it would violate the SLAs, hence the previously found configuration is valid.

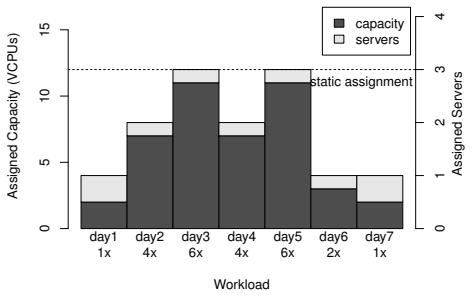

Figure 9: Assigned capacity and servers for a workload distribution over seven days.

4.4.4 Resource Usage and Efficiency

After evaluating the functionality of our approach, this section discusses its benefits. Imagine a workload distribution over seven days like the one depicted in Figure 9. In a static scenario, one would assign three dedicated servers to guarantee the SLAs for the peak load. However, with our approach one can dynamically assign the system resources. In the static scenario, one would use $7 \cdot 3 = 21$ servers, whereas our approach needs only $1+2+3+2+3+1+1 = 13$ servers. Hence, in such a scenario, only 62% of the resources of the static assignment are needed and thereby almost 40% of the resources available can be saved.

4.4.5 Limitations and Future Work

In this paper, we use features of virtualization for reconfiguration at runtime. To avoid the effects of shared resources, in our experiments virtual resources are dedicated and each VM is executed on a single physical machine to avoid resource contention. Nevertheless, we are investigating the influences of virtualization on system performance to integrate the gained insights into our performance models [9]. Moreover, for the validation, we used an experimental environment with homogeneous hardware. We plan to evaluate our approach on heterogeneous hardware environments in the future. Also, currently we only consider CPU as the resource type but our algorithm is designed to support arbitrary resources, e.g., hard disks or network, which we also plan to evaluate in further experiments.

5. RELATED WORK

Many related work in terms of resource allocation algorithms which is done offline can be found in the fields of capacity planning and resource management. In recent years with virtualization and Cloud Computing also other approaches on automatic resource management appear. In this section, we give a brief summary of the latter approaches and then present some works in more detail.

There has been many research done on the resource allocation problem using techniques like bin packing, multiple knapsack problems, etc. For dynamic resource allocation, previous research addresses this problem using linear optimization techniques [13] or non-linear optimization strategies based on simulated annealing [27], fuzzy logic [28], or others. However, the resource allocation problem in virtu-

alized environments is more complex because of the introduction of virtual resources. Along with the growth of cloud computing, there have been several approaches on QoS and resource management at runtime [19, 2, 22, 8, 11]. However, these approaches are often on a very high level of resource management and deal with very coarse-grained resource allocation (e.g. are restricted to only adding/removing VMs) [22] or focus on different optimization targets, e.g., the profit [19]. Recently, approaches on adaptation based on virtualized systems have been proposed [10, 24, 25]. Problem of all the approaches is that their performance predictions are rather restricted (if any) in terms of the level of detail. The approaches do either not consider run-time dynamics and aspects relevant to predicting the performance behavior in an evolving environment or the presented capacity management techniques abstract the platform as a black box.

In [14], algorithms placing virtual machines on physical servers while attempting to minimize the cost of migration and maintaining acceptable application performance levels are presented. This approach focuses on VMs as a whole and does neither consider the type of service executed in the VM nor use performance models to predict the impact of changes in virtualized environments.

Mistral [10] is a resource managing framework with a multi-level resource allocation algorithm considering similar reallocation actions: adapt a VM's CPU capacity, add or remove a VM, live-migrate a VM between hosts, and shutdown or restart a host. This approach considers power consumption, performance and transient costs in its reconfiguration algorithm. However, the approach is based on a simple multi-tier application with read-only transactions and a fixed web tier modeled with a layered queuing network (LQN), further details are missing. The extensive evaluation shows promising results, but is restricted to 4 different instances of the RUBiS package, and is not evaluating different types of services.

Steinder et al. propose an implementation [25] and a placement algorithm [13] for heterogeneous workloads in virtualized environments. Their results demonstrate how virtualization technology can be used to manage the performance goals of different workloads (emulated by different benchmarks). However, this approach does not use performance prediction techniques and hence is a reactive approach.

In [24], a feedback control system consisting of an online model estimator and a resource controller is presented. The target of this framework is to dynamically allocate resources to applications running in a virtualized environment. For performance prediction, this approach is based on transfer functions to model the dynamic relationship between a performance metrics and physical control features. Although the authors show that their model predicts the application performance accurately, the model is rather coarse-grained. As reconfiguration options, the authors focus on the hypervisor's cap and disk share parameters. Nevertheless, the evaluation with RUBiS and TPC-W in combination with a production-trace-driven workload is promising.

An example for QoS-aware dynamic resource management on the middleware level is [20]. This approaches uses load-balancing and middleware techniques (clustering of application servers) to ensure QoS in distributed enterprise applications. Their results show that the presented approach and its configuration based on monitoring data is suitable to meet SLAs while optimizing resource utilization. However, this approach is a purely reactive approach and does not use any prediction techniques or models.

Also interesting and closely related is the work [26], which proposes the SLAstic framework. The authors also consider runtime system reconfiguration by means of performance models. However, no experimental validation has been presented so far.

6. CONCLUSIONS

In this paper, we presented a novel approach to self-adaptive resource allocation at runtime. We used performance models to predict the effect of changes in the service workloads and the respective system reconfiguration actions. By using virtualization techniques, we applied these allocation changes to the SPECjEnterprise2010 benchmark to evaluate the use of such models for online performance prediction.

The results show that our approach can be applied to react on changes during runtime to find efficient resource allocations while satisfying specified SLAs. In an example, we showed that this approach can save up to 40% of the resources. More important to us is that this case study demonstrates that architecture-level performance models can be used effectively at runtime to support self-adaptiveness.

The findings of this case study underline the importance of further research on engineering of self-aware systems in virtualized environments. In the future, we plan to extend our resource allocation with improved heuristics for finding resource allocations. Moreover, we plan to evaluate our approach with different resource types. In addition, we intend to consider the effect of shared resources in virtualized environments by extending the performance models with virtualization effects [9] and of other physical resources like network or storage. This future research will be carried out as part of the Descartes Research Project [16, 7].

7. REFERENCES

[1] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 2009.

[2] M. N. Bennani and D. Menascé. Resource Allocation for Autonomic Data Centers using Analytic Performance Models. In *ICAC*, 2005.

[3] G. Box and G. Jenkins. *Time series analysis: forecasting and control*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1994.

[4] L. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE TSE*, (9), 2006.

[5] F. Brosig, S. Kounev, and K. Krogmann. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *Proceedings of ROSSA 2009*. ACM, Oct. 2009.

[6] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, 2009.

[7] Descartes Research Group. http://www.descartes-research.net, December 2010.

[8] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini. QoS-aware Clouds. In *IEEE Computer Society Press Cloud 2010*, 2010.

[9] N. Huber, M. von Quast, F. Brosig, and S. Kounev. Analysis of the Performance-Influencing Factors of Virtualization Platforms. In *DOA'10*, 2010.

[10] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu. Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures. In *ICDCS*, 2010.

[11] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu. Generating adaptation policies for multi-tier applications in consolidated server environments. In *ICAC*, 2008.

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

[13] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *International Conference on World Wide Web*, 2006.

[14] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 373–381. IEEE, 2006.

[15] S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE TSE*, (7), July 2006.

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

[17] S. Kounev and A. Buchmann. Performance Modeling and Evaluation of Large-Scale J2EE Applications. In *Proceedings of Intl. CMG Conference 2003*, 2003.

[18] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, August 2009.

[19] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai. Performance model driven QoS guarantees and optimization in clouds. In *CLOUD '09*, Washington, DC, USA, 2009. IEEE.

[20] G. Lodi, F. Panzieri, D. Rossi, and E. Turrini. SLA-driven clustering of QoS-aware application servers. *IEEE TSE*, 2007.

[21] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Capacity Planning and Performance Modeling˜- From Mainframes to Client-Server Systems*. Prentice Hall, Englewood Cliffs, NG, 1994.

[22] J.-M. Menaud, H. Nguyen Van, and F. Dang Tran. Performance and Power Management for Cloud Infrastructures. In *IEEE Cloud 2010*, 2010.

[23] Object Management Group (OMG). UML SPT, v1.1 (January 2005) and UML MARTE (May 2006).

[24] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of EuroSys '09*. ACM, 2009.

[25] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server virtualization in autonomic management of heterogeneous workloads. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 139–148, 2007.

[26] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In *Proc. of Warm Up Workshop for ICSE 2010*.

[27] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, and Q. Wang. Appliance-based autonomic provisioning framework for virtualized outsourcing data centre. In *ICAC 2007*.

[28] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. In *ICAC 2007*.