Pre-print version for personal use only!

Chapter 17
Online Learning of Run-time Models for Performance and Resource Management in Data Centers

Jürgen Walter, Antinisca Di Marco, Simon Spinner, Paola Inverardi,
Samuel Kounev

Abstract In this chapter, we explain how to extract and learn run-time models that a system can use for self-aware performance and resource management in data centers. We abstract from concrete formalisms and identify extraction aspects relevant for performance models. We categorize the learning aspects into: i) model structure, ii) model parametrization (estimation and calibration of model parameters), and iii) model adaptation options (change point detection and run-time reconfiguration). The chapter identifies alternative approaches for the respective model aspects. The type and granularity of each aspect depends on the characteristic of the concrete performance models.

Jürgen Walter
University of Würzburg, Department of Computer Science, Am Hubland, D-97074 Würzburg, Germany, e-mail: juergen.walter@uni-wuerzburg.de

Antinisca Di Marco
University of L'Aquila Via Vetoio 1, 67010 Coppito (AQ), Italy,
e-mail: antinisca.dimarco@univaq.it

Simon Spinner
University of Würzburg, Department of Computer Science, Am Hubland, D-97074 Würzburg, Germany e-mail: simon.spinner@uni-wuerzburg.de

Paola Inverardi
University of L'Aquila Via Vetoio 1, 67010 Coppito (AQ), Italy,
e-mail: paola.inverardi@univaq.it

Samuel Kounev
University of Würzburg, Department of Computer Science, Am Hubland, D-97074 Würzburg, Germany e-mail: samuel.kounev@uni-wuerzburg.de

Pre-print version for personal use only!

17.1 Introduction

In order to become self-aware, systems require an internal representation of themselves. According to the definition of self-aware computing systems given in Chapter 1, such systems

1. *learn models* capturing *knowledge* about themselves and their environment (such as their structure, design, state, possible actions, and run-time behavior) on an ongoing basis and
2. *reason* using the models (for example predict, analyze, consider, plan) enabling them to *act* based on their knowledge and reasoning (for example explore, explain, report, suggest, self-adapt, or impact their environment)

in accordance with *higher-level goals*, which may also be subject to change. In this chapter, we focus on learning performance models of IT systems and infrastructures that can be used for online performance and resource management in data centers. As stressed in the definition, the term learn does imply that some information based on which models are derived is obtained at system run-time, while also additional static information built into the system at design-time can be employed as well. Typically, a system may be built with integrated skeleton models whose parameters are estimated using monitoring data collected at run-time. Once an initial model is available, it is subjected to continuous updates at run-time to reflect any changes in the system itself and/or in the environment during operation.

A performance model is any abstraction of the system and its environment capturing knowledge that can be used to reason about the performance of the system. One main challenge in learning performance models is that the built models should present a right level of abstraction since it has to be detailed enough to guarantee accurate performance results while maintaining the adequate abstraction to be resolvable at run-time. The learned models, also called extracted models, can be descriptive, prescriptive and predictive models (see Chapter 1). Note that this classification is not mutually exclusive. One major challenge of the performance model learning is to find the right level of abstraction such that the constructed models are detailed enough to support accurate performance analysis, while at the same time they can be solved with reasonable overhead.

Approaches to automatically construct analytical performance models, such as Queueing Network(s)s (QNs), are very popular (e.g., [37, 39] and [40]). Often in such approaches, performance models are learned in a testing phase and applied at run-time. However, the constructed models are rather limited since they abstract the system at a very high level without capturing its architecture and configuration explicitly. For example, such models offer no means to express a change in the configuration of the virtualization platform, which may have a significant impact on the performance of system. Moreover, they often impose restrictive assumptions such as a single workload class or homogeneous servers. Furthermore, the model structure is typically assumed to be static. The structure is either derived by hand or tied to a specific application scenario (e.g, n-tier-web applications [1] or MapReduce jobs [48]) and only model parameters are derived using run-time monitoring data [29]. Fur-

Pre-print version for personal use only!

thermore, extraction case studies are often limited to certain technologies, e.g., Java EE systems [10] or certain specific target platforms like Oracle WebLogic server offering a proprietary monitoring infrastructure [8]. There is a lot of existing work on the extraction and maintenance of performance models. However, the extraction process often includes non-automated manual subparts or a priori knowledge about the system. Few solutions support a full automation and provide portability beyond one or two case studies.

In this chapter, we focus on gray-boxor white-box models as they contain structural and causal information (see Chapter 16), whereas black-box models do not have any a priori structure imposed. In contrast to white-box models, black-box models serve as interpolation of the measurements and lack information that is required for system adaptation (i.e., resource management at run-time). Hence, black-box model extraction, e.g., using genetic optimization techniques [15, 45], is not considered in the following. There are numerous performance modeling formalisms in the literature (cf. Chapter 16) and due to their high number, we cannot discuss the extraction process separately for each of them. However, most performance models provide a common set of features that have to be extracted, which allows us to abstract from the concrete modeling formalisms. We propose to structure the model extraction and maintenance into the following three disciplines, namely: i) model structure, ii) model parametrization (estimation and calibration of model parameters), and iii) model adaptation options (change point detection and run-time reconfiguration). We focus on the extraction of architectural performance models, as they provide the highest flexibility and potential for realizing the general idea of self-aware computing in data centers. Such models combine the descriptive and semantic aspects of architectural models, such as the Unified Modeling Language (UML), with the prediction capabilities of analytical models, like QN. Nonetheless, most of the techniques we present are not restricted to architectural performance models. In this chapter, we use the following acronyms:

APM Application Performance Management
DML Descartes Modeling Language
EJB Enterprise Java Beans
Java EE Java Enterprise Edition
LQN Layered Queueing Network(s)
QN Queueing Network(s)
QPN Queueing Petri Net(s)
PMF Performance Management Framework
S/T/A Strategies/Tactics/Actions
UML Unified Modeling Language

The remainder of this chapter is structured as follows: Section 17.2 discusses cross-cutting concerns for model extraction. Section 17.3 explains possibilities for system structure extraction. Section 17.4 discusses parametrization of models with resource demands and branching probabilities. Section 17.5 is about system and model adaptation. Section 17.6 concludes the chapter and discusses future work.

Pre-print version for personal use only!

17.2 Cross-Cutting Concerns

The learning of performance models requires several issues to be taken into account in order to guarantee the correctness of the process and its efficiency.

Data Collection Performance models are parameterized by means of monitoring data. Disseminated commercial Application Performance Management (APM) tools are e.g., Dynatrace, AppDynamics, NewRelic, and Riverbed Technology [9]. Additionally, free and open source performance monitoring tools exist, e.g., Kieker [9]. Typically, trace data is collected from the running system and is used to estimate model parameters, e.g, request arrival rates, service resource demands, or control flow parameters like routing probabilities. Problems that might follow include:

- The collected data may be more fine-grained than the performance model parameters, thus an aggregation step may be needed.
- The measurement overhead may significantly influence the system.

Modeling Abstraction and Formalism The constructed performance models have to be modified and evaluated online. This poses requirements on the models themselves. The choice of a suitable performance model of the system becomes one of the most important and critical steps. Indeed, the models should be as flexible as possible supporting dynamic adaptation when the system is reconfigured or its architecture evolves and at the same time they should support efficient performance analysis at run-time. In many cases, these two characteristics may be incompatible. The model flexibility requires detailed models that support changes, such as *re-parameterization* based on online monitoring data or *modification* in terms of their topology, in order to reflect a new system configuration. The efficient performance analysis requires models that can be solved in a *short* amount of time. This implies that in many cases only models having analytical/numerical solution may be usable at run-time. The challenge here is to design performance models expressive enough to describe different resource allocations and system configurations with respect to their performance behavior, but still having numerical/analytical solution.

Online Evaluation In order to control the state space explosion , system performance models should be as expressive as possible, omitting irrelevant details about individual system components behavior. This is important for lightweight and fast model evaluation. Of course, there is a trade-off between the simplicity of models and their support for detailed feedback facilitating online decision making.

Pre-print version for personal use only!

17.3 Model Structure

The learning of model structure involves extracting information about the software components of the system as well as information about resource landscape in which the system is deployed. The learning of inter-component interactions is covered in the next section since it includes dynamic aspects. Although different types of software components exist, for pragmatic reasons, we consider a component simply as an element of the software architecture providing one or more services. Software systems that are assembled from existing prepackaged components may be represented by the same components in a performance model [46]. Examples of components are: web services, Enterprise Java Beanss (EJBs) in Java Enterprise Edition (Java EE) applications [7, 8, 10], or `IComponent` extensions in .NET. Apart from building performance models of systems assembled from prepackaged components, previous research on architecture extraction targeted system reengineering scenarios. Examples of reverse engineering tools and approaches in this area are for example FOCUS [17], ROMANTIC [25], Archimetrix [44] or SoMoX [3]. These approaches are either clustering-based, pattern-based, or a combination of both. Components are identified from a reengineering perspective which does not necessarily correspond to deployable structures of the current implementation. Furthermore, the choice of an appropriate granularity is important. The complexity of a component decomposition can be reduced by merging sub-services. However, there is no rule that can be automatically applied to solve the granularity problem. To allow automated learning of the system structure, the system and its components should provide information about their boundaries. If no predefined component boundaries are provided, component identification requires manual effort. In general, the following guidelines can be applied in case of component-based systems implemented using an object-oriented programming language: i) classes that implement component interfaces form components, ii) all classes that inherit from a base class belong to the same component, iii) component A uses component B $\rightarrow$ A is a composite component containing B. Compared to component extraction, the automated identification of hardware and software resources in a system environment is already supported by industrial software tools For instance, [26] or [47] provide such functionalities. An open issue is to define common interfaces for resource extraction. This would greatly improve the integration of different tool chains supporting interoperability. Deployment information can be extracted using event logs containing identifiers for software components and resources. The extraction process creates one deployment component for each pair of software component and resource identifier. Summing up, there are still many open research questions and a lack of tool support for a full automation of the extraction of information about the system structure. However, many semi-automatic solutions have already been proposed and successfully applied in case studies.

Pre-print version for personal use only!

17.4 Model Parameterization

Model parameterization includes the determination of resource demands, determination of the inter-component interactions (control flow), and the extraction of load profile. In performance models, component resource demands are key parameters required for quantitative analysis. A resource demand describes the amount of a hardware resource needed to process one unit of work (e.g., a user request, a system operation, or an internal action). The granularity of resource demands depends on the abstraction level of the control flow in a performance model. Resource demands may depend on the values of input parameters. This dependency can be either captured by specifying the stochastic distributions of resource demands or by explicitly modeling parametric dependencies.

The estimation of resource demands is challenging as it requires the integration of application performance monitoring solutions with resource usage monitors of the operating system in order to obtain resource demand values. Operating system monitors often provide only aggregate resource usage statistics on a per-process level. However, many applications (e.g., web and application servers) serve different types of requests with one or more processes.

Profiling tools [20, 21], typically used during development to track down performance issues, provide information on call paths and execution times of individual functions. These profiling tools rely on either fine-grained code instrumentation or statistical sampling. However, these tools typically incur high measurement overheads, severely limiting their usage in production environments, and leading to inaccurate or biased results. In order to avoid distorted measurements due to overheads, [34,35] propose a two-step approach. In the first step, dynamic program analysis is used to determine the number and types of bytecode instructions executed by a function. In a second step, the individual bytecode instructions are benchmarked to determine their computational overhead. However, this approach is not applicable during operation and fails to capture interactions between individual bytecode instructions. APM tools enable fine-grained monitoring of the control flow of an application, including timings of individual operations. These tools are optimized to be also applicable to production systems.

Modern operating systems provide facilities to track the consumed CPU time of individual threads. This information is, for example, also exposed by the Java Runtime Environment and can be used to measure the CPU resource consumption by individual requests as demonstrated for Java in [10] and at the operating system level in [2]. This requires application instrumentation to track which threads are involved in the processing of a request. This can be difficult in heterogeneous environments using different middleware systems, database systems, and application frameworks. The accuracy of such an approach heavily depends on the accuracy of the CPU time accounting by the operating system and the extent to which request processing can be captured through instrumentation.

Over the years, a number of approaches to estimate the resource demands using statistical methods have been proposed. These approaches are typically based on a combination of aggregate resource usage statistics (e.g., CPU utilization) and

Pre-print version for personal use only!

coarse-grained application statistics (e.g., end-to-end application response times or throughput). These approaches do not depend on a fine-grained instrumentation of the application and are therefore widely applicable to different types of systems and applications incurring only insignificant overheads. Different approaches from queuing theory and statistical methods have been proposed [43], e.g., response time approximation least-squares regression, robust regression techniques, cluster-wise regression, Kalman Filter, adaptive filtering, Bayesian estimation, optimization techniques, Support Vector Machines, Independent Component Analysis, Maximum Likelihood Estimation, and Gibbs Sampling. These approaches differ in their required input measurement data, their underlying modeling assumptions, their output metrics, their robustness to anomalies in the input data, and their computational overhead. A detailed analysis and comparison is provided in [43], where a library for resource demand estimation - LibReDE - is described.

We identify the following areas of future research on resource demand estimation:

1. Current work is mainly focused on CPU resources. More work is required to address the specifics of other resource types, such as memory, network, or I/O devices. The challenge with these resource types is, among others, that resource utilization is often not as clearly defined as for CPU, and the resource access may be asynchronous. For instance, utilization of a storage I/O device w.r.t. throughput is hard to quantify since the maximum Input/Output Operations Per Second of a device is workload dependent itself.
2. Comparisons between statistical estimation techniques and direct measurement approaches are missing. This would help to better understand their implications on accuracy and overhead.
3. Most approaches are focused on estimating the mean resource demand. However, in order to obtain reliable performance predictions it is also important to determine the correct distribution of the resource demands.
4. Modern system features (e.g., multi-core CPU, dynamic frequency scaling, virtualization) can have a significant impact on the resource demand estimation.

The extraction of information about the interactions between components differs for design time and run-time. At design time, models can be created based on designer expertise and design documents as proposed in [42], [38], [41], and [14]. The automated extraction of structural information based on monitoring logs has the advantage that it tracks the behavior of the actual product as executed at run-time. An *effective architecture* can be extracted which means that only executed system elements are extracted [27]. Furthermore, runtime monitoring data enables to extract branching probabilities for different call paths [6,10]. The approaches for extraction of information about inter-component interactions by [23], [5], and [27] use monitoring information based on probes injected in the beginning of each response and propagated through the system.

Pre-print version for personal use only!

17.5 Adaptation

Software systems and surroundings are continuously subject to change (e.g., hardware breakdown, workload increase or decrease). Therefore, performance models have to be maintained up-to-date and the system has to be adapted to guarantee the satisfaction of performance requirements. In this section, we face the problem of maintaining performance models in online scenarios,continuously refining and calibrating them the performance models to allow them to better fulfill the purpose for which they are used. First of all, in Section 17.5.1, we discuss adaptation points which specify what may be changed within a system. Section 17.5.2 is about detection of changes in the system (e.g., hardware breakdown) and its surroundings (e.g., workload). Section 17.5.3 is about model-based reconfiguration in general, followed by the description of two exemplary reconfiguration frameworks in Section 17.5.4.

17.5.1 Adaptation Points

To support suitable automatic adaptations of a self-aware system, it has to be defined what, in the system, can be subject to change and what not. First, one has to define what changes shall or can be detected. Secondly, one has to define how the system may adapt itself. Therefore, it is necessary to identify adaptation points in the system and respectively in its model. The points where the system architecture can be adapted can be formalized in an adaptation point meta-model, as proposed in [24]. Thereby, this sub-model reflects the boundaries of the systems configuration space; i.e., it defines the possible valid states of the system architecture. The adaptation points at the model level correspond to adaptation operations executable on the real system at run-time, e.g., adding virtual CPUs to VMs, migrating VMs or software components, load-balancing requests, variation of algorithms or the size of a thread-pool. In general, the detection of the change points cannot be executed automatically since they are typically limited by constraints imposed by the execution environment, design choices (e.g., the usage of monolithic component/service) or even by business issues (e.g., in case of infrastructure-as-service we have a maximum amount of resources that we can use fixed by the contract). Adaptation point models are application specific and research on adaption point extraction is in its infancy. For example, the authors are not aware of an extraction mechanism that says if a component may be replicated or not.

In the future, guidelines or semi-automatic approaches should improve adaption point model creation. However, whenever identified and formally specified, such adaption points can enter the online adaptation mechanisms in order to select actions respecting them and hence to be considered in the adaptation.

Pre-print version for personal use only!

17.5.2 Detection of Changes

To detect changes in the system and its surrounding, monitoring infrastructures can be used to capture the occurrences of relevant events. Even in large and distributed systems, it is possible to generate, combine and filter huge amounts of events to timely detect changes in the behavior of the systems. Among existing monitoring infrastructures, it is worth to mention Glimpse [4], a flexible monitoring infrastructure, developed with the goal of decoupling the event specification from the monitoring and analysis mechanism. Glimpse was initially proposed to support behavioral learning, performance and reliability assessment, security and trust management. Many changes can be detected directly using event logs, e.g., deploy and undeploy or allocation and deallocation of hardware resources [22]. In such cases, an incoming event triggers a model update directly. However, there are changes in the real world that require changes of the model that cannot be mapped to an event directly. For instance, the behavior of software components often depends on parameters that are not available as input parameters passed upon service invocation. Such parameters are not traceable over the service interface and tracing them requires looking beyond the component boundaries [6]. For example, parameters might be passed to another component in the call path and/or they might be stored in a database structure queried by the invoked service. Moreover, the behavior of component services may also depend on the state of data containers such as caches or on persistent data stored in a database. For example, databases often behave differently for different load levels due to caching behavior. Such changes, non-traceable by events, require a continuous periodic surveillance of the system and a validation and adaptation of the model both with respect to its structural [22] and parameterization aspects [18]. Note that the prediction of workload changes is covered in detail in Chapter 18. so we refer the interested reader to that chapter.

17.5.3 Adaptation Mechanisms

Besides externally driven changes, systems may proactively change according to model-driven reconfigurations based on predictive analysis. This system-triggered change idea is close to the self-aware computing vision as it includes deduction of future states. The automated and dynamic nature of the reconfiguration process poses new challenges on the decision step that aims at choosing the next system configuration in order to overcome the observed problem. Most approaches use predetermined strategies coded in the application or in the reconfiguration framework [16]. However, in QoS management, a predetermined schema of decision can prevent the implementation of smart alternatives more suitable to effectively overcome the observed problems. Compared to threshold-based approaches, the use of predictive system performance models improves the adaptation process. It allows the choice of the system reconfiguration alternative that is predicted to satisfy the performance requirements of the system [11, 12, 19, 36].

In case of self-aware resource management in data-centers, a general framework observes the software application during its execution to monitor performance attributes of the software application. Whenever the performance constraints are no longer satisfied, the adaptation management process will start. The monitoring data is evaluated in order to identify the performance problem and the portion of the system affected by it. Such information is used to plan changes in the system configuration in order to overcome the observed problem. Whenever a new system configuration is determined, the changes are enacted and the system configuration is modified accordingly. Such general framework has been realized in different application domains (e.g., in service oriented software, in cloud computing, in component based systems).

At design time the full design space can be explored [31]. At run-time it is not possible to explore all design alternatives. Instead of a full exploration, greedy approaches are common. For example, in the Performance Management Framework (PMF) [11], the configuration alternatives are built on-the-fly by applying reconfiguration policies suitable for the application. At PMF, the initial performance model is specified by performance specialists. The next ones instead are generated on-the-fly by modifying the current performance model during the evaluation of the reconfiguration policies. Finally, the information collected during the monitoring phase is used to evaluate the predictive performance model(s).

The description of the alternatives needs to be simple in order to reduce the overhead of the model generation and evaluation, and, hence, of the decision step. This simplification may have impact on the accuracy of the measured performance indices, but the evaluation should be accurate enough for the choice of the reconfiguration alternative. The intuition here is that all the alternatives are represented at the same level of abstraction and the actual data are observed through the same abstractions thus providing a uniform workbench to consistently evaluate different alternatives. In [11], this intuition is confirmed by an empirical experimentation that showed that the chosen alternative was indeed the best among the generated ones. There are some major open issues in online model adaptation.

1. When should the reconfiguration be performed? The condition that triggers the adaptation process is a very critical issue in run-time performance management. It influences the execution frequency of the reconfiguration loop. Conditions that are verified too often lead to a high overhead: the management framework can consume more resources than the application itself. Conversely, conditions that rarely trigger the reconfiguration can prevent a timely management of performance problems. The critical issue here is to determine the best trade-off between computational overhead and timely resolution of performance problems.
2. Cost-benefit analysis for the next reconfiguration step. In order to be effective, the reconfiguration process must actually improve the performance of the managed system. Indeed, complex systems must address several non-functional requirements. The risk of having a degradation of some other non-functional property (e.g., security) related to the reconfiguration is avoided by allowing only a controlled set of configuration alternatives, which are decided by the de-

Pre-print version for personal use only!

Pre-print version for personal use only!

veloper according to the risks associated with the reconfiguration. Moreover, at each reconfiguration step, the costs to place the system in the new selected configuration, should be considered during the selection. This can be achieved by combining the result of the model evaluation provided by the solver with a coefficient representing the cost of the reconfiguration process.
3. Extensive search for the best reconfiguration versus finding a sufficient reconfiguration very fast.

Whenever a system and the corresponding performance model are adapted, the monitoring infrastructure that supports the adaptation framework could be subject to adaptation itself. For example: reploying some components may require redeploying the probes as well; dynamic binding of a different service would also change the catch and monitor event. As a consequence, an adaptable monitoring infrastructure able to reflect the system changes must be used. An example of such a flexible monitoring infrastructure can be found in [28]. However, this opens a wide research area that is out of scope of this chapter.

17.5.4 Model-based Adaptation Frameworks

In this section, we discuss two frameworks for model-based system adaptation at run-time. Section 17.5.4.1 presents the Performance Management Framework [11] and Section 17.5.4.2 the Strategies/Tactics/Actions [24] framework. Extraction of system structure and calibration will be mostly left out for complexity reasons.

17.5.4.1 Performance Management Framework

Performance Management Framework (PMF) [11] is an environment that focuses on run-time management of performance requirements of complex software systems. It monitors the current performance of the application and, when some problem occurs, it chooses a new configuration based on the feedback provided by the online evaluation of the performance models corresponding to different reconfiguration alternatives. The main characteristic of PMF is the heuristic mechanism to generate such alternatives. Differently from other approaches it does not rely on a fixed repository of predefined configurations but on a reconfiguration policy defined as a suitable combination of basic reconfiguration rules. Such basic reconfiguration rules guarantee the validity of the final reconfiguration policy with respect to external constraints (e.g, usage of legacy systems, usage of resources).

The reconfiguration policy is evaluated on the data retrieved by the online monitoring (that represents a snapshot of the current system state), thus generating a number of new configurations. Once such alternatives have been generated, the online evaluation is carried out to predict which one is most suitable to solve the observed problem.

The use of predictive system performance models improves the reconfiguration process. It allows the choice of the system reconfiguration alternative that guarantees the performance constrains and shows, in the predictive analysis, better performance. However, the run-time evaluation of predictive models representing the software systems poses strong requirements on the models themselves as discussed later.

The PMF approach is based on: i) monitoring of the running system to collect data, ii) dynamic reconfiguration to change the running configuration, and iii) model-based performance analysis to decide the next system configuration among the available ones.

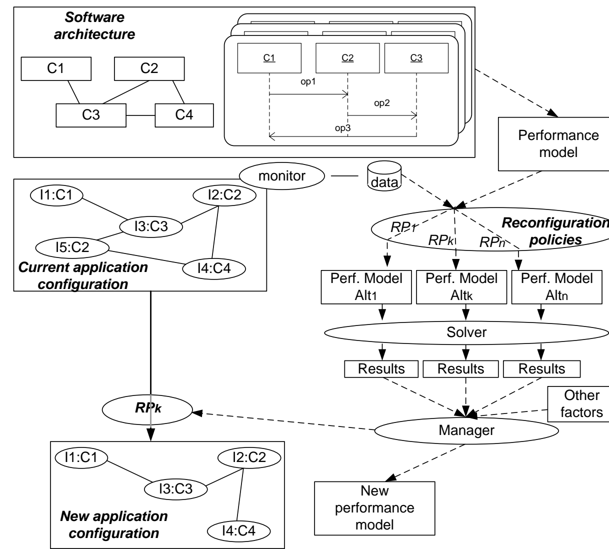

Fig. 17.1: The Performance Management Framework (PMF).

Figure 17.1 outlines the PMF process and its flow of activities. PMF observes the software application during its execution to monitor performance attributes of the software application. Whenever the performance constraints are no longer satisfied, the adaptation management process is triggered. The monitoring data is evaluated in order to identify the performance problem and the portion of the system affected by it. Such information is used to plan changes in the system configuration in order to overcome the observed problem. Whenever a new system configuration is determined, the changes are enacted and the system configuration is modified accordingly.

In the PMF, the configuration alternatives are built on-the-fly by applying reconfiguration policies suitable for the application. The initial performance model is, in general, specified by performance specialists. The next ones instead are generated on-the-fly by modifying the current performance model during the evaluation of the

Pre-print version for personal use only!

Pre-print version for personal use only!

reconfiguration policies. Finally, the information collected during the monitoring phase is used to evaluate the predictive performance model(s).

The description of configuration alternatives must be simple in order to reduce the overhead of the model generation and evaluation, and, hence, of the decision step. This simplification may have impact on the accuracy of the measured performance indices, but the evaluation should be accurate enough for the choice of the reconfiguration alternative. The intuition here is that all the alternatives are represented at the same level of abstraction and the actual data are observed through the same abstractions thus providing a uniform workbench to consistently evaluate different alternatives. In [11], this intuition is confirmed by an empirical experimentation that showed that the chosen alternative was indeed the best among the generated ones.

Application of PMF on SIENA publish/subscribe middleware

In [11], PMF has been used to dynamically reconfigure the SIENA middleware [13] topology depending on the *utilization* and *throughput* of SIENA routers. According to the general publish/subscribe model introduced above, the SIENA architecture (depicted in Figure 17.2) defines two main entities: (i) the *clients* and (ii) the *event-service*. Clients may be both *publishers* (i.e., objects of interest) and *subscribers* (i.e., recipients) that express their interest in certain kinds of events by supplying a *filter*. The event-service, composed of one or more servers interconnected in a hierarchical fashion (shown in Figure 17.3), forms a store-and-forward network that is responsible for delivering events from publishers to the subscribers that submitted a filter matching the respective events.

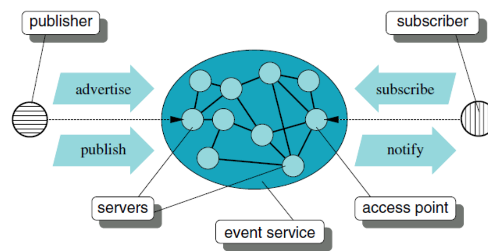

Fig. 17.2: SIENA Architecture.

The performance of a SIENA network depends on the performance of each SIENA server within the event-service and of the SIENA network topology. The performance of a SIENA router depends on the number of stored filters, as well as on the traffic generated by the clients connected to it. The SIENA network topology affects the routing of subscriptions and publications, and thus the global performance

Pre-print version for personal use only!

of the middleware. Indeed, since these indices are correlated by the utilization law,

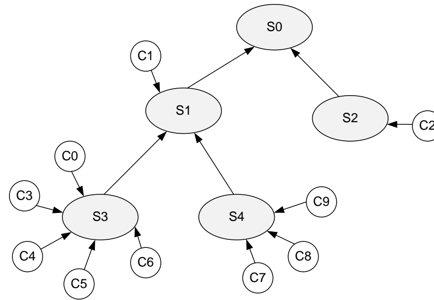

Fig. 17.3: A possible configuration for the SIENA network

in the experimental setup of SIENA publish subscribe middleware [11] we use the routers utilization as a basis to decide when a reconfiguration has to be performed.

Due to the event dispatcher rules and the dynamism of the network, it can happen that one or more routers are overloaded and degrade the performance of the whole network. For example, a router might be the access point of too many clients or it might be the root of a large sub-network, while other routers are unloaded (in this case the hierarchical structure is not balanced). When a SIENA router is overloaded (i.e., its utilization is high), we aim to reconfigure the network in order to prevent critical performance scenarios. The possible policies to reconfigure the network are specified as follows:

Moving SIENA clients – One or more SIENA clients are moved from the overloaded SIENA router to the unloaded one(s). This policy aims at balancing the workload among the routers. Note that, in order to obtain a significant improvement, the receiving routers must not belong to the sub-hierarchy of the overloaded router.

Changing SIENA routers' internal parameters – The router implementation allows the modification of the number of its internal threads satisfying the service requests of external software entities. In this way, it is possible to add (software) processing capabilities to each router.

Changing SIENA routers topology – One or more routers are switched from the overloaded router to the unloaded ones. This policy aims at balancing the workload among the routers switching them from one master to another. Again, to reach an improvement, the reconfigured routers must be attached to a master that does not belong to the sub-hierarchy of the overloaded router.

Adding/removing SIENA routers – The last possible reconfiguration policy is to remove/add router instances in order to increase/decrease the processing capacity of the network. Of course, we add new router instances if we need more (software) processing capacity, whereas we remove router instances whenever there are too many routers with respect to the needs.

Pre-print version for personal use only!

The online adaptation of the performance model has to respect the changing requirements of the software system the model represents. Moreover, a reconfiguration is normally not intended to change the application functionality (e.g., the substitution of a component with a new one providing different services). This restriction is necessary since functional changes in the application would normally imply a re-design of the performance model, and not only a change in its topology or in some parameters. Consequently, the reconfiguration process could not be realized in a completely automated manner.

In line with the above observations, in PMF [11] the allowed model reconfigurations are of two kinds: they may change internal parameters of software components (such as the number of threads or other features defined by the component developer); they may change the system topology by adding/removing component and/or connector instances. To relax this restriction, the online reconfiguration of models should rely on a database containing several different implementations of a component together with their performance models. When the reconfiguration policy requires the substitution of an implementation of a component, the adaptation of the performance model is done by replacing the sub-model of the first implementation with the one of the new implementation retrieved from the database.

PMF has been the first approach in literature to self-adapt system that uses performance models at run-time. In PMF, the predictive models are used to support the decision of how to reconfigure the system to overcome performance problems observed through the monitoring. In this way, the selected reconfiguration guarantees the performance requirements satisfaction until both system and environment characteristics do not change considerably.

17.5.4.2 S/T/A Adaptation Framework

In this section, we illustrate, how an architectural performance model of a software system (modeled with the Descartes Modeling Language (DML) described in Chapter 16 or [30]) can be updated and kept in sync with the real system using the Strategies/Tactics/Actions (S/T/A) adaptation framework [24]. We demonstrate this based on an industrial case study. The example model was created as part of a cooperation with Blue Yonder GmbH & Co. KG, a leading service provider in the field of predictive analytics and big data. The modeled system (called Blue Yonder system) provides forecasting services used by customers for predicting, e.g., sales, costs, churn rates. These services are based on compute-intensive machine-learning techniques and subject to customer SLAs. In this case study, the DML models were used to predict the resource requirements for a given usage scenario and optimize the resource allocation to reduce costs. Figure 17.4 depicts an excerpt of a DML model in a UML-like notation. Blue Yonders system consist of three types of components: `Gateway Server`, `Database`, and `Prediction Server`. These components run on a heterogeneous resource environment composed of low-cost desktop computers and high-end machines. The `Prediction Server` provides two services: `train` and `predict`. The `train` service infers a mathematical

Pre-print version for personal use only!

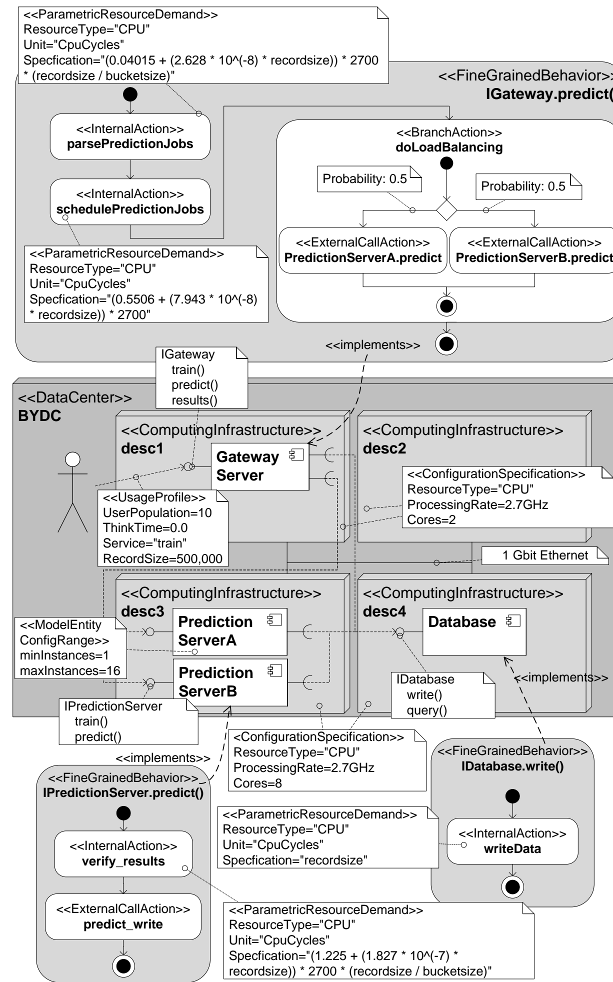

Fig. 17.4: Excerpt of the DML BlueYonder System Model

model for the available historical data. The `predict` service uses this model to return forecasts.

To enable performance predictions, services have to be equipped with model variables (e.g., branching probabilities, resource demands). For model parameterization, DML supports a hybrid approach: all model variables can be declared as either *explicit* or *empirical*. The model parameters in the example model in Figure 17.4 are all explicit, i.e., the values of the parameters are defined at model creation time. Empirical model parameters need to be learned based on monitoring data. Thus, it is possible to specify some model parameters based on expertise knowledge in advance while others can be learned from monitoring data collected

Pre-print version for personal use only!

at system run-time. Further, the model in Figure 17.4 includes an example for an adaptation point. Adaptation points can be either associated with model parameters (e.g., number of CPU cores) or with model entities. In the example model, the adaptation point (see `ModelEntityConfigRange`) is associated with a component specifying the minimum and maximum number of instances of this component that are allowed. In the example, the `PS` component is instantiated twice (`PredictionServerA` and `PredictionServerB`). The depicted configuration can be changed using adaptation processes. Figure 17.5 shows a schematic representation of an adaptation process for the Blue Yonder system. The objective is

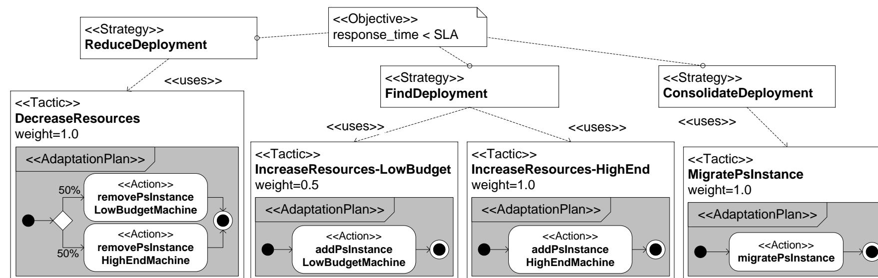

Fig. 17.5: Adaptation process for the Blue Yonder system.

to keep the response time below a certain SLA (while at the same time optimize the resource efficiency). In order to achieve this objective, the adaptation mechanism uses the following strategies: `FindDeployment`, `ReduceDeployment`, and `ConsolidateDeployment`. The `FindDeployment` strategy launches new `PS` instances until all customer SLAs are fulfilled. It contains two different tactics starting the new `PS` instances on low-budget machines or on high-end machines, respectively. `ReduceDeployment` removes unnecessary `PS` instances from machines to save operating costs, e.g., if the workload of a customer has decreased. Finally, `ConsolidateDeployment` migrates `PS` instances between machines with the goal to improve efficiency.

If a strategy contains different tactics to reach a certain objective, S/T/A assigns weights to the tactics, which are dynamically updated based on their predicted impact. The tactic with the highest weight is always applied first until the objective is reached or the tactic's weight is decreased and another tactic becomes the tactic with the highest weight. An educated setting of initial weights may speed up convergence, however, it is not required.

The reconfiguration planning step ends when the adaptation process has determined a series of reconfigurations that results in all application objectives being fulfilled. The reconfigurations are then applied on the real system. Thus, the model-based adaptation ensures that all reconfigurations are first applied on the model level, and their impact is predicted before applying them on the real system.

Pre-print version for personal use only!

DML can be used to predict the impact of changes in the workload and system configuration on the performance and resource usage of the application. DML is a descriptive model designed for high expressiveness and good understandability. In order to enable performance predictions, DML relies on mathematical analysis techniques based on existing stochastic modeling formalisms, such as (Layered) Queueing Networks (LQNs) and Queueing Petri Nets (QPNs). In [6], three different model-to-model transformations are defined providing different levels of prediction speed and accuracy: (a) bounds analysis uses operational analysis from queueing theory to determine asymptotic bounds on the average throughput and response time, (b) an LQN solver offering fast analytical model solution, and (c) a QPN solver supporting the analysis of larger models using simulation. The benefit of the transformation approach is that it provides the flexibility to switch between different prediction techniques depending on the prediction goals. The prediction goals comprise the requested performance metrics, the required accuracy and the time constraints. In [6], an algorithm is described that automatically selects a suitable transformation depending on the prediction goals. This algorithm is able to tailor the input model and remove parts that are not relevant for predicting the requested performance metrics. Thus, the model complexity can be reduced to speed up the analysis.

Another important issue is how to determine when the model-based adaptation should be triggered. At Blue Yonder, users have to book services in advance which allows to predict future incoming requests. In a general setting, enabling proactive adaptations of a system requires the use of forecasting methods to predict changes in the workload (described by the usage profile in DML). Chapter 18 introduces such forecasting methods, which can be combined with DML to detect changes in the usage profile. If workload changes are detected, the previously described mechanisms can be used to determine if and how the system should be reconfigured

17.6 Conclusion and Open Challenges

In order to be self-aware, a system needs to create an internal model representation of itself. The extraction of performance models yields an abstraction of the real system capturing only a subset of factors influencing the performance of the system. We briefly discuss the open challenges for completely automating the model extraction process. Existing research on automated model extraction is mostly based on small case studies and the majority of approaches proposed in the literature have not yet been validated in the context of large real-life systems. Improvements can be achieved by developing extraction tool benchmarks (which are currently missing) and by defining specific extraction tool design goals. Existing model extraction approaches differ in accuracy, granularity, and update behavior. So far, there are almost no comparisons between different approaches. Besides this, we identify the following challenges for performance model extraction:

Pre-print version for personal use only!

- Testing if the extracted model accurately reflects the system behavior. The assessment of validity and accuracy of extracted models is often based on trial and error. An improvement would be to equip models with confidence intervals that provide hints on their validity.
- Models may become outdated if they are not updated as the system evolves. Change point detection mechanisms are required to learn when models get out of date and when to update them.
- Devise guidelines or semi-automatic approaches for the extraction of adaptation points to derive actions for online adaptation.
- Current performance modeling formalisms barely ensure the traceability between models and the systems they represent. Explicit traceability information should be stored as part of the models.
- The automated inspection of the system under test often requires technology specific solutions. One approach to enable tools that are less technology-specific might be a definition of self-descriptive resources using standardized interfaces.
- A system can be modeled at different granularity levels. Extraction approaches usually support only one. Automated identification of an appropriate model granularity level and model reduction techniques are promising research areas.
- The automated identification and extraction of parametric dependencies in call paths and resource demands would enable significant improvements in the prediction accuracy of extracted models. Basic approaches, based on static code analysis, have been proposed in [33].
- Model extraction should support parallelism (multi-core systems and asynchronous calls) [32].

References

1. Mahmoud Awad and Daniel A. Menascé. *Computer Performance Engineering: 11th European Workshop, EPEW 2014, Florence, Italy, September 11-12, 2014. Proceedings*, chapter On the Predictive Properties of Performance Models Derived through Input-Output Relationships, pages 89–103. Springer International Publishing, Cham, 2014.
2. Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI'04)*, pages 18–18. USENIX Association, 2004.
3. Steffen Becker, Michael Hauck, Mircea Trifu, Klaus Krogmann, and Jan Kofron. Reverse engineering component models for quality predictions. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*, pages 199–202. IEEE, 2010.
4. Antonia Bertolino, Antonello Calabrò, Francesca Lonetti, and Antonino Sabetta. Glimpse: A generic and flexible monitoring infrastructure. In *Proceedings of the 13th European Workshop on Dependable Computing*, EWDC '11, pages 73–78, New York, NY, USA, 2011. ACM.
5. Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions of Software Engineering*, 32(9):642–663, 2006.
6. Fabian Brosig. *Architecture-Level Software Performance Models for Online Performance Prediction*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2014.

Pre-print version for personal use only!

7. Fabian Brosig, Nikolaus Huber, and Samuel Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Proceedings of the 26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*, 2011.
8. Fabian Brosig, Samuel Kounev, and Klaus Krogmann. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *Proceedings of the 1st International Workshop on Run-time mOdels for Self-managing Systems and Applications (ROSSA 2009)*. ACM, 2009.
9. Andreas Brunnert, Andre van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, Anne Koziolek, Johannes Kroß, Simon Spinner, Christian Vögele, Jürgen Walter, and Alexander Wert. Performance-oriented DevOps: A research agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), 2015.
10. Andreas Brunnert, Christian Vgele, and Helmut Krcmar. Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications. In MariaSimonetta Balsamo, WilliamJ. Knottenbelt, and Andrea Marin, editors, *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin Heidelberg, 2013.
11. Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi. Model-based system reconfiguration for dynamic performance management. *Journal of Systems and Software*, 80(4):455–473, 2007.
12. Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. MOSES: A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Trans. Software Eng.*, 38(5):1138–1159, 2012.
13. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.
14. Vittorio Cortellessa, Antinisca Di Marco, and Paola Inveradi. *Model-based software performance analysis*. Springer-Verlag, 2011.
15. Marc Courtois and Murray Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2nd International Workshop on Software and Performance (WOSP '00)*, pages 105–114. ACM, 2000.
16. Antinisca Di Marco, Paola Inverardi, and Romina Spalazzese. Synthesizing self-adaptive connectors meeting functional and performance concerns. In Marin Litoiu and John Mylopoulos, editors, *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2013, San Francisco, CA, USA, May 20-21, 2013*, pages 133–142. IEEE Computer Society, 2013.
17. Lei Ding and Nenad Medvidovic. Focus: A light-weight, incremental approach to software architecture recovery and evolution. In *In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, pages 191–200. IEEE, 2001.
18. Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering, (ICSE 2009)*, pages 111–121. IEEE, 2009.
19. Carlo Ghezzi, Valerio Panzica La Manna, Alfredo Motta, and Giordano Tamburrelli. Performance-driven dynamic service selection. *Concurrency and Computation: Practice and Experience*, 27(3):633–650, 2015.
20. Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.
21. Robert J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering (ICSE '92)*, pages 296–306. ACM, 1992.
22. Robert Heinrich, Eric Schmieders, Reiner Jung, Kiana Rostami, Andreas Metzger, Wilhelm Hasselbring, Ralf Reussner, and Klaus Pohl. Integrating run-time observations and design component models for cloud system analysis. In *Proceedings of the 9th Workshop on Models@run.time co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014.*, pages 41–46, 2014.

Pre-print version for personal use only!

23. Curtis E. Hrischuk, C. Murray Woodside, Jerome A. Rolia, and Rod Iversen. Trace-based load characterization for generating performance software models. *IEEE Transactions of Software Engineering*, 25(1):122–135, January 1999.
24. Nikolaus Huber, André van Hoorn, Anne Koziolek, Fabian Brosig, and Samuel Kounev. Modeling Run-Time Adaptation at the System Architecture Level in Dynamic Service-Oriented Environments. *Service Oriented Computing and Applications Journal (SOCA)*, 8(1):73–89, March 2014.
25. Marianne Huchard, A. Djamel Seriai, and Alae-Eddine El Hamdouni. Component-based architecture recovery from object-oriented systems via relational concept analysis. In *Proceedings of the 7th International Conference on Concept Lattices and Their Applications (CLA 2010)*, pages 259–270, 2010.
26. Hyperic. Hyperic (2014). http://www.hyperic.com, 2014.
27. Tauseef Israr, Murray Woodside, and Greg Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software*, 80(4):474–492, 2007.
28. Gregory Katsaros, George Kousiouris, Spyridon V. Gogouvitis, Dimosthenis Kyriazis, Andreas Menychtas, and Theodora Varvarigou. A self-adaptive hierarchical monitoring mechanism for clouds. *Journal of Systems and Software*, 85(5):1029 – 1041, 2012.
29. Samuel Kounev, Konstantin Bender, Fabian Brosig, Nikolaus Huber, and Russell Okamoto. Automated simulation-based capacity planning for enterprise data fabrics. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools '11)*, pages 27–36. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.
30. Samuel Kounev, Fabian Brosig, and Nikolaus Huber. The Descartes Modeling Language. Technical report, Department of Computer Science, University of Wuerzburg, 2014.
31. Anne Koziolek, Heiko Koziolek, and Ralf Reussner. PerOpteryx: automated application of tactics in multi-objective software architecture optimization. In Ivica Crnkovic, Judith A. Stafford, Dorina C. Petriu, Jens Happe, and Paola Inverardi, editors, *Joint proceedings of the Seventh International ACM SIGSOFT Conference on the Quality of Software Architectures and the 2nd ACM SIGSOFT International Symposium on Architecting Critical Systems (QoSA-ISARCS 2011)*, pages 33–42. ACM, New York, NY, USA, 2011.
32. Heiko Koziolek, Steffen Becker, Jens Happe, Petr Tuma, and Thijmen de Gooijer. Towards Software Performance Engineering for Multicore and Manycore Systems. *SIGMETRICS Perform. Eval. Rev.*, 41(3):2–11, December 2013.
33. Klaus Krogmann. *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2010.
34. Michael Kuperberg, Martin Krogmann, and Ralf Reussner. ByCounter: Portable runtime counting of bytecode instructions and method invocations. In *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2008.
35. Michael Kuperberg, Martin Krogmann, and Ralf Reussner. TimerMeter: Quantifying accuracy of software times for system analysis. In *Proceedings of the 6th International Conference on Quantitative Evaluation of SysTems (QEST) 2009*, 2009.
36. Moreno Marzolla and Raffaela Mirandola. Performance aware reconfiguration of software systems. In Alessandro Aldini, Marco Bernardo, Luciano Bononi, and Vittorio Cortellessa, editors, *Computer Performance Engineering - 7th European Performance Engineering Workshop, EPEW 2010, Bertinoro, Italy, September 23-24, 2010. Proceedings*, volume 6342 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2010.
37. Daniel A. Menascé, Mohamed Bennani, and H.Honglei Ruan. On the use of online analytic performance models, in self-managing and self-organizing computer systems. In *Self-star Properties in Complex Information Systems*, pages 128–142, 2005.
38. Daniel A. Menascé and Hassan Gomaa. A method for design and performance modeling of client/server systems. *IEEE Transactions of Software Engineering*, 26(11):1066–1085, 2000.
39. Daniel A. Menascé, Honglei Ruan, and Hassan Gomaa. QoS management in service-oriented architectures. *Perform. Eval.*, 64(7-8):646–663, 2007.

Pre-print version for personal use only!

40. Adrian Mos. *A framework for adaptive monitoring and performance management of component-based enterprise applications*. PhD thesis, Dublin City Universit, 2004.
41. Dorin C. Petriu and C. Murray Woodside. Software performance models from system scenarios in use case maps. In *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS '02)*, pages 141–158. Springer-Verlag, 2002.
42. Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.
43. Simon Spinner, Giuliano Casale, Xiaoyun Zhu, and Samuel Kounev. LibReDE: A library for resource demand estimation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE '14)*, pages 227–228. ACM, 2014.
44. Markus von Detten. Archimetrix: A tool for deficiency-aware software architecture reconstruction. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering (WCRE '12)*, pages 503–504. IEEE Computer Society, 2012.
45. Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 190–199, 2012.
46. Xiuping Wu and Murray Woodside. Performance modeling from software components. In *Proceedings of the 4th International Workshop on Software and Performance (WOSP '04)*, pages 290–301. ACM, 2004.
47. Zenoss. Zenoss (2014). http://www.zenoss.com, 2014.
48. Zhuoyao Zhang, L. Cherkasova, and Boon Thau Loo. Automating platform selection for mapreduce processing in the cloud. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, pages 125–136, Sept 2015.