Model-based Techniques for Performance Engineering of Business Information Systems

Samuel Kounev[1], Nikolaus Huber[1], Simon Spinner[2], and Fabian Brosig[1]

[1] Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
{kounev|fabian.brosig|nikolaus.huber}@kit.edu
[2] FZI Research Center for Information Technology
Haid-und-Neu-Straße 10
76131 Karlsruhe, Germany
spinner@fzi.de

Abstract. With the increasing adoption of virtualization and the transition towards Cloud Computing platforms, modern business information systems are becoming increasingly complex and dynamic. This raises the challenge of guaranteeing system performance and scalability while at the same time ensuring efficient resource usage. In this paper, we present a historical perspective on the evolution of model-based performance engineering techniques for business information systems focusing on the major developments over the past several decades that have shaped the field. We survey the state-of-the-art on performance modeling and management approaches discussing the ongoing efforts in the community to increasingly bridge the gap between high-level business services and low level performance models. Finally, we wrap up with an outlook on the emergence of self-aware systems engineering as a new research area at the intersection of several computer science disciplines.

Key words: business information systems, performance, scalability, predictive modeling, simulation

1 Introduction

Modern business information systems are expected to satisfy increasingly stringent performance and scalability requirements. Most generally, the *performance* of a system refers to the degree to which the system meets its objectives for timeliness and the efficiency with which it achieves this [55, 25]. Timeliness is normally measured in terms of meeting certain response time and/or throughput requirements, *response time* referring to the time required to respond to a user request (e.g., a Web service call or a database transaction), and *throughput* referring to the number of requests or jobs processed per unit of time. Scalability, on the other hand, is understood as the ability of the system to continue to meet its objectives for response time and throughput as the demand for the services it provides increases and resources (typically hardware) are added. Numerous studies, e.g., in the areas of e-business, manufacturing, telecommunications, health care and transportation, have shown that a failure to meet performance requirements can lead to serious financial losses, loss of customers and reputation, and in some cases even to loss of human lives. To avoid the pitfalls of inadequate Quality-of-Service (QoS), it is important to analyze the expected performance and scalability characteristics of systems during all phases of their life cycle. The methods used to do this are part of the discipline called *Performance Engineering*. Performance Engineering helps to estimate the level of performance a system can achieve and provides recommendations to realize the optimal performance level. The latter is typically done by means of performance models (e.g., analytical queueing models or simulation models) that are used to predict the performance of the system under the expected workload.

However, as systems grow in size and complexity, estimating their performance becomes a more and more challenging task. Modern business information systems based on the Service-Oriented Architecture (SOA) paradigm are often composed of multiple independent *services* each implementing a specific business activity. Services are accessed according to specified workflows representing business processes. Each service is implemented using a set of software components distributed over physical tiers as depicted in Figure 1. Three tiers exist: presentation tier, business logic tier, and data tier. The presentation tier includes Web servers hosting Web components that implement the presentation logic of the application. The business logic tier normally includes a cluster of application servers hosting business logic components that implement the business logic of the application. Middleware platforms such as Java EE, Microsoft .NET, or Apache Tomcat are often used in this tier to simplify application development by leveraging some common services typically used in enterprise applications. Finally, the data tier includes database servers and legacy systems that provide data management services.

The inherent complexity of such architectures makes it difficult to manage their end-to-end performance and scalability. To avoid performance problems, it is essential that systems are subjected to rigorous performance evaluation during the various stages of their lifecycle. At every stage, performance evaluation is conducted with a specific set of goals and constraints. The goals can be classified in the following categories, some of which partially overlap:

Platform selection: Determine which hardware and software platforms would provide the best scalability and cost/performance ratio?

Platform validation: Validate a selected combination of platforms to ensure that taken together they provide adequate performance and scalability.

Evaluation of design alternatives: Evaluate the relative performance, scalability and costs of alternative system designs and architectures.

Performance prediction: Predict the performance of the system for a given workload and configuration scenario.

Performance tuning: Analyze the effect of various deployment settings and tuning parameters on the system performance and find their optimal values.

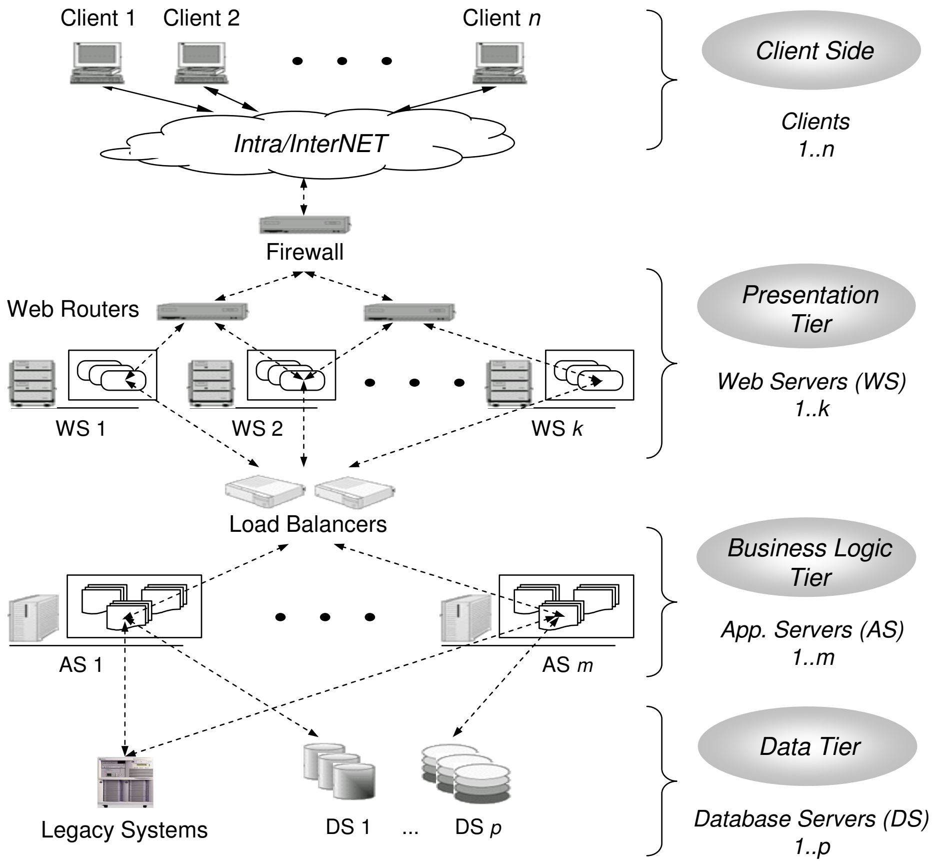


Fig. 1. Modern business information system.

Performance optimization: Find the components with the largest effect on performance and study the performance gains from optimizing them.

Scalability and bottleneck analysis: Study the performance of the system as the load increases and more hardware is added. Find which system components are most utilized and investigate if they are potential bottlenecks.

Sizing and capacity planning: Determine how much hardware resources are required to guarantee certain performance levels.

Run-time performance and power management: Determine how to vary resource allocations during operation in order to ensure that performance requirements are continuously satisfied while optimizing power consumption in the face of frequent variations in service workloads.

Two broad approaches are used in Performance Engineering for performance evaluation of software systems: performance measurement and performance modeling. In the first approach, load testing tools and benchmarks are used to generate artificial workloads on the system and to measure its performance. In the second approach, performance models are built and then used to analyze the performance and scalability characteristics of the system.

In this paper, we focus on performance modeling since it is normally much cheaper than load testing and has the advantage that it can also be applied in the early stages of system development before the system is available for testing. We present a historical perspective on the evolution of performance modeling techniques for business information systems over the past several decades, focusing on the major developments that have shaped the field, such as the increasing integration of software-related aspects into performance models, the increasing parametrization of models to foster model reuse, the increasing use of automated model-to-model transformations to bridge the gap between models at different levels of abstraction, and finally the increasing use of models at run-time for online performance management.

The paper starts with an overview of classical performance modeling approaches which is followed by an overview of approaches to integrate performance modeling and prediction techniques into the software engineering process. Next, automated model-to-model transformations from architecture-level performance models to classical stochastic performance models are surveyed. Finally, the use of models at run-time for online performance management is discussed and the paper is wrapped up with some concluding remarks.

2 Classical Performance Modeling

The performance modeling approach to software performance evaluation is based on using mathematical or simulation models to predict the system performance under load. A *performance model* is an abstract representation of the system that relates the workload parameters with the system configuration and captures the main factors that determine the system performance [45].

A number of different methods and techniques have been proposed in the literature for modeling software systems and predicting their performance under load. Most of them, however, are based on the same general methodology that proceeds through the steps depicted in Figure 2 [46, 55, 24]. First, the goals and objectives of the modeling study are specified. After this, the system is described in detail in terms of its hardware and software architecture. Next, the workload of the system is characterized and a workload model is built. The workload model is used as a basis for building a performance model. Before the model can be used for performance prediction, it has to be validated. This is done by comparing performance metrics predicted by the model with measurements on the real system obtained in a small testing environment. If the predicted values do not match the measured values within an acceptable level of accuracy, the model must be refined and/or calibrated. Finally, the validated performance model is used to predict the system performance for the deployment configurations and workload scenarios of interest. The model predictions are analyzed and used to address the goals set in the beginning of the modeling study.

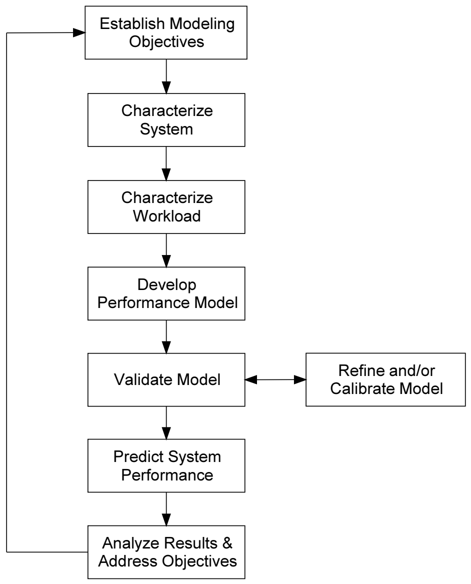

Establish Modeling Objectives
Characterize System
Characterize Workload
Develop Performance Model
Validate Model
Refine and/or Calibrate Model
Predict System Performance
Analyze Results & Address Objectives

Fig. 2. Performance modeling process.

2.1 Workload Characterization

Workload characterization is the process of describing the workload of the system in a qualitative and quantitative manner [44]. The result of workload characterization is a nonexecutable workload model that can be used as input to performance models. Workload characterization usually involves the following activities [55, 43]:

1. *The basic components of the workload are identified.* Basic component refers to a generic unit of work that arrives at the system from an external source [42]. Some examples include HTTP requests, Web service invocations, database transactions, and batch jobs. The choice of basic components and the decision how granular they are defined depend on the nature of the services provided by the system and on the modeling objectives.
2. *Basic components are partitioned into workload classes.* To improve the representativeness of the workload model, the basic components are partitioned into classes (called *workload classes*) that have similar characteristics. The partitioning can be done based on different criteria, depending on the type of system modeled and the goals of the modeling effort [42, 48].
3. *The system components and resources used by each workload class are identified.* For example, an online request to place an order might require using a Web server, application server, and backend database server. For each server, the concrete hardware and software resources used must be identified and characterized.
4. *The inter-component interactions and processing steps are described.* The aim of this step is to described the processing steps, the inter-component

interactions, and the flow of control for each workload class. Also for each processing step, the hardware and software resources used are specified.

5. *Service demands and workload intensities are quantified.* The goal is to quantify the load placed by the workload components on the system. *Service demand parameters* specify the average total amount of service time required by each workload class at each resource. *Workload-intensity parameters* provide for each workload class a measure of the number of units of work, that contend for system resources.

One of the greatest challenges in workload characterization is to obtain values for service demand parameters. Most techniques require the availability of a system to take measurements. If this is not possible, some techniques can also be used to estimate service demand parameters in the early stages of system development before the system is available for testing [47]. The process to obtain service demands through measurements at a running systems usually consists of three steps. First, the performance metrics that need to be monitored for quantifying service demands are selected. It is usually not possible to measure the service demands directly. Instead, the service demands must be derived from other metrics, which can be readily observed at the system. Typical metrics are the aggregate CPU utilization, the throughput and the transaction response time. Second, measurement data for the selected metrics is gathered from the system. This is usually done either by running controlled experiments in a test environment or by monitoring production systems while serving real workloads. Finally, the measurement data gathered in the previous step is analyzed and transformed in order to derive service demands.

A common approach to derive service demands from indirect measurements is based on the Service Demand Law [46]. This operational law states that the service demand $D_{i,r}$ of class r transactions at resource i is equal to the average utilization $U_{i,r}$ of resource i by class r transactions divided by the average throughput $X_{0,r}$ of class r transactions during the measurement interval, i.e.

$$D_{i,r} = \frac{U_{i,r}}{X_{0,r}}. \tag{1}$$

However, system monitors usually provide only statistics of the overall utilization of a resource aggregated over all workload classes. There are two approaches to determine values for $U_{i,r}$: conduct a single experiment injecting transactions from all workload classes simultaneously or conduct several experiments injecting only transactions of a single workload class at a time. In the former case, interactions between workload classes are not included in the service demands. In the latter case, methods to apportion the measured total utilization between workload classes are required, as described in [42, 45, 46]. However, there is often some unattributed resource usage due to system overheads. It is hard to find a fair distribution of the unattributed resource usage between workload classes [23].

Other approaches to resource demand estimation have been proposed over the years, e.g., based on linear regression [3, 51, 49], general optimization techniques [62, 35, 1], or Kalman filters [63, 33]. Each of these approaches makes certain assumptions, e.g., regarding the type, amount and quality of measurements. The decision which of these estimation approaches can be used depends heavily on the modeled system.

2.2 Stochastic Performance Models

Performance models have been employed for performance prediction of software systems since the early seventies. In 1971, Buzen proposed modeling systems using queueing network models and developed solution techniques for several important classes of models. Since then many advances have been made in improving the model expressiveness and developing efficient model analysis techniques as well as accurate approximation techniques. A number of modeling techniques utilizing a range of different performance models have been proposed including standard queueing networks, extended and layered queueing networks, stochastic Petri nets, queueing Petri nets, stochastic process algebras, Markov chains, statistical regression models and simulation models. Performance models can be grouped into two main categories: *simulation models* and *analytical models*. One of the greatest challenges in building a good model is to find the right level of abstraction and granularity. A general rule of thumb is: Make the model as simple as possible, but not simpler! Including too much detail might render the model intractable, on the other hand, making it too simple might render it unrepresentative.

Simulation Models. Simulation models are software programs that mimic the behavior of a system as requests arrive and get processed at the various system resources. Such models are normally stochastic because they have one or more random variables as input (e.g., the times between successive arrivals of requests). The structure of a simulation program is based on the states of the simulated system and events that cause the system state to change. When implemented, simulation programs count events and record the duration of time spent in different states. Based on these data, performance metrics of interest (e.g., the average time a request takes to complete or the average system throughput) can be estimated at the end of the simulation run. Estimates are provided in the form of confidence intervals. A confidence interval is a range with a given probability that the estimated performance metric lies within this range. The main advantage of simulation models is that they are very general and can be made as accurate as desired. However, this accuracy comes at the cost of the time taken to develop and run the models. Usually, many long runs are required to obtain estimates of needed performance measures with reasonable confidence levels.

Several approaches to developing a simulation model exist. The most time-consuming approach is to use a general purpose programming language such as C++ or Java, possibly augmented by simulation libraries (e.g., CSIMor SimPack, OMNeT++, DESMO-J). Another approach is to use a specialized simulation language such as GPSS/H, Simscript II.5, or MODSIM III. Finally, some simulation packages support graphical languages for defining simulation models (e.g., Arena, Extend, SES/workbench, QPME). A comprehensive treatment of simulation techniques can be found in [34, 2].

Analytical Models. Analytical models are based on mathematical laws and computational algorithms used to derive performance metrics from model parameters. Analytical models are usually less expensive to build and more efficient to analyze compared to simulation models. However, because they are defined at a higher level of abstraction, they are normally less detailed and accurate. Moreover, for models to be mathematically tractable, usually many simplifying assumptions need to be made impairing the model representativeness. Queueing networks and generalized stochastic Petri nets are perhaps the two most popular types of models used in practice.

Queueing networks provide a very powerful mechanism for modeling hardware contention (contention for CPU time, disk access, and other hardware resources). A number of efficient analysis methods have been developed for a class of queueing networks called product-form queueing networks allowing models of realistic size and complexity to be analyzed with a minimum overhead [9]. The downside of queueing networks is that they do not provide direct means to model software contention aspects accurately (contention for processes, threads, database connections, and other software resources), as well as blocking, simultaneous resource possession, asynchronous processing, and synchronization aspects. Even though extensions of queueing networks, such as extended queueing networks [37] and layered queueing networks (also called stochastic rendezvous networks) [60], provide some support for modeling software contention and synchronization aspects, they are often restrictive and inaccurate.

In contrast to queueing networks, generalized stochastic Petri net models can easily express software contention, simultaneous resource possession, asynchronous processing, and synchronization aspects. Their major disadvantage, however, is that they do not provide any means for direct representation of scheduling strategies. The attempts to eliminate this disadvantage have led to the emergence of queueing Petri nets [4], which combine the modeling power and expressiveness of queueing networks and stochastic Petri nets. Queueing Petri nets enable the integration of hardware and software aspects of system behavior in the same model [28]. A major hurdle to the practical use of queueing Petri nets, however, is that their analysis suffers from the state space explosion problem limiting the size of the models that can be solved. Currently, the only way to circumvent this problem is by using simulation for model analysis [29].

Details of the various types of analytical models are beyond the scope of this article. The following books can be used as reference for additional information [9, 57, 5]. The Proceedings of the ACM SIGMETRICS Conferences and the Performance Evaluation Journal report recent research results in performance modeling and evaluation. Further relevant information can be found in the Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE), the Proceedings of the International Conference on Quantitative Evaluation of SysTems (QEST), the Proceedings of the Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of

Computer and Telecommunication Systems (MASCOTS), and the Proceedings of the International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS).

3 Software Performance Engineering

A major hurdle to the adoption of classical performance modeling approaches in industry is the fact that performance models are expensive to build and require extensive experience and expertise in stochastic modeling which software engineers typically do not possess. To address this issue, over the last fifteen years, a number of approaches have been proposed for integrating performance modeling and prediction techniques into the software engineering process. Furthermore, using models and automatic transformation and processing can simplify the software performance engineering approach, making it less error-prone.

3.1 Software Performance Meta-Models

Efforts introducing performance models in the software engineering process were initiated with Smith's seminal work pioneered under the name of *Software Performance Engineering (SPE)* [53]. Since then a number of languages (i.e., meta-models) for describing performance-relevant aspects of software architectures and execution environments have been developed by the SPE community, the most prominent being the UML SPT profile (UML Profile for Schedulability, Performance and Time) and its successor the UML MARTE profile (UML Profile for Modeling and Analysis of Real-time and Embedded Systems). The latter are extensions of UML (Unified Modeling Language) as the de facto standard modeling language for software architectures. Other proposed architecture-level performance meta-models include SPE-MM [54], CSM [50] and KLAPER [17]. The common goal of these efforts is to enable the automated transformation of architecture-level performance models into analytical or simulation-based performance models that can be solved using classical analysis techniques (see Section 3.2).

In recent years, with the increasing adoption of Component-Based Software Engineering (CBSE), the SPE community has focused on adapting and extending conventional SPE techniques to support component-based systems. A number of architecture-level performance meta-models for component-based systems have been proposed as surveyed in [31]. Such meta-models provide means to describe the performance-relevant aspects of software components (e.g., internal control flow and resource demands) while explicitly capturing the influences of their execution context. The idea is that once component models are built they can be reused in multiple applications and execution contexts. The performance of a component-based system can be predicted by means of compositional analysis techniques based on the performance models of its components. Over the last five years, research efforts have been targeted at increasing the level of parametrization of component models to capture additional aspects of their execution context.

An example of a mature modeling language for component-based systems is given by the Palladio Component Model (PCM) [6]. In PCM, the component execution context is parameterized to explicitly capture the influence of the component's connections to other components, its allocated hardware and software resources, and its usage profile including service input parameters. Model artifacts are divided among the developer roles involved in the CBSE process, i.e., component developers, system architects, system deployers and domain experts.

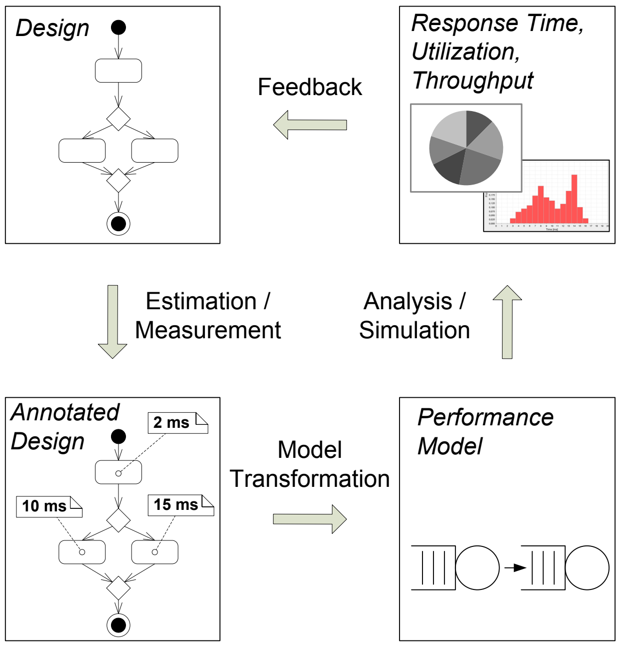


Fig. 3. Model-driven Performance Engineering Process

3.2 Model-to-Model Transformations

To bridge the gap between architecture-level performance models and classical stochastic performance models, over the past decade the SPE community has focused on building automated model-to-model transformations (see Figure 3) which make it possible to exploit existing model solution techniques from the performance evaluation community [39]. In the following, we provide an overview of the most common transformations available in the literature.

Marco and Inverardi transform UML models annotated with SPT stereotypes into a multichain queueing network [38]. UML-ψ, the UML Performance SImulator [40], transforms a UML instance annotated with the SPT profile to a simulation model. The results from the analysis of the simulation model are reported back to the annotated UML instance [39]. Another approach uses the stochastic process algebra PEPA as analysis model [56]. In this case, only UML activity diagrams are considered, which are annotated with a subset of the MARTE profile. A software tool implementing this method is also available. Bertolino and Mirandola integrate their approach into the Argo-UML modeling tool, using the RT-UML performance annotation profile [8]. An execution graph and a queueing network serve as the target analysis formalisms.

Other approaches use UML, but do not use standardized performance profile annotations: the approach in [18] uses XSLT, the eXtensible Stylesheet Language Transformations, to execute a graph pattern based transformation from a UML instance to LQNs. Instead of annotating the UML model, it has to be modeled in a way so that the transformation can identify the correct patterns in the model. The authors of [7] consider only UML state charts and sequence diagrams. A transformation written in Java turns the model into GSPN sub-models that are then combined into a final GSPN. Gomaa and Menascé use UML with custom XML performance annotation [16]. The performance model is not described in detail, but appears to be based on queueing networks. In [61], the authors use UML component models together with a custom XML component performance specification language. LQN solvers are used for the analysis.

Further approaches exist that are not based on UML: [11, 10] builds on the ROBOCOP component model and use proprietary simulation framework for model analysis. [15] proposes a custom control flow graph model notation and custom simulation framework. [20] employs the COMTEK component technology, coupled with a proprietary analysis framework. In [52], the authors specify component composition and performance characteristics using a variant of the big-O notation. The runtime analysis is not discussed in detail.

Several model-to-model transformations have been developed for the Palladio Component Model (PCM). Two solvers are based on a transformation to Layered Queueing Networks (LQNs) [32] and a transformation to Stochastic Regular Expressions [30], respectively. Stochastic Regular Expressions can be solved analytically with very low overhead, however, they only support single user scenarios. Henßproposes a PCM transformation to OMNeT++, focusing on a realistic network infrastructure closer to the OSI reference network model [19]. The PCM-Bench tool comes with the SimuCom simulator [6] which is based on a model-to-text transformation used to generate Java code that builds on DESMO-J, a general-purpose simulation framework. The code is then compiled on-the-fly and executed. SimuCom is tailored to support all of the PCM features directly and covers the whole PCM meta-model. Meier et al. present a transformation of the PCM to Queuing Petri Nets (QPN) [41]. This transformation enables the analysis of PCM model instances with simulation and analysis techniques developed for QPNs [29]. The work also illustrates and compares important aspects concerning the accuracy and overhead of the solvers for PCM model instances (SimuCom, LQNS and LQSim) and SimQPN, the solver for QPNs. The results show that LQN-based solvers are less accurate regarding mean response times and that solvers can have significantly different analysis overhead.

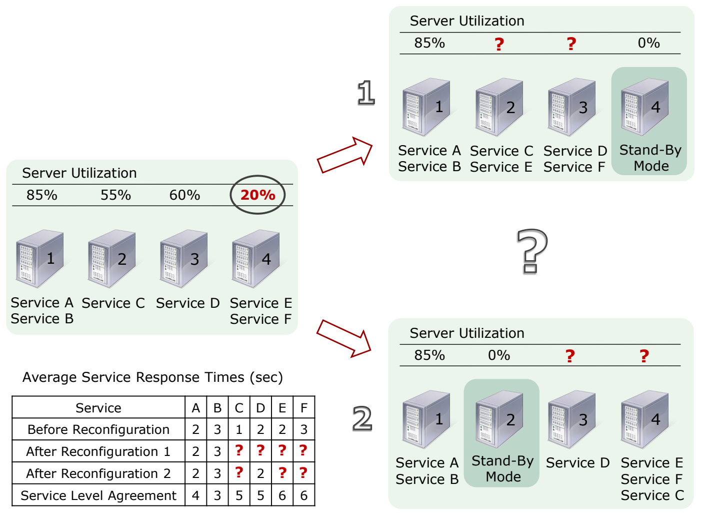
Finally, a number of intermediate languages (or kernel languages) for specifying software performance information have been proposed in the literature. The aim of such efforts is to reduce the overhead for building transformations, i.e., only $M + N$ instead of $M \cdot N$ transformations have to be developed for M source and N target meta-models [39]. Some examples of intermediate languages include SPE-MM [54], KLAPER (Kernel LAnguage for PErformance and Reliability analysis) [17] and CSM (Core Scenario Model) [50].

4 Run-Time Performance Management

With the increasing adoption of virtualization and the transition towards Cloud Computing platforms, modern business information systems are becoming increasingly complex and dynamic. The increased complexity is caused by the introduction of virtual resources and the resulting gap between logical and physical resource allocations. The increased dynamicity is caused by the complex interactions between the applications and services sharing the physical infrastructure. In a virtualized service-oriented environment changes are common during operation, e.g., new services and applications can be deployed on-the-fly, service workflows and business processes can be modified dynamically, hardware resources can be added and removed from the system, virtual machines (VMs) can be migrated between servers, resources allocated to VMs can be modified to reflect changes in service workloads and usage profiles. To ensure adequate performance and efficient resource utilization in such environments, capacity planning needs to be done on a regular basis during operation. This calls for *online* performance prediction mechanisms.

An example of a scenario where online performance prediction is needed is depicted in Figure 4. A service-oriented system made of four servers hosting six different services is shown including information on the average service response times, the response time service level agreements (SLAs) and the server utilization. Now assume that due to a change in the demand for services E and F, the average utilization of the fourth server has dropped down to 20% over an extended period of time. To improve the system's efficiency, it is considered to switch one of the servers to stand-by mode after migrating its services to other servers. Two possible ways to reconfigure the system are shown. To ensure that reconfiguring the system would not break the SLAs, the system needs a mechanism to predict the effect of the reconfiguration on the service response times.

Given the variety of changes that occur in modern service-oriented environments, online performance prediction techniques must support variations at all levels of the system including variations in service workloads and usage profiles, variations in the system architecture, as well as variations in the deployment and execution environment (virtualization, middleware, etc). To predict the impact of such variations, *architecture-level* performance models are needed at run-time



Average Service Response Times (sec)

Service	A	B	C	D	E	F
Before Reconfiguration	2	3	1	2	2	3
After Reconfiguration 1	2	3	?	?	?	?
After Reconfiguration 2	2	3	?	2	?	?
Service Level Agreement	4	3	5	5	6	6

Fig. 4. Online performance prediction scenario.

that explicitly capture the influences of the system architecture, its configuration, and its workload and usage profiles.

While many architecture-level performance prediction techniques exist in the literature, most of them suffer from two significant drawbacks which render them impractical for use at run-time: i) performance models provide limited support for reusability and customization, ii) performance models are static, creating and maintaining them manually during operation is prohibitively expensive [59].

While techniques for component-based performance engineering have contributed a lot to facilitate model reusability, there is still much work to be done on further parameterizing component models before they can be used for online performance prediction. In particular, current techniques do not provide means to model the layers of the component execution environment explicitly. The performance influences of the individual layers, the dependencies among them, as well as the resource allocations at each layer should be captured as part of the models. This is necessary in order to be able to predict at run-time how a change in the execution environment (e.g., modifying resource allocations at the virtualization layer) would affect the overall system performance.

As to the second issue indicated above, building architecture-level performance models that accurately capture the different aspects of system behavior is a challenging task and requires a lot of time when applied manually to large and complex real-world systems. Often, no explicit architecture documentation of the system exists and hence, the model must be built from scratch. Additionally, experiments and measurements must be conducted to parameterize the model such that it reflects the system behavior accurately.

Current performance analysis tools used in industry mostly focus on profiling and monitoring transaction response times and resource consumption. They often provide large amounts of low-level data while important information about the end-to-end performance behavior is missing (e.g., service control flow and resource demands).

In research, approaches such as [58, 13] use systematic measurements to build black-box mathematical models or models obtained with genetic optimization. However, these approaches are purely measurement-based, the models serve as interpolation of the measurements, and neither a representation of the system architecture nor its performance-relevant factors are extracted. Services are modeled as black boxes and many restrictive assumptions are often imposed such as a single workload class, single-threaded components, homogeneous servers or exponential request interarrival times. Given these limitations, such models are rarely applied in practice and instead ad hoc mechanisms for performance and resource management are employed. Performance-relevant details of the virtualization platform and the applications running inside the hosted virtual machines are not considered explicitly preventing detailed performance predictions which are necessary for efficient resource management. Approaches aiming at the extraction of architectural information are presented in, e.g., [21, 22, 12]. They use call path tracing, a form of dynamic analysis to gain reliable data on the actual execution of an application. However, the model extraction techniques for architecture-level performance models are usually not automated or applicable at run-time [21, 22] and they do not consider the virtualization layer explicitly [12].

Existing work concerning the quantification of virtualization overheads is mainly based on system benchmarking. The performance of several virtualization solutions such as Xen, VMware workstation, Linux-VServer, OpenVZ, etc. are compared. However, the focus is on the overall performance overhead, individual performance-influencing factors are not analyzed. In contrast, Lu et al. [36] present a calibration process based on application usage traces that covers the main resource types CPU, memory, network and disk I/O and is applicable at run-time. Approaches to automatically extract performance models of the virtualization layer are normally very specific and do not provide models which can be used for performance predictions at the application level. Therefore, there is a major need for an approach which addresses these deficiencies and combines methods for automated model extraction and performance prediction in virtualized environments at run-time.

The heart of the problem is in the fact that architecture-level performance models are normally designed for offline use and as such they are decoupled from the system components they represent. Models do not capture dynamic aspects of the environment and therefore they need to be updated manually after every change in the system configuration or workload. Given the frequency of such changes, the amount of effort involved in maintaining performance models is prohibitive and therefore in practice such models are rarely used after deployment.

Research Challenges The described limitations of the existing work lead to the following two general research challenges:

– Designing abstractions for modeling performance-relevant aspects of services in *dynamic virtualized* environments. The abstractions should be structured

around the system components involved in processing service requests. Individual layers of the software architecture and execution environment, context dependencies and dynamic system parameters should be modeled explicitly.
– Developing methods for the automatic online model extraction, maintenance, refinement and calibration *during operation*. This includes the efficient resolution of service context dependencies including dependencies between service input parameters, resource demands, invoked third-party services and control flow of underlying components.

To address these challenges, current research efforts are focusing on developing *online* architecture-level performance models designed specifically for use at run-time, e.g., the Descartes Meta-Model [14, 27]. Such models aim at capturing all information, both static and dynamic, relevant to predicting the system's performance on-the-fly. They are intended to be integrated into the system components and to be maintained and updated automatically by the underlying execution platform (virtualization and middleware) reflecting the evolving system environment.

Online performance models will make it possible to answer performance-related queries that arise during operation such as: What would be the effect on the performance of running applications if a new application is deployed in the virtualized infrastructure or an existing application is migrated from one physical server to another? How much resources need to be allocated to a newly deployed application to ensure that SLAs are satisfied? How should the system configuration be adapted to avoid performance issues or inefficient resource usage arising from changing customer workloads?

The ability to answer queries such as the above provides the basis for implementing techniques for *self-aware* performance and resource management [27]. Such techniques will be triggered automatically during operation in response to observed or forecast changes in application workloads. The goal will be to *proactively* adapt the system to such changes in order to avoid anticipated QoS problems or inefficient resource usage. The adaptation will be performed in an autonomic fashion by considering a set of possible system reconfiguration scenarios (e.g, changing VM placement and/or resource allocations) and exploiting the online performance models to predict the effect of such reconfigurations before making a decision.

Self-aware systems engineering [14, 26] is currently emerging as a new research area at the intersection of several computer science disciplines including software architecture, computer systems modeling, autonomic computing, distributed systems, and more recently, Cloud Computing and Green IT. It raises a number of big challenges that represent emerging hot topics in the systems engineering community and will be subject of long-term fundamental research in the years to come. The resolution of these challenges promises to revolutionize the field of systems engineering by enabling guaranteed QoS, lower operating costs and improved energy efficiency.

5 Concluding Remarks

We presented a historical perspective on the evolution of model-based performance engineering techniques for business information systems, focusing on the major developments over the past four decades that have shaped the field, such as the increasing integration of software-related aspects into performance models, the increasing parametrization of models to foster model reuse, the increasing use of automated model-to-model transformations to bridge the gap between models at different levels of abstraction, and finally the increasing use of models at run-time for online performance management. We surveyed the state-of-the-art on performance modeling and management approaches discussing the ongoing efforts in the community to increasingly bridge the gap between high-level business services and low level performance models. Finally, we concluded with an outlook on the emergence of self-aware systems engineering as a new research area at the intersection of several computer science disciplines.

References

1. Computing missing service demand parameters for performance models. In *CMG 2008*, pages 241–248, 2008.
2. J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation.* Prentice Hall, Upper Saddle River, NJ 07458, third edition, 2001.
3. Y. Bard and M. Shatzoff. Statistical methods in computer performance analysis. *Current Trends in Programming Methodology*, III, 1978.
4. F. Bause. Queueing Petri Nets - A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models, Toulouse, France, October 19-22*, 1993.
5. F. Bause and F. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory.* Vieweg Verlag, second edition, 2002.
6. S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Syst. and Softw.*, 82:3–22, 2009.
7. S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In *Proc. on WOSP '02*, pages 35–45, 2002.
8. A. Bertolino and R. Mirandola. CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Edinburgh, UK*, volume 3054 of *LNCS*, pages 233–248, May 2004.
9. G. Bolch, S. Greiner, H. D. Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications.* John Wiley & Sons, Inc., 2nd edition, Apr. 2006.
10. E. Bondarev, P. de With, M. Chaudron, and J. Muskens. Modelling of input-parameter dependency for performance predictions of component-based embedded systems. In *Proc. on EUROMICRO '05*, pages 36–43, 2005.
11. E. Bondarev, J. Muskens, P. de With, M. Chaudron, and J. Lukkien. Predicting real-time properties of component assemblies: a scenario-simulation approach. In *Proc. of the 30th Euromicro Conference*, pages 40–47, 2004.

12. F. Brosig, N. Huber, and S. Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*, Oread, Lawrence, Kansas, November 2011.
13. M. Courtois and M. Woodside. Using regression splines for software performance analysis. In *Proceedings of the International Workshop on Software and Performance*, 2000.
14. Descartes Research Group. http://www.descartes-research.net, December 2011.
15. E. Eskenazi, A. Fioukov, and D. Hammer. Performance prediction for component compositions. In *Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, pages 280–293. 2004.
16. H. Gomaa and D. Menascé. Performance engineering of component-based distributed software systems. In *Performance Engineering*, volume 2047 of *LNCS*, pages 40–55. 2001.
17. V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software*, 80(4):528–558, April 2007.
18. G. P. Gu and D. C. Petriu. XSLT transformation from UML models to LQN performance models. In *Proc. on WOSP '02*, pages 227–234, 2002.
19. J. Henss. Performance prediction for highly distributed systems. In *Proc. on WCOP '10*, volume 2010-14, pages 39–46. Karlsruhe Institue of Technology, 2010.
20. S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Packaging predictable assembly. In *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 108–124. 2002.
21. C. E. Hrischuk, M. Woodside, J. A. Rolia, and R. Iversen. Trace-Based Load Characterization for Generating Performance Software Models. *IEEE Trans. on Softw. Eng.*, 1999.
22. T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *J. Syst. Softw.*, 2007.
23. S. Kounev. *Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction.* PhD Thesis, Shaker Verlag, December 2005.
24. S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, July 2006.
25. S. Kounev. *Wiley Encyclopedia of Computer Science and Engineering*, chapter Software Performance Evaluation. Wiley-Interscience, John Wiley & Sons Inc, ISBN-10: 0471383937, ISBN-13: 978-0471383932, September 2008.
26. S. Kounev. Self-Aware Software and Systems Engineering: A Vision and Research Roadmap. In *GI Softwaretechnik-Trends, 31(4)*, Nov 2011.
27. S. Kounev, F. Brosig, N. Huber, and R. Reussner. Towards self-aware performance and resource management in modern service-oriented systems. In *Proc. of the 7th IEEE Intl. Conf. on Services Computing (SCC 2010)*. IEEE Computer Society, 2010.
28. S. Kounev and A. Buchmann. Performance Modelling of Distributed E-Business Applications using Queuing Petri Nets. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, 2003.
29. S. Kounev and A. Buchmann. SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4-5):364–394, May 2006.

30. H. Koziolek. *Parameter dependencies for reusable performance specifications of software components*. PhD thesis, University of Karlsruhe (TH), 2008.
31. H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, August 2009.
32. H. Koziolek and R. Reussner. A model transformation from the palladio component model to layered queueing networks. In *Proc. on SIPEW '08*, pages 58–78, 2008.
33. D. Kumar, A. Tantawi, and L. Zhang. Real-time performance modeling for adaptive software systems. In *VALUETOOLS '09: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
34. A. Law and D. W. Kelton. *Simulation Modeling and Analysis.* Mc Graw Hill Companies, Inc., third edition, 2000.
35. Z. Liu, L. Wynter, C. H. Xia, and F. Zhang. Parameter inference of queueing models for IT systems using end-to-end measurements. *Performance Evaluation*, 63(1):36–60, 2006.
36. L. Lu, H. Zhang, G. Jiang, H. Chen, K. Yoshihira, and E. Smirni. Untangling mixed information to calibrate resource utilization in virtual machines. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 151–160, New York, NY, USA, 2011. ACM.
37. E. A. MacNair. An introduction to the Research Queueing Package. In *WSC '85: Proceedings of the 17th conference on Winter simulation*, pages 257–262, New York, NY, USA, 1985. ACM Press.
38. A. D. Marco and P. Inverardi. Compositional generation of software architecture performance QN models. *Software Architecture, Working IEEE/IFIP Conf. on*, 0:37, 2004.
39. A. D. Marco and R. Mirandola. Model transformation in software performance engineering. In *QoSA*, 2006.
40. M. Marzolla and S. Balsamo. UML-PSI: The UML performance simulator. *Quantitative Eval. of Syst.*, 0:340–341, 2004.
41. P. Meier, S. Kounev, and H. Koziolek. Automated Transformation of Palladio Component Models to Queueing Petri Nets. In *In 19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011)*, Singapore, July 25-27 2011.
42. D. Menascé and V. Almeida. *Capacity Planning for Web Performance: Metrics, Models and Methods.* Prentice Hall, Upper Saddle River, NJ, 1998.
43. D. Menascé and V. Almeida. *Scaling for E-Business - Technologies, Models, Performance and Capacity Planning.* Prentice Hall, Upper Saddle River, NJ, 2000.
44. D. Menascé, V. Almeida, R. Fonseca, and M. Mendes. A Methodology for Workload Characterization of E-commerce Sites. In *Proceedings of the 1st ACM conference on Electronic commerce, Denver, Colorado, United States*, pages 119–128, Nov. 1999.
45. D. A. Menascé, V. Almeida, and L. W. Dowdy. *Capacity Planning and Performance Modeling - From Mainframes to Client-Server Systems.* Prentice Hall, Englewood Cliffs, NG, 1994.
46. D. A. Menascé, V. Almeida, and L. W. Dowdy. *Performance by Design.* Prentice Hall, 2004.
47. D. A. Menascé and H. Gomaa. A Method for Desigh and Performance Modeling of Client/Server Systems. *IEEE Transactions on Software Engineering*, 26(11), Nov. 2000.

48. J. Mohr and S. Penansky. A forecasting oriented workload characterization methodology. *CMG Transactions*, 36, June 1982.
49. G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi. CPU demand for web serving: Measurement analysis and dynamic estimation. *Performance Evaluation*, 65(6-7):531–553, 2008.
50. D. Petriu and M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and Systems Modeling (SoSyM)*, 6(2):163–184, June 2007.
51. J. Rolia and V. Vetland. Parameter estimation for performance models of distributed application systems. In *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, page 54. IBM Press, 1995.
52. M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy. Performance specification of software components. *SIGSOFT Softw. Eng. Notes*, 26(3):3–10, 2001.
53. C. U. Smith. *Performance Engineering of Software Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
54. C. U. Smith, C. M. Lladó, V. Cortellessa, A. Di Marco, and L. G. Williams. From UML models to software performance results: an SPE process based on XML interchange formats. In *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, pages 87–98, New York, NY, USA, 2005. ACM Press.
55. C. U. Smith and L. G. Williams. *Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software.* Addison-Wesley, 2002.
56. M. Tribastone and S. Gilmore. Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile. In *Proc. on WOSP '08*, 2008.
57. K. S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* John Wiley & Sons, Inc., second edition, 2002.
58. D. Westermann and J. Happe. Towards performance prediction of large enterprise applications based on systematic measurements. In *WCOP*, 2010.
59. M. Woodside, G. Franks, and D. Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE'07)*, pages 171–187, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
60. M. Woodside, J. Neilson, D. Petriu, and S. Majumdar. The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software. *IEEE Transactions on Computers*, 44(1):20–34, Jan. 1995.
61. X. Wu and M. Woodside. Performance modeling from software components. *SIGSOFT Softw. E. Notes*, 29(1):290–301, 2004.
62. L. Zhang, C. H. Xia, M. S. Squillante, and W. N. M. Iii. Workload service requirements analysis: A queueing network optimization approach. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '02, page 23ff, Washington, DC, USA, 2002. IEEE Computer Society.
63. T. Zheng, C. M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, 2008.