Improved Feedback for Architectural Performance Prediction Using Software Cartography Visualizations

Klaus Krogmann[1], Christian M. Schweda[2], Sabine Buckl[2], Michael Kuperberg[1], Anne Martens[1], and Florian Matthes[2]

[1] Software Design and Quality Group
Universität Karlsruhe (TH), Germany
{krogmann,mkuper,martens}@ipd.uka.de
[2] Software Engineering for Business Information Systems
Technische Universität München, Germany
{schweda,buckl,matthes}@in.tum.de

Abstract. Software performance engineering provides techniques to analyze and predict the performance (e.g., response time or resource utilization) of software systems to avoid implementations with insufficient performance. These techniques operate on models of software, often at an architectural level, to enable early, design-time predictions for evaluating design alternatives. Current software performance engineering approaches allow the prediction of performance at design time, but often provide cryptic results (e.g., lengths of queues). These prediction results can be hardly mapped back to the software architecture by humans, making it hard to derive the right design decisions. In this paper, we integrate *software cartography* (a map technique) with software performance engineering to overcome the limited interpretability of raw performance prediction results. Our approach is based on model transformations and a general software visualization approach. It provides an intuitive mapping of prediction results to the software architecture which simplifies design decisions. We successfully evaluated our approach in a quasi experiment involving 41 participants by comparing the correctness of performance-improving design decisions and participants' time effort using our novel approach to an existing software performance visualization.

1 Introduction

Performance is a complex and cross-cutting property of every software system. If performance targets (e.g., maximum response times) are not met, a redesign or even a reimplementation of the system is needed, which leads to significant costs. Model-based software performance prediction approaches (cf. [1,31]) estimate the performance of software architectures[1] at design time, before fully implementing them. These approaches are also referred to as *Software Performance Engineering* (SPE, [26]), and several implementations of it exist. Making the correct design decisions using SPE requires proper understanding and interpretation of the prediction results.

[1] We will use the term *architecture* to capture static structure, behavior, and deployment of a software.

Nevertheless, many companies do not use SPE in software development [4]. Woodside et al. [31] highlight the *limited interpretability* in current SPE approaches: "Better methods and tools for interpreting the results and diagnosing performance problems are a future goal". Mapping performance prediction results back to the analyzed software architecture is difficult because the underlying concept's abstract entities (e.g., places in Petri Nets [15] or queue lengths in Layered Queuing Networks [8,23]) are not directly linked to a software architecture. Also, performance prediction results are insufficiently aggregated so that performance data is at a lower abstraction level than architectural elements (e.g. only resource demands of single steps of behavior specifications are available, while reallocation decisions require resource demands at the component level). Both aspects result in high time demands for the identification of performance problems like bottlenecks. For example, the SPE tool of [26] presents the resource demands of single steps in the behavior specification ("software execution graph") as well as the overall demand of whole use cases or parts thereof ("scenarios"). Components are not reflected in the models. Thus, for decisions like reallocation, one needs to manually collect the results for all scenarios of the component.

The contribution of this paper is a novel integration of the approaches of *software performance engineering* (SPE) and *software cartography* to support design decisions by performance result visualizations at the level of software architectures. For SPE, our approach uses Palladio [2], a representative state-of-the-art model-based performance prediction approach. It is going to be integrated with software cartography [29], a scientific discipline which reuses cartographic techniques for visualizing large applications, i.e. software systems and their interconnections. Such a visualization technique is necessary as large applications form highly complex and interdependent systems, which are not easy to comprehend without graphical support. The visualizations are designed according to the viewpoints of individual stakeholders[2] to meet requirements of user convenience and usability.

The targeted benefits of the presented approach are a more intuitive result interpretation, a better usability, and thus the speedup of decision processes. These benefits are achieved by assisting the software architect through visualization of performance prediction results at an *architectural* level, which allows to map performance prediction results to the elements of an architectural software model (cf. Fig. 2 / 5). Also, multiple information layers are available to ease trade-off decisions and to help in identifying crosscutting design impacts. Our solution overcomes the limited interpretability of performance evaluation results as provided by Petri Nets, or Layered Queuing Networks. It transforms an architectural model, a prediction result model, and graph description models into a software architecture visualization which includes prediction results. The generality of our visualization approach allows different kinds of viewpoints.

We successfully evaluated the approach in a quasi experiment involving both experienced software performance analysts and less experienced computer science students. Our experiment shows an increased precision and effectiveness of design decisions processes, making it more likely to choose the right design options. The approach is applicable also for users with little experience in SPE.

[2] The terms *stakeholder* and *viewpoint* are used here in accordance to their definitions in [13].

In the remainder of this paper, Section 2 surveys related work, Section 3 describes the foundations of software cartography and visualizations while Section 4 introduces SPE. In Section 5, we present our novel integrated visualization approach, which is evaluated in Section 6 in a quasi experiment, before Section 7 concludes the paper.

2 Related Work

Model-based software performance prediction is surveyed by Balsamo et al. in [1]. In this area, the existing approaches either support (i) *result aggregation* or (ii) *links to the software architecture*, but no approach integrates both aspects, as described in the following. The commercial SPE-ED [26] tool by Smith et al. highlights critical actions in a behavioral model and highly utilized servers, and thus allows fast result interpretation for software architects. Still, there is no support in SPE-ED for high-level architectural constructs like composite components. In other commercial SPE tools like "Performance Optimizer" or "Capacity Manager" from Hyperformix [12], visualizations are mostly table-based or bar charts to illustrate impacts of changing deployment, but the visualizations are not connected to models of the software architecture. For Hyperformix products, only little information on visualization techniques is available, and the software itself is not freely available or affordable for universities to perform case studies or experiments. Of the approaches surveyed in [1], those that do link performance results back to architectural elements of the design model, like [30,14,5], still do not provide *aggregated* information on an architectural level, e.g. for components. The same is also true for similar newer approaches such as UML-Ψ [17].

Software model visualization in the field of software engineering is dominated by the unified modeling language (UML) [21], which provides the common basis for modeling single software systems. For performance modelling, the UML MARTE profile [20] has been suggested, which introduces performance-related tagged values for UML constructs. However, while the tagged values can be displayed in all UML diagram types, they do not provide a specialized visualization of the performance results (e.g. resource utilization visualization) to support fast insight and design decisions (e.g. reallocation of components to spread the load). Furthermore, the creation of different viewpoints according to the concerns of various stakeholders is not supported in UML (cf. [19]).

Visualization approaches of arbitrary models like *GMF* (Graphical Modeling Framework [6]) support the creation of graphical notations for arbitrary models (e.g., software or performance models), but put a special emphasis on creating graphical editors. In GMF, the user is granted a maximum amount of flexibility regarding the layout of visualizations – means for specifying layout rules are limited in the GMF approach. Also, GMF implies that the visualization of a specific concept uses a distinct unique type of visualization – to add other graphical elements for the visualization, GMF needs to create a *new* editor.

Other **software performance visualization approaches** [16,11,32,25,33] have already been proposed and they all support stakeholders in making design decisions to improve software performance. However, none of theses approaches supports *architectural* design decisions by visualizations. Instead, they usually focus on parallel programs and often require executable implementations to monitor the software behaviour.

All of these approaches have in common that their visualization is not empirically evaluated. An additional limitation is that their outdated GUI techniques cannot be integrated into modern IDEs like Eclipse. General performance evaluation tools (e.g. profilers or performance monitors) are also implementation-centric and must execute the finished application.

3 Software Cartography

Conventional cartography (making geographical maps; see e.g. [10]) provides techniques well-suited for presenting complex information to a wide variety of people from different educational backgrounds. These techniques range from color-coding according to property values to the separation of the *base map* and layered additions.

Software cartography [29] re-uses cartographic techniques for visualitions of complex interconnected systems, called *software maps*. These maps are stakeholder-specific visualizations, especially designed as means for management support. In analogy to *urban planning*, a city and a software application landscape share a number of characteristics [18]:

- They form networked, open systems with autonomous and active constituents.
- They are constantly evolving and (mostly) have no designated end of lifetime.
- Many people are involved as stakeholders, with different educational backgrounds.
- Different stakeholders have different concerns regarding the system, such that a balance of interests has to be achieved.

Especially the people-centric characteristics motivate the idea of using cartographic techniques for visualizing systems. Existing stakeholder-specific graphical notations for certain aspects, e.g. the diagram types introduced in UML for the software development process, are not widely known outside the respective domain – neither business architects nor system administrators are likely to understand UML sufficiently. Map-like visualization, although having no such well-defined semantics, can be more easily understood by the various stakeholders.

One might argue that software maps do not provide a well-defined semantics, as the same type of symbol (e.g. a rectangle) might have different meanings in different visualizations (i.e. software maps). While this is true, the meaning of a specific symbol on one software map is clarified using another cartographic technique, the *legend*. A legend in *conventional* maps provides textual information on the meaning of the symbols used in the map. Such information is also contained in a *software* map legend, which also comprises information on the meaning of (relative) positioning, as different maps can employ different positioning rules to express certain underlying information.

Relative positioning rules are especially of interest in the context of the base map, leading to a distinction between different types of base maps – the so-called *software map types*. There are three distinct basic types, of which one – a cluster map – is shown in Figure 1 as an example. The cluster map uses the principle of *clustering* (i.e. nesting of symbols into other symbols) to visualize relationships, preferably hierarchical ones. On this map, a distance measure between the visualized concepts emerges from the assignment of the concepts to parenting clusters – concepts in the same cluster are

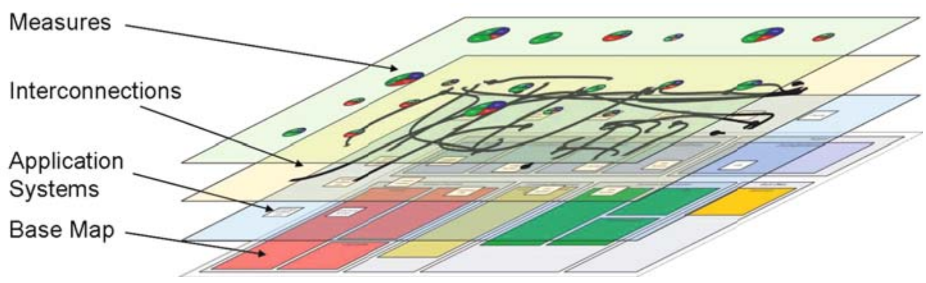

Fig. 1. Cluster map utilizing the layering principle

closer to each other than concepts in different clusters. Wittenburg [29] provides a more in-depth introduction to the software map types.

Regarding the involvement of various stakeholders, as for example in the context of performance predictions, the utilization of the so-called *layering principle* from cartography is especially beneficial. This concept allows the setup of a base map containing the logical constituents of the system under consideration, further utilizing relative positioning of symbols to represent certain types of associations as spatial relationships, which can be easily perceived. Additional information, only necessary for a certain stakeholder group, can be presented on an additional layer, which can be displayed or hidden on demand. Figure 1 illustrates this principle. In the context of software performance predictions, especially the layers showing *measures* and *interconnections* are of interest. The *measures* layer contains pie charts showing certain properties of the system under consideration on an aggregated, e.g. business-relevant, level. The actual interconnections, from which the properties of the system emerge, are contained in the *interconnections* layer.

In this setting, a stakeholder from a business group can identify a certain system based on the pie chart aggregation. Then the layer containing the interconnections can be overlaid to achieve a link to underlying technical information. By the utilization of the layering principle, it is ensured that the software system under consideration does not change its graphical position between the different viewpoints, which would not necessarily be the case, if stakeholder-specific visual notations were employed.

4 Software Performance Prediction and Visualization

It is significantly more expensive to deal with performance problems in an implemented software system than to *prevent* performance problems *before* they occur [26]. Thus, it is advantageous to *detect* performance deficiencies of a planned software system during the design phase. As actual performance measurements of the system are not possible during the design phase, the performance must be estimated (predicted) based on information available at that moment.

Performance prediction is also needed in other scenarios where measurements are impossible or cost too much effort. For example, to answer sizing questions on the server to be used in relation to an expected workload, performance prediction can help to evaluate and to make design decisions ("is it more effective to buy server X or to replace components C and D with ones having higher performance, but remaining on an old server Y?").

To answer such question on scalability, sizing etc., Software Performance Engineering (SPE [28]) is a *systematic* and well-studied approach for early estimation and prediction of software performance. As input, SPE approaches take a system's *usage model* (i.e. workload and expected user behavior) and an *architectural model* (e.g. in UML) which includes a static part (components and connectors), a behavioral part (e.g., a strong abstraction of control and data flow), the resource environment (available server and networks), and component allocation (a mapping between component and resource environment). SPE is used to assess the feasibility of given performance requirements; achieving the optimal performance is usually not an objective because of costs.

Visualization in SPE. The performance of an executed software application depends on its usage of *resources* (e.g. CPU, HDD, network). *Resource contention* occurs when competing requests to a shared resource have to wait. Resource contention leads to a significant performance impact, which must be accounted for during performance prediction. After transforming the architectural models into analysis models such as Queueing Networks, SPE approaches analyze performance by employing simulation or analytical solutions. In the analysis, the resources maintain queues, with resource requests waiting if the resource is busy. Simulation or analytical solution of the modeled resources allows to estimate performance metrics such as utilization, waiting time in the queue, and response time. The length of such a queue allows to draw conclusions about resource utilization *over time*, and to identify which resources are bottlenecks. To "feed back" SPE results into their architecture, SPE users must be able to map SPE results to the architectural models – but in practice, there is a significant gap between (usually formal) analysis models of SPE (such as Queuing Networks and Petri Nets), and the architectural models.

Furthermore, the results returned by SPE are often aggregated: the response time for a coarse-grained application service does not allow to easily conclude which of the used components is consuming the largest amount of CPU power. Similarly, the textual output that reports the usage percentages of several CPUs still means that the SPE user must manually map these results to the (usually graphical) deployment model. Finally, to compare different design options, the respective SPE prediction results for them need to be visualized so that the SPE user can comfortably compare them in a unified way.

Visualization Requirements. Thus, results that need to be visualized are not only "end-user" performance metrics (e.g. service response time), but also "internal" metrics, such as the utilization of a resource. Visualization should allow to detect performance problems and, based on this, to make the right design decisions. To increase software architects' effectiveness and correctness when designing with respect to performance, the visualization must be easy to understand. Apart from presenting single, isolated results, visualization of performance prediction results should consider *architecture relation*, *highlighting*, *correlation*, and *decomposition*:

– The visualization should offer the software architect the same components as in the design phase. Otherwise, the back-mapping of performance results gets ambiguous,

error-prone and as our studies in Section 6 show, it leads to less correct and more time-consuming design decisions. Consequently, results must be presented at the architectural level.
- To make it easier to match the performance prediction results and the architecture models, the aggregated results should be an overlay over the respective architecture models; an overlay can be textual (e.g., a tooltip showing the median utilization), or graphical (e.g., specific coloring of bottleneck resources).
- Visualizations that use multiple data dimensions (e.g., response times vs. more powerful resources) must be supported to allow trade-off analyses (e.g., cost of introducing cache vs. higher worst-case latency), and evaluation of cross-cutting concerns (e.g., enabling time-consuming security features).
- Multiple viewpoints (e.g., static architecture viewpoint vs. deployment viewpoint) of the same architectural prediction results should enable the stakeholder to delve into details. The software architect can then analyze an architecture with different focuses.

5 Integrated Approach

In our approach, we bring together visualizations from software cartography and performance predictions at the level of software architecture. This combination promises improved understanding of performance issues and the potential to easily optimize software design.

Palladio [2] is a model-based state-of-the-art SPE approach that features an integrated, Eclipse-based tool chain for modeling and evaluating software architectures. Palladio will be used in the quasi experiment (cf. [24, p. 4]) of our approach because we intend to study whether its visualization techniques must be enhanced, and which benefits the enhancements would provide. Internally, Palladio uses Queueing Networks for simulation-based performance prediction. Application models in Palladio are built on the Palladio Component Metamodel (PCM), following the paradigms of model-driven software development (MDSD).

In Palladio, raw performance prediction results are stored in a database. In the original visualization, data is basically accumulated to depict pie charts, bar charts/histograms, and line charts. However, the original visualization results are separated from the architecture model (the prediction results were simply *labeled* with service names, component names, and resource names). For example, the response time of a service is visualized as a general probability distribution of response times (cf. Fig. 2).

5.1 Chosen Visualizations in the New Approach

With the new, extended visualization, software architecture and performance prediction results are brought together in the visualization. For that, a new *decorator model* [9] annotates elements of a PCM model instance with prediction results. The decorator model contains prediction results for each component, server node, and network connection.

To realize the requirements listed at the end of section 4, we used four basic viewpoints to visualize performance results (these viewpoints are used in the evaluation

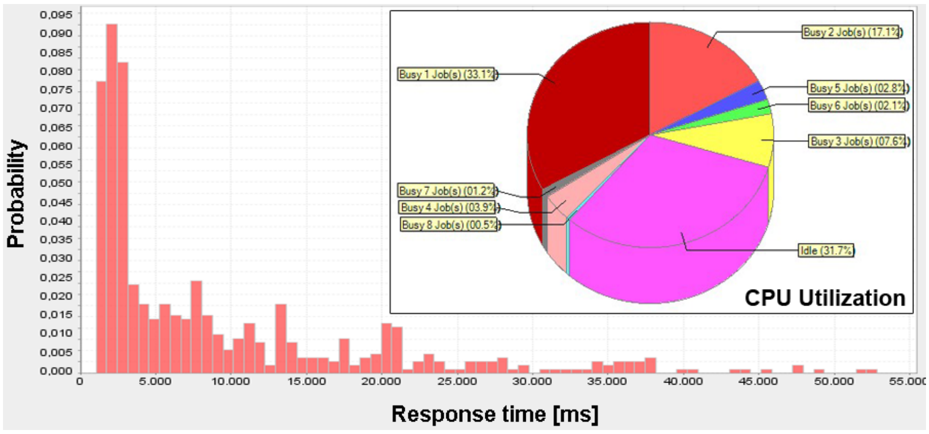

Fig. 2. Original result visualization in Palladio: Histogram of a general distribution function of service's response time and a pie chart showing the utilization of a resource (top right)

discussed in Section 6, where the actual visualizations are also shown; cf. Fig. 5) in our approach.

In the **black-box resource environment viewpoint**, server nodes and network connections are visualized together with their *utilization* (i.e. percentage of busy and idle times). In the visualization, strong network connections are thicker than slower ones. For the utilization of a server node, we use the weighted average of the utilization of its inner resources of one type (CPU, HDD, ...), weighting with their processing rate. Thus, we obtain a measure relative to the overall processing rate of all resources of this type. For passive resources such as thread pools, the utilization is analogously determined based on their capacity. If the server contains resources of different types (CPUs, HDDs, and thread pools), the utilization of the resource type with the highest average utilization is used, to allow fast bottleneck recognition.

In the **white-box resource environment viewpoint**, the first viewpoint is enhanced by visualizing server nodes and nested hardware resources (such as CPUs, hard disks, network connections, or thread pools) together with their respective utilization. Performance metrics here also include the *utilization per resource* (e.g. CPU or HDD) of a server node. In this viewpoint, also the average *wait time* (time for which requests are waiting in a queue; relative to the wait time of all resources in the model) and the average *demanded absolute time* per resource can be visualized. The distinction between server nodes and resources allows users to delve into details. If a server has multiple CPUs, we can detect that, for example, only one CPU is under heavy load, while others are idle. For scenarios in which scheduling can be influenced or where different types of CPUs are used, one can then try to shift computations to another CPU.

In the **white-box allocation viewpoint**, the deployment of software components into the execution environment is included into the visualization. In addition to server nodes and network connections between them, also all components for each server, the connection of components, their individual *resource demands* as well as the *absolute response time* are visualized. The resource demand is shown relative to the summed-up resource demands of all components on this server node (for composite components, also the internal resource demand is accumulated).

In the **software component viewpoint**, only software components, interfaces, and connectors between components are displayed. For each component the absolute response time is displayed.

Performance prediction result data is strongly aggregated with the intention to support understandability and to avoid overwhelming users with information. If the aggregated data is not sufficient, software architects and performance analysts can still delve into details using the original Palladio visualization, which provides access to specific results like the response time of individual services of components.

To ease the interpretation and speed of recognition, coloring indicates potential bottlenecks or critical architectural elements. *Components* with relative performance metrics (e.g., utilization) with more than 75% are colored red; while for *resources*, starting from 75% orange and starting from 85% red is used. Beyond coloring, also multiple data layers can be visualized for the same topology (same server nodes, network connections, and component allocation). In our approach, we use the data layers to visualize different usage scenarios to allow the estimation of impact for different usage scenarios (e.g., resource utilization with few vs. many users). Compared to the visualization requirements from Section 4, our visualization approach provides a) a relation between architecture and performance prediction results, b) highlights critical architectural elements, c) eases correlation analysis through multiple layers, and d) enables decomposition by multiple viewpoints at different levels of detail. We will detail on visualizations in the evaluation discussed in Section 6.

5.2 Applying the Software Cartography Approach to Visualizing Performance Information

Software maps, as introduced in Section 3, are consistent visualizations of information, i.e. a viewpoint on the information utilizing cartographic means. As creating these

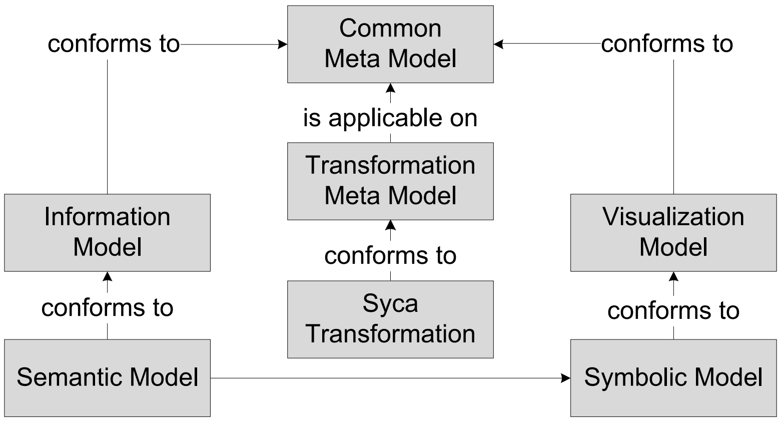

Fig. 3. Basic principle behind the Syca transformation approach to visualization generation

visualizations manually is both a time-consuming and error-prone process, an approach for generating software maps from input information is presented in [7]. This approach is based on object-oriented models of both the information and the visualization as well as a model-to-model transformation, the so-called *Syca transformation*. The core concepts of the approach are sketched in Figure 3 and lay the basis for the integration explained subsequently. Prior to details on the integration, the core models of the visualization generation approach are introduced:

Information model. This (meta) model se ts up the language for describing the modeling subject, i.e. it introduces the core concepts, which are used to create a model of the subject's reality. In the context of software performance prediction, the information model defines concepts such as *components*, *connectors*, and attributes such as *average utilization*. The elements contained in an object-oriented information model are *classes*, *attributes*, and *relationships*.

Semantic model. The semantic model contains instance data modeled according to the respective information model, i.e. it contains *information objects*. In the context of software performance prediction, these objects are instances of components and interconnections, having assigned values for the attributes.

Visualization model. This (meta) model defines graphical concepts, either visual ones – so called *map symbols* – or *visualization rules*, which describe graphical relationships between the symbols. These rules can be used to specify relative positionings, size, or the overall appearance of symbols at the level of graphical concepts without having to supply all details of positioning or layout.

Symbolic model. This model contains visualization concept instances, i.e. map symbols and visualization rules, modeled according to the respective visualization model. By assigning visualization rules to the symbols, the relative positioning is determined, although no absolute positions are specified.

All aforementioned models are described using the Meta Object Facility (MOF) [22]. This language therefore provides the common meta model for both information model and visualization model.

The common meta model further lays the basis for expressing a mapping from information model to visualization model concepts in terms of a model-to-model transformation. This Syca transformation can be used to specify how a concept from the information model, e.g. a component, should be visualized, e.g. as a rectangle. Executing the thus specified transformation, instances from the semantic model can be mapped to instances in a symbolic model, which together describe a visualization. The transformed symbolic model is subsequently handed over to a layouting component, which computes the actual positions and sizes of the symbols in accordance with the respective rules. For details on the layouting mechanism, see e.g. [7].

The above approach can handle arbitrary object-oriented information models and corresponding instance data. Therefore, it is possible to apply it on the performance prediction data (Result Decorator) as computed by the Palladio approach, which is also represented in a decorated object-oriented model. The basic make-up of a respective integration is sketched in Figure 4. Selected technical details of the integration are described below.

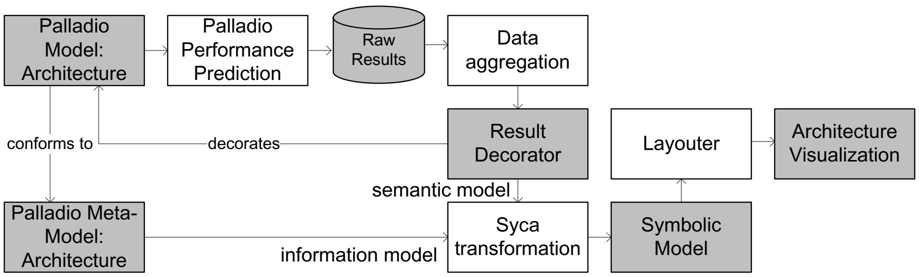

Fig. 4. Processing of models and prediction results

The model-transformation based approach for visualization generation has been implemented in a prototypic Eclipse based tool, the *SoCaTool* [3]. It allows the technical integration to the Palladio Bench, which is also realized based on the Eclipse platform. In this integration, the decorated results from the performance prediction are read as semantic models into the SoCaTool, which subsequently hands the data over to the Syca transformation component. From this, a generated symbolic model is passed to the layouter, which finally computes the absolute positioning of the symbols and thus creates a visualization. Central cornerstone of the technical integration is hence the realization of an appropriate Syca transformation. This transformation is built on the data contained in the decorated resource model.

For this paper, the existing Palladio and SoCaTool have been extended by decorator support, the Result Decorator itself, and the new Syca transformation. Screenshots of the realized visualization approach are available online[3].

6 Evaluation

To evaluate the chosen visualization of performance results, we conducted a quasi experiment involving students and faculty members, all from the field of computer science. Only about half of the faculty members was familiar with Palladio.

In the quasi experiment, we investigated four research questions:

Q1: Can participants make correct design decisions based on the visualization? To do so, the participants need to correctly identify performance bottlenecks and be able to solve performance issues (changing the architecture in the right way).
Q2: How long does it take the participants to evaluate a scenario?
Q3: How does the participants' experience in software performance influence the above metrics?
Q4: Do users of the new visualization perform better than users of the original Palladio visualization?

To evaluate the quality of the improved visualization with respect to these questions, we conducted the experiment in two phases. In the first phase, we studied all four

[3] http://sdqweb.ipd.uka.de/wiki/SoCa-Palladio

questions above on a mixed group of 21 participants, both faculty members and master students.

The participants filled out a questionnaire with overall six different scenarios for which they needed to choose one or more valid design decisions for optimizing a given scenario. The scenario itself was roughly described in text, whereas the performance results were present in the visualizations only. 14 of the participants used the original Palladio visualization ($P\ old$), the remaining 7 participants used the new visualization ($C\ new$). All participants needed to answer the same questionnaire, but were provided with the different visualizations. Participants using the new visualization $C\ new$ were provided with result visualization as shown in Figure 5 and described in Section 6.1. Participants using the original Palladio visualization $P\ old$ were provided pie charts for resource utilization, mean values and cumulated density functions for response time, and/or behavior specifications of component internals for resource demands. Each visualization diagram provided a short legend.

The questionnaires provided multiple design options for each scenario. There were correct and required design options (that would resolve the performance problems), wrong design options (that would actually worsen the performance) and optional design options (that might slightly improve performance or save cost). For each participant and each scenario, we evaluated whether all correct and required design options and no wrong design options were checked. In this case, the decision of the participant for this scenario was considered correct. The optional design options had no influence on this assessment. Thus, although we provided a multiple choice questionnaire, sheer guessing would have led to very few correct decisions.

We asked the participants to self-assess their experience in software performance. Based on the self-assessment, we can distinguish our results for participants with “low” or “high” experience.

For the comparison (Q4), we performed statistical hypothesis testing on Q1 and Q2. Our hypothesis were $H1_1$: “Participants using $C\ new$ make in average more correct design decisions over the evaluated scenarios than participants using $P\ old$” and $H2_1$: “Participants using $P\ old$ need longer for the questionnaire in average than participants using $C\ new$”. We decide whether to reject the opposite null hypotheses $H1_0$ and $H2_0$ based on Welch’s t-test [27] and a significance level $\alpha = 0.05$.

In the second phase, we studied how well participants totally untrained in both software development and software performance can handle the new visualization for further insight into Q3. We asked another 20 undergraduate students in their second year to interpret the new visualization and also fill out the questionnaire described above.

We do not claim to have conducted a controlled experiment, in which we have controlled all influencing factors (such as common knowledge of the participants). Rather, our study allows an initial evaluation of the improved, new visualization $C\ new$.

6.1 Scenarios: Performance Design Decisions

To illustrate the possible visualizations, we created six different performance scenarios in which changes in the software architecture design are required. All scenarios are minimalistic and basic. More complex scenarios would be a combination of the visualization of basic scenarios.

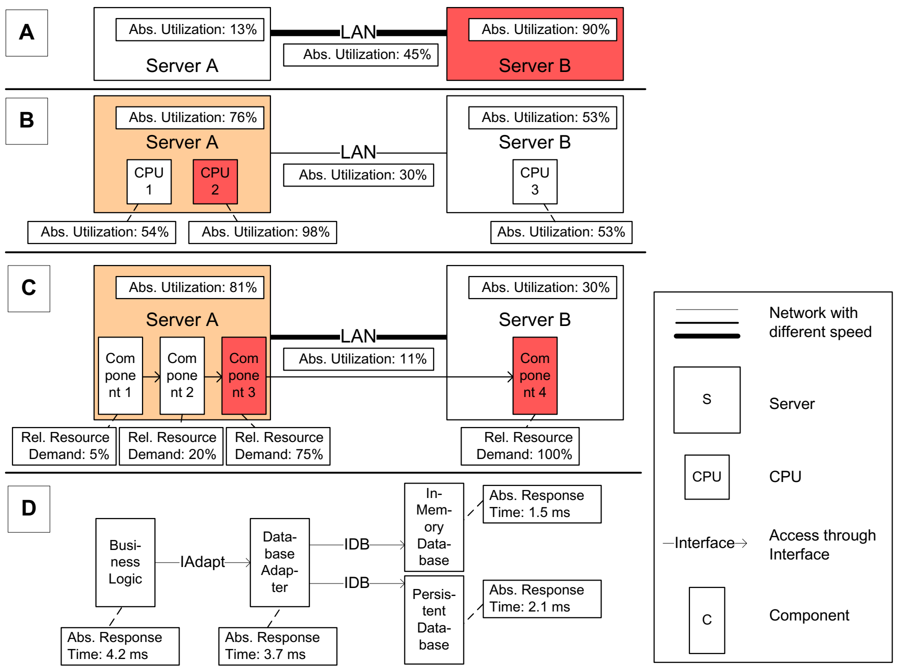

Fig. 5. New visualizations

1) Server bottleneck. In the first scenario (cf. Fig. 5 A), two servers are connected by a network connection. One of the servers is indicating a very high utilization (>=90%), while the other has a normal utilization. In this case and at this level of information, there are two basic alternatives for increasing the performance of the system: a) use stronger server hardware for the highly utilized server, or b) consider changing component allocations (see scenario 4). For this scenario it is crucial to allow fast bottleneck recognition.

2) Network bottleneck. This scenario is comparable to the previous one (same topology; similar to Fig. 5 A), beside the fact that another resource – the network connection – is in an overload situation. Again, here two design alternatives are imaginable: a) increase the bandwidth of the network connection, or b) change the component allocations such that network traffic is reduced. Scenario 4 presents possible component relocations.

3) Replicated resources. If a single server has multiple CPUs or CPU cores, but just one or few of them are utilized, a lack of parallelism is indicated (cf. Fig. 5 B). In this case, a) other versions of utilized components with increased optimizations for parallel execution can be used, or b) low utilized server nodes can be down-sized to avoid wasting computational power, or c) further components can be allocated to the same server (and possibly bound to single CPUs) to improve the utilization of unused CPUs / cores. If the performance requirements are not fulfilled, also faster CPUs / cores can be used to avoid a bottleneck.

4) Component's resource demand. If server nodes are in an overload situation (cf. Fig. 5 C), it is desirable to also examine the component's allocation to server nodes for gaining insight on the causes of load. Therefore, each component's resource demand must be known. If the resource demand is normalized per server node, one can easily identify large demands which are likely to cause the overload: a) If at the same time other server nodes have only little load, it is a possible alternative to move the component with high resource demand to the other server nodes, b) if both, a server node and also a connected network connection have a high load, one needs to rethink the complete allocation and should also check whether for example the network connection could be improved to support solution a), c) for the server node with high utilization, also faster CPUs can be used.
5) Compatible components. In scenarios where two or more components offer the same interface, they become exchangeable (cf. Fig. 5 D). This is the case for databases with standardized interfaces, for example. By comparing the response times of exchangeable components, the faster implementation should be easily selectable. Such component selection decisions in general cannot be met, as component response time is sensible for the usage profile. If for example, a database is optimized for read requests, but fed with mostly writes, another database optimized for that purpose is much faster. Thus the decision cannot be met in advance, instead performance results (here: response time) must be visualized for a specific load situation.
6) Multiple usage profiles. As indicated in the previous case, components are subjected to different usage profiles. Sometimes, also for a deployed component, the usage profile still changes: Peak load situations, batch runs, or increasing users over time need to be handled. To estimate the scalability and sensitivity of a software architecture for changes in the usage profile, it is preferable to directly compare their impact on an architecture level. If the architecture does not satisfy the requirements for different anticipated load situations, the above scenarios 1-5 can be applied. Visualizations can apply multiple data layers which can be interactively hidden / unhidden. In the experiment, the participants compared two different usage profiles, each of which was visualized similarly to figure 5 C.

6.2 Phase 1 Results

All participants completed the entire questionnaire with the above scenarios in an average of 26.43 minutes (new visualization $C\ new$) or 40.17 minutes (original Palladio visualization $P\ old$).

Table 1 compares the results for question 1 (correctness) and question 2 (duration) for the different levels of experience (question 3) and for the two visualizations (question 4). Our first observation is that participants using the new visualization $C\ new$ were able to make better design decisions (92.8% correct vs. 72.7% correct for $P\ old$, p-value $= 0.007 < \alpha$, thus we reject $H1_0$). Both low and highly experienced participants performed similarly for their visualisation. It is remarkable that high experienced participants using $P\ old$ performed worse than low experienced. On the questionnaires of some of these participants, we found manual calculations, which possibly lead to wrong results. In column "total # of decisions", we show the total number of decisions

Table 1. Results for cartography (*C new*) and the original Palladio visualization (*P old*)

Experience	% of correct decisions		total number of decisions		duration in min		standard deviation of duration	
	C new	*P old*	*C new*	*P old*	*C new*	*P old*	*C new*	*P old*
low	91.7%	75.0%	24	36	28.25	37.00	15.44	4.24
high	94.5%	70.8%	18	48	24.00	41.75	13.08	12.45
total	92.8%	72.7%	42	84	26.43	40.17	13.46	10.13

made by all participants in the corresponding group: Each of the 8 participants evaluated up to 6 scenarios.

For the duration, we again notice that participants using *C new* were able to finish the decision process (including interpretation and the decision itself) more quickly (p-value $= 0.03 < \alpha$, thus we reject $H2_0$). Here, for low experienced participants using *C new*, the standard deviation of the duration is three times higher than for *P old*. Interestingly, more experienced participants needed longer than low experienced participants for interpreting *P old*. Possibly, they tried to delve deeper in the information available for *P old*, for example behavior specification (which did not lead to better decisions).

Overall, *C new* performed significantly better with respect to both correctness and duration. We can accept both hypotheses $H1_1$ and $H2_1$ stated above. The experience of the participants (with at least basic knowledge in software engineering) does not seem to have a strong impact on the results.

Table 2. Percentage of correct decisions for each scenario

	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5	Scenario 6
C new	100.0%	100.0%	100.0%	85.7%	85.7%	85.7%
P old	92.9%	92.9%	78.6%	35.7%	64.3%	71.4%
Average	95.2%	95.2%	85.7%	52.4%	71.4%	76.2%

Table 2 shows the percentage of correct decisions for each scenario and the two visualizations. For the simpler scenarios 1 and 2, most subjects decided for the right design options. For the more complex scenarios 3, 5, and 6, the ratio of correct decisions is slightly lower for *C new*, whereas for *P old*, the ratio dropped about one quarter. For scenario 4, the differences between *C new* and *P old* are most pronounced, as the *P old* visualization seems to have been hard to interpret for most subjects: In *P old*, resource environment and software architecture are split into two distinct views, which might have been problematic to link. Here, most *P old* subjects could not correctly assess the consequences of their proposed re-allocation decision.

6.3 Phase 2 Results

In phase 2, the 20 participants untrained on both software development and software performance were able to make correct design decisions in 68.3% of a total number of 99 completed scenarios (some participants only answered 3 or 4 scenarios). The duration was on average approx. 40 minutes. Thus, the new visualization $C\ new$ seems to have helped the participants, even though totally inexperienced, in gaining some insight in the performance problems of the described scenarios. However, the lower percentage of correct decisions shows that the visualization is probably not sufficient to allow these totally inexperienced participants to make correct design decisions in general and to reliably solve performance problems.

7 Conclusions

In this paper, we have presented an integrated approach for the visualization of software performance at an architectural level. With visualization means from software cartography, our approach depicts software performance prediction results in an software architecture visualization. As our quasi experiment involving 41 participants shows, the approach enables users with at least a basic level of experience in software development to correctly derive design decisions from visualizations of performance prediction. Compared to a reference group applying the original visualization of the Palladio approach, participants performed more correctly and faster, independent of in-depth experience with software performance prediction. Thus, our approach enables users to map performance prediction results back to a software architecture model and lets them decide for correct design alternatives.

Our approach meets demands of software cartography visualizations (cf. Section 4) as it a) relates architecture and performance prediction results, b) highlights critical architectural elements, c) eases correlation analysis through multiple layers, and d) enables decomposition by multiple viewpoints at different levels of detail. Nevertheless, our approach is not useful for completely untrained users, which have no experience with software development and software performance engineering. Currently, our visualizations are limited to four different views, although including support for an arbitrary number of data layers.

For our future work, we plan to further push the automation of the integration of the Palladio performance prediction approach with the SoCaTool software cartography. Besides adding further views to the approach, we also would like to integrate other quality attributes like reliability and maintainability. As our approach is not limited to Palladio, it would be beneficial to use the proposed visualizations with other SPE approaches.

References

1. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Trans. on SE 30(5), 295–310 (2004)
2. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82, 3–22 (2009)
3. Buckl, S., Ernst, A.M., Lankes, J., Matthes, F., Schweda, C., Wittenburg, A.: Generating visualizations of enterprise architectures using model transformation (extended version). Enterprise Modelling and Information Systems Architectures – An International Journal 2(2) (2007)

4. Compuware. Applied performance management survey (October 2006), `http://www.cnetdirectintl.com/direct/compuware/Ovum_APM/APM_Survey_Report.pdf` (last retrieved 2009-02-10)
5. de Miguel, M., Lambolais, T., Hannouz, M., Betgé-Brezetz, S., Piekarec, S.: UML extensions for the specification and evaluation of latency constraints in architectural models. In: Workshop on Software and Performance, pp. 83–88 (2000)
6. Eclipse Foundation. Graphical modeling framework homepage
7. Ernst, A.M., Lankes, J., Schweda, C.M., Wittenburg, A.: Using model transformation for generating visualizations from repository contents. Technical report, Technische Universität München, Munich (2006)
8. Franks, G., Hubbard, A., Majumdar, S., Neilson, J., Petriu, D., Rolia, J., Woodside, M.: A toolset for performance engineering and software design of client-server systems. Perform. Eval. 24(1-2), 117–136 (1995)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addision-Wesley, Reading (1995)
10. Hake, G., Grünreich, D., Meng, L.: Kartographie. Walter de Gruyter, Berlin (2002)
11. Heath, M.T., Etheridge, J.A.: Visualizing the performance of parallel programs. IEEE Software 8(5), 29–39 (1991)
12. HyPerformix Inc. Hyperformix homepage (2007), `http://www.hyperformix.com` (last retrieved 2009-01-22)
13. IEEE. IEEE Std 1471-2000 for Recommended Practice for Architectural Description of Software-Intensive Systems (2000)
14. Kähkipuro, P.: UML-based performance modeling framework for component-based distributed systems. In: Dumke, R.R., Rautenstrauch, C., Schmietendorf, A., Scholz, A. (eds.) WOSP 2000 and GWPESD 2000. LNCS, vol. 2047, pp. 167–184. Springer, Heidelberg (2001)
15. Kounev, S.: Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. IEEE Trans. on SE 32(7), 486–502 (2006)
16. Lehr, T., Segall, Z., Vrsalovic, D.F., Caplan, E., Chung, A.L., Fineman, C.E.: Visualizing performance debugging. Computer 22(10), 38–51 (1989)
17. Marzolla, M.: Simulation-Based Performance Modeling of UML Software Architectures. PhD Thesis TD-2004-1, Università Ca' Foscari di Venezia, Mestre, Italy (February 2004)
18. Matthes, F.: Softwarekartographie. Informatik Spektrum 31(6) (2008)
19. Nassar, M.: VUML: a viewpoint oriented UML extension. In: 18th IEEE International Conference on Automated Software Engineering, pp. 373–376 (October 2003)
20. Object Management Group (OMG). UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP (realtime/05-02-06) (2006)
21. Object Management Group (OMG). Unified Modeling Language: Superstructure Specification: Version 2.1.2, Revised Final Adopted Specification (formal/2007-11-02) (2007)
22. OMG. Meta Object Facility (MOF) Core Specification, version 2.0 (formal/06-01-01) (2006)
23. Petriu, D.C., Shen, H.: Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 159–177. Springer, Heidelberg (2002)
24. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2), 131–164 (2009)
25. Sarukkai, S.R., Kimelman, D., Rudolph, L.: A methodology for visualizing performance of loosely synchronous programs. In: Scalable High Performance Computing Conference, 1992. SHPCC 1992. Proceedings, pp. 424–432. IEEE, Los Alamitos (1992)
26. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, Reading (2002)

27. Welch, B.L.: The generalization of student's problem when several different population variances are involved. Biometrika 34, 28–35 (1947)
28. Williams, L.G., Smith, C.U.: Making the Business Case for Software Performance Engineering. In: Proceedings of the 29th International Computer Measurement Group Conference, Dallas, Texas, USA, December 7-12, 2003, pp. 349–358. Computer Measurement Group (2003)
29. Wittenburg, A.: Softwarekartographie: Modelle und Methoden zur systematischen Visualisierung von Anwendungslandschaften. PhD thesis, Technische Universität München (2007)
30. Woodside, C.M., Hrischuk, C.E., Selic, B., Bayarov, S.: Automated performance modeling of software generated by a design environment. Perform. Eval. 45(2-3), 107–123 (2001)
31. Woodside, M., Franks, G., Petriu, D.C.: The Future of Software Performance Engineering. In: Proceedings of ICSE 2007, Future of SE, pp. 171–187. IEEE Computer Society Press, Washington (2007)
32. Yan, J., Sarukkai, S., Mehra, P.: Performance measurement, visualization and modeling of parallel and distributed programs using the aims toolkit. Softw. Pract. Exper. 25(4), 429–461 (1995)
33. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward Scalable Performance Visualization with Jumpshot. Int. J. High Perf. Comp. Appl. 13(3), 277–288 (1999)