Facilitating performance predictions using software components

Jens Happe, Heiko Koziolek, Ralf Reussner

1. Introduction

Composing software systems of independent building blocks is the central vision of component-based software engineering (CBSE) since the sixties. It became a core research topic of software engineering during the nineties [**1**]. CBSE advocated the reuse of software building blocks without requiring understanding of their internals, which led to the term "software component". In this black-box reusability, components deviated from objects and classes.

Since 2000, CBSE widened its perspective from a purely functional view on components to an extended extra-functional view on quality attributes, which allows for example performance [**2**] or reliability [**3**] evaluation of component-based software architectures (CBSA).

Performance prediction of CBSA follows other engineering disciplines that reason about a system's quality based on the properties of its components. In civil and electrical engineering, the early assessment of quality attributes ensures reliable and cost-efficient development and production. In software development, performance is often mainly considered at the end of the development cycle when the complete system can be tested. This so-called 'fix-it-later' approach can become expensive if the late detection of quality problems requires architectural changes and thus re-implementation which can cause considerable time-to-market delays.

Performance modeling can assist in detecting potential performance problems during early development stages and identifying alternative solutions. One benefit of component-based performance prediction is the ability to divide the performance specification work among software architects and component developers, thus leading to reduced complexity. Furthermore, a potential reuse of component performance specifications enables cost-efficient predictions.

Component-based systems are specifically suited for early design-time quality predictions. The impact of implementation decisions influencing performance properties is reduced, because implemented and well-tested components reduce potential performance problems and may already be accompanied with performance specifications.

Several CBSE performance prediction methods have emerged during the last decade [**2**], e.g., PECT [**4**], KLAPER [**5**], CBML [**6**], PROCOM [**7**], as well as our own approach called Palladio [**8**]. In addition to these core contributions, many supplemental approaches have been developed that contribute to the performance analysis of component-based systems. These approaches concern the estimation of resource demands as machine-specific values [**9**] or as machine-independent measures [**10**] by using test-beds for components as well as the inclusion of middleware properties in prediction models [**11**].

In this article, we use Palladio to demonstrate how componentization can be exploited for software performance analysis. While the concrete realization is specific to our approach, the principles

introduced here are generally valid. The remainder of this article will illustrate typical performance questions, information required for performance models, and prediction results for a concrete setup.

2. Running example: MediaStore

Our running example "MediaStore" is a component-based system with intentionally reduced complexity for illustration purposes. Figure 1 visualizes the static architecture of the system with UML diagrams. The figure shows a combined component and deployment diagram. The system allows users to download and upload various media files (e.g., audio/video files) to a central database. The expected usage profile is 40 user requests per minute, where each requests consists of 8-12 files with a size of 1-8 MB each, which can be modeled as a probability distribution.

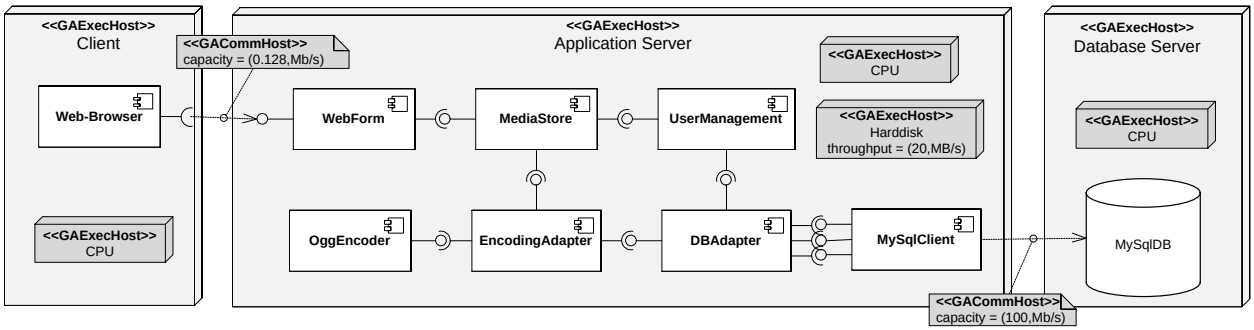


Figure 1: MediaStore system: components, servers, and network links

The software architect has assembled the system from a number of software components, which are potentially provided by third-parties. For example, the "Webform" component handles user interactions, while the "MediaStore" component contains the business logic. Upon an upload request the store re-encodes the files provided by the user using the external OggEncoder component, which writes the files to disk and encodes them to a standard bitrate.

The initial design in Figure 1 still has many degrees of freedom (e.g., size of the servers, allocation of the components to servers, alternative components The software architect is interested in analyzing the initial design from a performance viewpoint as detailed in the next section.

3. Typical performance questions and performance influencing factors

Software architects often ask the following questions to understand the performance interrelationships within their design prior to deployment:

1) How does the *architecture* affect response times and throughputs for the expected workload?
2) How does the implementation of specific *components* influence performance?
3) How does the *allocation of* components to resources influence performance?
4) How does the system perform if its *workload* increases unexpectedly?

These questions capture four main influences on software performance detailed in the following. For each question we give a concrete question to be answered for the MediaStore example, which will be evaluated later.

Architecture

The architecture's design specifies the connection of components and for white-box components also their high-level internal behavior. Both can influence the software performance. Software architects can apply different architectural patterns to optimize performance, such as caches or replication in combination with load balancing. For the MediaStore example, software architects can ask: *How does a database caching component affect the maximum throughput of the MediaStore?*

Component Selection

In some cases, multiple components with similar functionality are available for reuse. For example, there are many different audio codecs or databases accessible through generic interfaces (e.g., WAV-file input, ODBC). Software architects can either buy these components off-the-shelf (COTS) or decide for an in-house development. Software architects need to consider the effect of the component implementation (e.g., algorithm efficiency) on system quality. For the MediaStore, an interesting question is: *To what extent does a 20% faster (w.r.t. average response time) encoder improve the end-to-end response time?*

Resource Allocation and Sizing

Once components have been selected and an architecture has been designed, components can be allocated to hardware nodes and resources (CPU, HDD, and network) can be sized. Multiple criteria can affect the allocation of components. Allocating individual components to dedicated nodes may be beneficial for performance and reliability but also increases costs. While distribution can provide more processing power for individual components, remote communication can bear expensive performance overheads that might not be justified. Furthermore, the used middleware usually includes a number of configuration options, such as thread pool sizes, message encryption or data caching all of which can affect performance. For the MediaStore, one alternative allocation would be: *How does a two-tier deployment (Application and DB on a single but faster server) affect CPU utilisation and the maximum throughput?*

Usage profile

A system's usage profile is characterized by its *load* (How many users are working concurrently on the system?) and its user *work* (Which actions does a user execute? What amount of data is involved?). Architecture design and sizing target a specific workload specified in extra-functional requirements (e.g., support for 100 requests / minute). For the MediaStore, software architects can evaluate: *Can the system handle the load if the user arrival rate increases from 40 users per minute to 60 users per minute? How does the distribution of file sizes influence end-to-end response time?*

To construct a performance model, software architects and component developers need to provide information about the system under study as discussed in the following section.

4. Information required for modeling

A key postulate of CBSE is the separation of *component developers* implementing components according to specifications and *software architects* assembling components to build an application. This separation of concerns (implementing and assembling components) also implies that performance-relevant information about components and the architecture is divided between these roles. We further consider the role of *system deployers* who define the system's execution environment and allocate components to nodes and *domain experts* who are familiar with user behavior. All roles need to provide information to build a predictive performance model.

To support these roles, we have designed and implemented the Palladio Component Model (PCM). It is defined by a metamodel segmented into individual parts for each developer role. In the following, we describe the information required from each role and how this information is captured in the PCM (Figure 2).

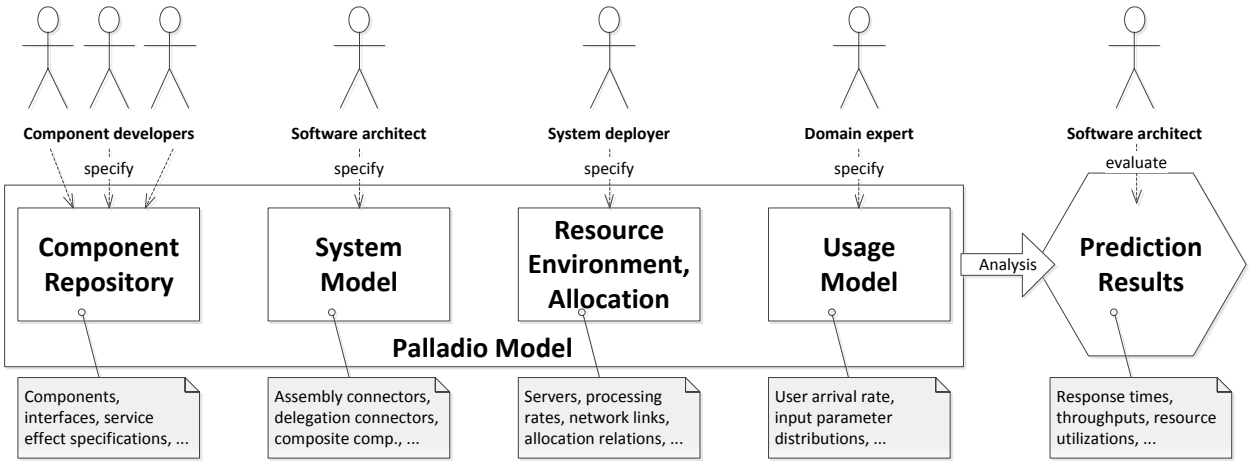


Figure 2: Information required for performance modeling from different roles in CBSE

Information from component developers

The *component developers'* choice of algorithms, design patterns, and tuning of the implementation determine the component's performance properties to a large extent. Ideally, this knowledge is held only by component developers as components are supposed to be black-boxes. To enable performance analyses, component developers need to specify (i.e., publish) how their components use hardware resources (e.g., execution times for algorithms) as well as other components (via calls to required interfaces). Depending on the development stage of the component, developers can roughly estimate or measure the necessary values as discussed in Section 5.

The PCM provides a component repository, where component developers can publish components and interfaces. For performance analysis, component developers can add abstract component behavior models for each component service to the component repository. The PCM includes a special modeling language for component behaviors called "Resource Demanding Service Effect Specification" (RDSEFF).

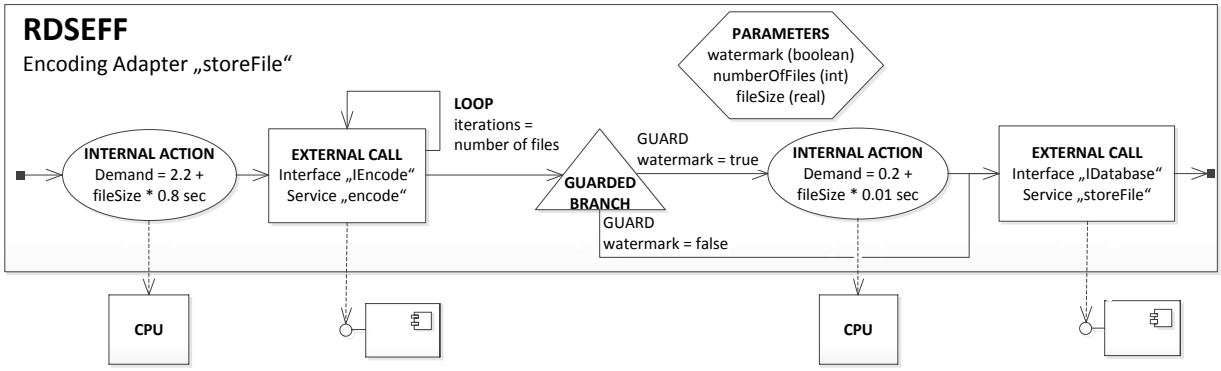


Figure 3: Example service effect specification

Figure 3 provides an example of an RDSEFF for the service “storeFile” of the “EncodingAdapter” component from Figure 1. It consists of a flow of internal actions (i.e., computations on hardware resources) and external calls (i.e., invocation of required services). Internal actions may subsume a large amount of code in a compact form and include a parameterizable demand specification (e.g., a linear CPU demand depending on the input file sizes). External calls are bound to required interfaces, not to concrete components, because the component developer is unaware of the concrete connected components. In our example, the two external calls will be directed to the OggEncoder and DBAdapter components. An RDSEFF flow may contain branches, loops, and forks, which may in turn depend on certain parameters (Figure 3) provided by other developer roles.

Information from software architects

At the architectural level, the choice of components and their connections influence performance. *Software architects* need to specify the selection and assembly of components, but should be unaware of component internals. Therefore, the PCM provides a modeling language (System Model) specifically for software architects. It is an architecture description language that allows horizontal composition (i.e., connecting component to other components) and vertical composition (i.e., nesting components inside composite components). Figure 1 shows the System Model of the MediaStore as a component diagram. Internals of the components are not visible in the System Model. Software architects can evaluate different design alternatives by, for example, exchanging components or adding components that realize performance patterns such as caches or load balancers to their architecture.

Information from system deployers

The execution environment and deployment of component-based systems is managed by *system deployers*. For software performance prediction, system deployers specify the execution environment (i.e., servers and middleware) and allocate components to nodes. The PCM captures performance-relevant information about the execution environment within a dedicated modeling language, which includes the speed of processors and additional devices, network latencies, middleware configurations, operating system parameters, virtualization, etc. Furthermore, system deployers can allocate components to resource containers. Figure 1 illustrates the allocation of components used in the MediaStore.

Information from domain experts

Finally, user behavior and workload have a large influence on system performance. *Domain experts* can provide information about different classes of users and their behavior as well as the input data they provide. Thus, the PCM has decoupled the usage specification from the architecture and environment model, so that it can be specified independently. For the MediaStore example, the arrival rate of customers (requests per minute) and their behavior (e.g., browse, download, and upload) determine the load of the system. This includes usage parameters, such as the number of files uploaded into the store (e.g., always 2 to 8 files) or their file sizes (e.g., between 5 and 10 MB).

Obtaining the necessary data for the models requires further considerations, especially for the component developers, as detailed in the next section

5. Data collection

Collecting data to construct parameterized performance models as in Figure 3 is a challenging task. The problem has to be approached differently depending on the availability of an implementation.

Estimation

If no implementation is available because the component is still in a design stage or the third-party provides no information, the software architect has to estimate resource demands. The software performance engineering methodology [**12**] provides guidelines on how to estimate the performance of software at design time. While it is hard to provide accurate time consumptions for individual computations steps, developers can often specify upper and lower bounds of a computation with confidence.

Estimations can be based on the amount of data involved in a computation or the complexity of the involved algorithms. For the MediaStore example, developers expect certain file sizes (e.g., MP3 files between 5 and 10 MB). They also have an initial understanding of the algorithmic complexity of an audio encoding or watermarking algorithm and can provide an estimate of the time consumption based on small experience with existing encoders (e.g., between 40 and 80 seconds to encode a 5 MB file or between 2 and 4 seconds for watermarking).

The upper and lower bound of such an estimation can be successively refined (i.e, narrowed down) during the development once implementation artifacts or prototypes become available.

Measurement

If an implementation of the component is available, component developers can inspect the source code, set up a test-bed, and run multiple experiments to get the necessary data for the performance model. For white-box components, the developer can use profiling tools (e.g., gprof, Intel VTune, dotTrace, JProfiler, JVMPI) or manually instrument the source code. For black-box components, monitoring tools (e.g., Eclipse TPTP) assist the developer. In case of the MediaStore components, we instrumented their Java code and used the `System.nanoTime()` method from the Java API to get precise timing values.

Component developers have to account for unknown usage profiles, unknown required service implementations, and unknown hardware environments as explained in the following.

To account for unknown *usage profiles*, the component developer can configure load drivers to supply input parameters. Existing unit tests can often be modified and used as load drivers. Another option is to inspect the source code to determine parameter dependencies. For the MediaStore, the encoding algorithm was measured for different file sizes and a linear function relating file sizes to execution time was derived.

To account *for unknown hardware environments*, the component developer either needs to execute the measurements on reference hardware or has to use an abstract resource demand measure, such as the number and types of bytecode instructions in Java code, the number of transactions, or the SAP Application Performance Standard (SAPS). For the MediaStore example, we determined the resource demands in a reference hardware environment and specified them in milliseconds.

Special considerations are necessary for *middleware* properties. Application containers may have a significant impact on the overall system performance, e.g., by managing thread pools for optimal concurrency or marshaling for remote communication [**13**]. Software architects can capture their influence using standard benchmarks (e.g., SPECjEnterprise2010, SPECjms2007, .NET stock trader) to determine their processing overhead.

. Such performance data can be woven into a performance model by replacing the abstract connectors between components with concrete marshaling actions and network latencies [**11**]. In our running example, the connectors between the “WebBrowser” and “WebForm” as well as between the “MySQLClient” and the “MySQLDB” could be refined reflecting the marshaling overheads for remote communication.

Tool support: PALLADIO Modeling Workbench

The Palladio Modeling Workbench is an Eclipse-based tool providing graphical editors to construct component repositories, system models, resource models, and usage models. It seamlessly integrates various analysis tools (e.g., a discrete-event simulator, the layered queuing network solver) and provides software architects immediate feedback on performance bottlenecks. The PCM-Bench is an open source project available at www.palladio-simulator.com.

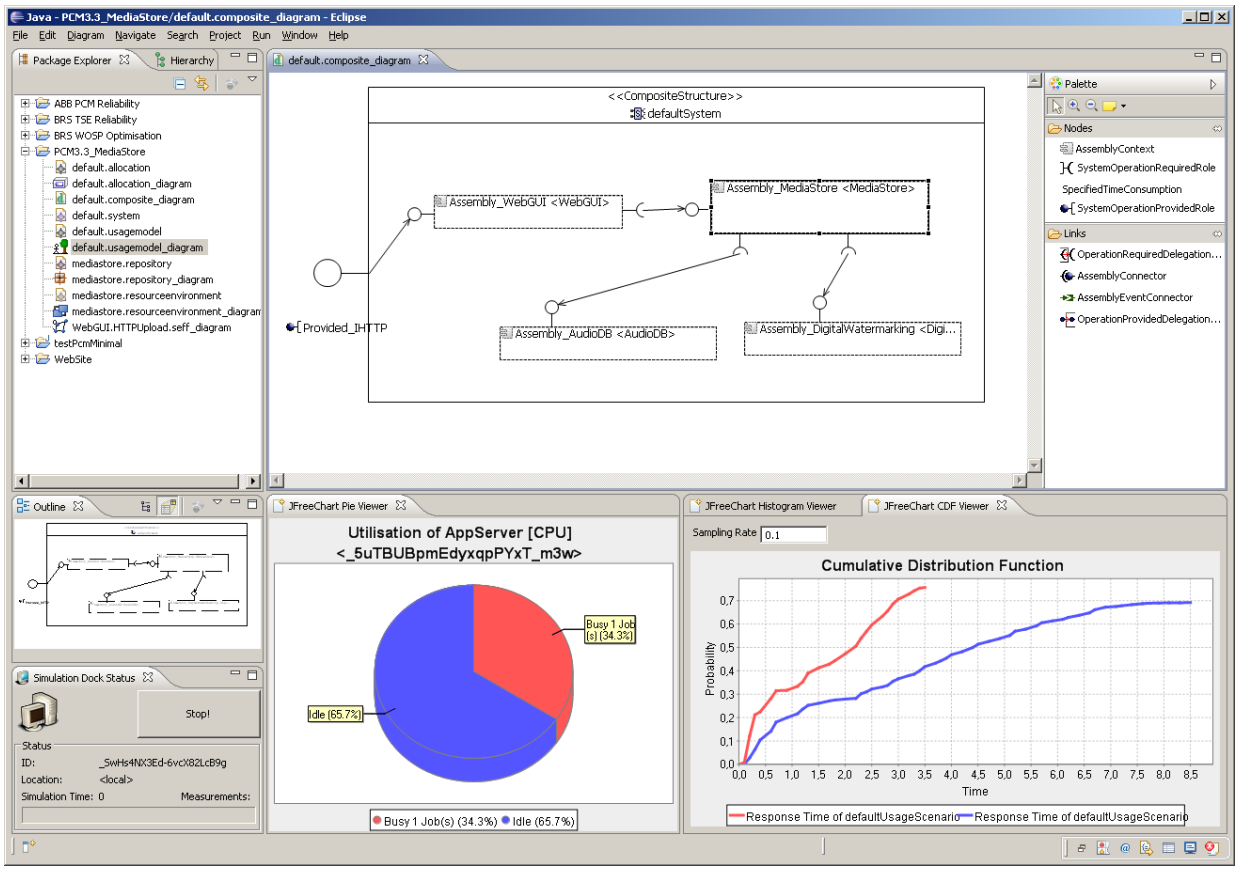
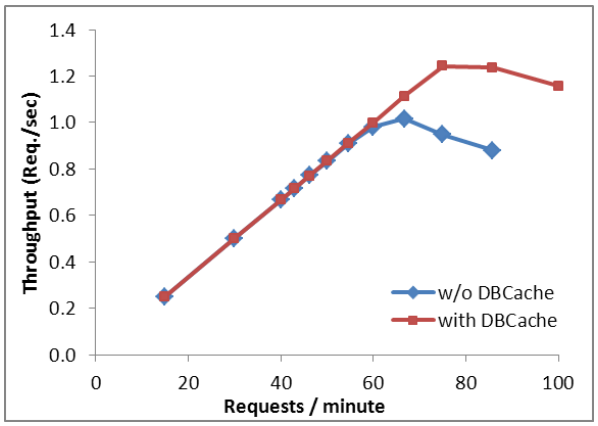


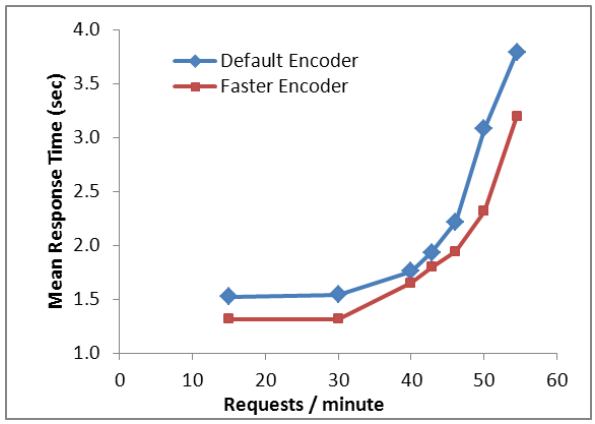
Figure 4: Screenshot Palladio Modeling Workbench showing the system model editor and prediction result visualizations

6. Predictions

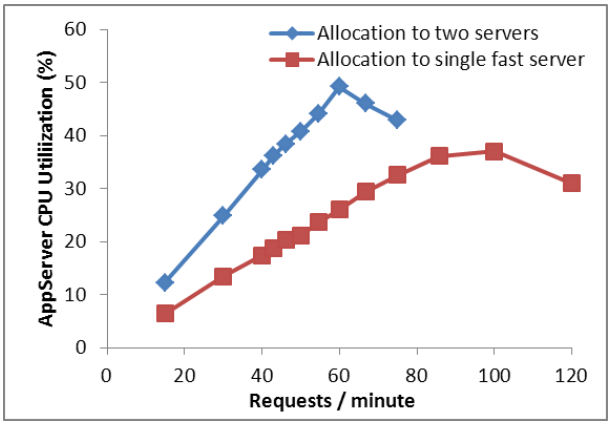
Once the models of all roles have been collected, a complete performance model (e.g., a Palladio model) can be constructed and analyzed automatically. Many recent approaches rely on mathematical formalisms, such as (layered) queueing networks, stochastic Petri nets, and stochastic process algebra, or simulation [**2**]. Ideally, tools encapsulate these formalisms and hide their details from software architects.



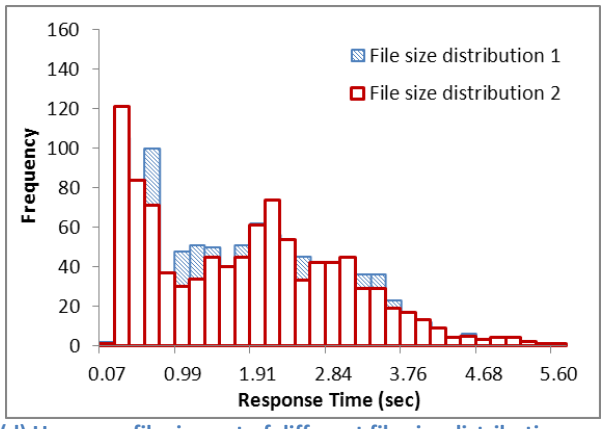
(a) Architecture: Impact of adding a DBCache component on the maximum throughput



(b) Component Selection: Impact of using a faster encoder on end-to-end response time



(c) Resource Allocation & Sizing: Impact of deploying all components to a faster server



(d) Usage profile: impact of different file size distributions on response time (two histograms)

Figure 5: Performance prediction results for the MediaStore.

Figure 5 to 8 illustrate typical performance metrics provided by the Palladio Modelling Workbench based on predictions for the MediaStore example. We initially calibrated the MediaStore performance model with measurement data. We showed in a former study [**8**] that the deviation between the measured and predicted performance is less than 10 percent if accurate input data is provided. The predictions answer the performance questions raised earlier:

- Architecture (Figure 5(a)): altering the architecture by adding a database caching component with a cache hit rate of 20% leads to an approx. 22% higher maximum throughput of the system.
- Component selection (Figure 5(b)): exchanging the default encoder with a faster (but also more expensive) encoder leads to slightly reduced mean end-to-end response times. The main

bottleneck in the system is the hard disk of the database server, thus this faster CPU computation has only limited impact.

- Resource environment (Figure 5(c)): using a faster application server leads to a reduced load on its CPU even though the database component was additionally allocated to this server. The system would be able to handle approx. 90 instead of 60 requests per minute. However, the CPU gets not saturated (i.e., only exhibits a maximum utilization of 40 percent) as the bottleneck of the system is the hard disk.
- Usage profile (Figure 5(d)): the response times of the MediaStore depend on the size of files requested by the customers. Assuming a given file size distribution, thus also the user response times follow this distributions. In Figure 5(d), two predictions were performed for two different file size distributions. The resulting histograms show a certain deviation.

The effort required for modeling depends on the required accuracy and granularity. Several companies have used the Palladio Modelling Workbench (e.g., IBM [**14**], ABB [**15**], and Ericsson under guidance in joint projects. The studies ranged from performance evaluations of I/O concepts of virtualization layers to process control systems. The effort to analyze typical scenarios was in the range of 3-6 person months. It has been demonstrated that early design-time performance analysis can produce accurate prediction results (e.g., less than 20 percent prediction error in [**16**]), which enables the comparison of different design alternatives. The PCM provides means to include influence of middleware platforms into the prediction. However, even if exact predictions are not possible, the PCM provides good results with respect to the ranking of design alternatives.

7. Conclusions

The component paradigm has proved useful not only for the functional design of software systems, but also for the evaluation of software performance and other quality attributes such as reliability and maintainability. We have shown in this article how component performance can be modeled, how the required data can be collected, and what kinds of analyses are possible.

CBSE posed two new major challenges for software performance engineering. First, the development process is distributed among multiple parties (at minimum component developer and component consumer). For performance analysis, this distribution has to be reflected in the design of prediction models. Second, performance specifications of components have to be parameterized with respect to their usage and deployment.

While early reasoning on the extra-functional properties of a software system based on its components cannot rule out all quality problems, it has however proved to be an important step towards a true engineering approach to software design.

8. Bibliography

[1] Clemens Szyperski, Dominik Gruntz, and Stephan Murer, *Component Software: Beyond Object-Oriented Programming*.: Addison-Wesley, 2002.

[2] Heiko Koziolek, "Performance Evaluation of Component-based Software Systems: A Survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634-658, August 2010.

[3] A. Immonen and E. Niemelä, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," *Journal on Software and System Modeling*, vol. 7, no. 1, pp. 49-65, 2008.

[4] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau, "Packaging Predictable Assembly," in *Component Deployment*, 2002, pp. 108-124.

[5] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta, "Filling the gap between design and performance/reliability modelsof component-based systems: A model-driven approach," *JSS Vol. 80(4)*, pp. 528-558, 2007.

[6] Xiuping Wu and Murray Woodside, "Performance Modeling from Software Components," *SIGSOFT Software Engineering Notes Vol. 29(1)*, pp. 290-301, 2004.

[7] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković, "A Component Model for Control-Intensive Distributed Embedded Systems," in *Component-Based Software Engineering*, 2008, pp. 310-317.

[8] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *JSS Vol. 82(1)*, vol. 82, no. 1, pp. 3-22, January 2009.

[9] M. Woodside, V. Vetland, M. Courtois, and S. Bayarov, "Resource Function Capture for Performance Aspects of Software Components and Sub-Systems," in *Performance Engineering: State of the Art and Current Trends*, 2001, pp. 239-256.

[10] Michael Kuperberg, Klaus Krogmann, and Ralf Reussner, "Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models," in *Component Based Software Engineering*, 2008, pp. 48-63.

[11] Jens Happe, Steffen Becker, Christoph Rathfelder, Holger Friedrich, and Ralf H. Reussner, "Parametric performance completions for model-driven performance prediction," *Performance Evaluation*, vol. 67, no. 8, pp. 694-716, August 2010.

[12] Connie Smith and Lloyd Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*.: Addison-Wesley, 2003.

[13] Y. Liu, A. Fekete, and I. Gorton, "Design-level performance prediction of component-based applications," *IEEE Transactions on Software Engineering*, vol. 31, no. 11, pp. 928-941, 2005.

[14] N. Huber, S. Becker, C. Rathfelder, J. Schweflinghaus, and R. Reussner, "Performance Modeling in Industry:A Case Study on Storage Virtualization," in *ICSE'2010 - Industrial Track*, 2010, pp. 1-10.

[15] Heiko Koziolek et al., "An Industrial Case Study on Quality Impact Prediction for Evolving Service-Oriented Software," in *ICSE'2011 - Industrial Track*, 2011, p. to appear.

[16] Samuel Kounev, "Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 486-502, July 2006.

[17] M.D. McIlroy, "Mass Produced Software Components," in *Software Engineering, Report on a conference sponsored by the NATO Science Committee*, 1968, pp. 138--150.

[18] Ralf H. Reussner, Heinz W. Schmidt, and Iman H. Poernomo, "Reliability prediction for component-based software architectures," *JSS Vol. 66(3)*, vol. 66, no. 3, pp. 241-252, June 2003.