# The Performance Cockpit Approach: A Framework for Systematic Performance Evaluations

Dennis Westermann*, Jens Happe*, Michael Hauck†, and Christian Heupel‡
*SAP Research, Vincenz-Priessnitz-Strasse 1, 76131 Karlsruhe, Germany
†FZI Forschungszentrum Informatik, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
‡Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany
Email: dennis.westermann@sap.com, jens.happe@sap.com, hauck@fzi.de, heupel@kit.edu

***Abstract*—Evaluating the performance (timing behavior, throughput, and resource utilization) of a software system becomes more and more challenging as today's enterprise applications are built on a large basis of existing software (e.g. middleware, legacy applications, and third party services). As the performance of a system is affected by multiple factors on each layer of the system, performance analysts require detailed knowledge about the system under test and have to deal with a huge number of tools for benchmarking, monitoring, and analyzing. In practice, performance analysts try to handle the complexity by focusing on certain aspects, tools, or technologies. However, these isolated solutions are inefficient due to the small reuse and knowledge sharing. The Performance Cockpit presented in this paper is a framework that encapsulates knowledge about performance engineering, the system under test, and analyses in a single application by providing a flexible, plug-in based architecture. We demonstrate the value of the framework by means of two different case studies.**

## I. Introduction

Software performance (timing behaviour, throughput, and resource utilization) is one of the key quality attributes of a software system that is directly visible to the user. Therefore, the evaluation of performance metrics has to be considered during the whole life cycle of the system. Furthermore, large enterprise application systems are rarely developed from scratch but usually built on a large basis of existing software (middleware, legacy applications, and third party services). In many cases, the sheer size and complexity of a software system hinders the application of performance engineering techniques. Moreover, performance engineering requires knowledge about the system under test, how to measure the right metrics, and how to analyze and interpret the results. Bringing together these different expertises is another problem when analyzing large and heterogeneous software systems. In recent years, many approaches have been published in the context of software performance engineering [9]. Model-based approaches require detailed knowledge about the structure and the internals of the system under test, which is in today's enterprise applications often not available (due to the complexity of the systems). Thus, in practice mainly measurement-based approaches are used to analyze the performance of a software system [15].

As the performance of a system is affected by multiple factors on each layer of the system, performance analysts have to deal with a huge number of tools for instrumentation and monitoring. Measurement data is distributed among various locations and interfaces which makes it difficult to run holistic analyses. Furthermore, the knowledge about the tools is distributed among various stakeholders along the system stack (e.g. system administrator, middleware expert, application developer, etc.).

Existing solutions often focus on certain parts of the system [12], technologies [10], or problem areas [20] to overcome the complexity. However, this leads to the problem that performance analyses are conducted by different departments using their isolated applications which hinders reuse and knowledge transfer. Moreover, it increases the effort of analyzing a system when components, such as the middleware, are changed and have to be integrated. Another problem is the complexity of existing benchmark applications. Deploying and running a benchmark is time consuming and requires detailed knowledge about system administration and measurement techniques. Moreover, benchmark configuration is often done via property files and scripts which is inconvenient and hard to validate.

The contribution of this paper is an extendable framework for measurement-based performance evaluation called Performance Cockpit. The framework allows the derivation of performance models based on automated measurements and the execution of regressions benchmarks together with the system's build process [8]. The cockpit captures knowledge about performance measurement and related tools in a single application by using a plug-in based architecture. Stakeholders implement only those parts of the system they are an expert in, while the basis functionality necessary to control benchmark systems is provided by the framework. The configuration is based on a generic meta-model in order to keep the framework platform-independent.

The presented framework aims for (i) less effort for setting up performance tests, (ii) better knowledge transfer, (iii) flexible measurement environment, (iv) better usability, and thus enabling performance analyses to a broader audience. These goals are addressed by a plug-in based approach where each plug-in is developed by a corresponding domain

expert. The plug-ins can be reused in multiple scenarios. If a component in the system under test is changed, one can easily switch the plug-ins without adapting the actual measurement application. Additionally, the provisioning of a single-point of configuration via a configuration meta-model increases the usability and eases the set up of the experiments.

The generality of our approach allows its application in various scenarios. We present two different case studies where we used our prototypical implementation of the framework. In the first case study, we automatically inferred a performance model for Message-oriented Middleware [6]. The other case study uses the framework to derive the properties of multi-core schedulers.

The paper is structured as follows. Section II describes the idea and the goals of the Performance Cockpit approach. Section III discusses related work. In Section IV, we illustrate the architecture of the proposed framework. Section V shows two case studies that use the current prototype of the Performance Cockpit. Finally, Section VI concludes the paper.

## II. The Idea Behind The Performance Cockpit

With the increasing complexity of software systems, performance engineering becomes more and more challenging. The goal of the Performance Cockpit presented in this paper is to support the performance analyst by conducting different performance engineering tasks. This includes sizing and adaptation questions, regression benchmarking during development, dependency analyses between system components (e.g. for root-cause-analyses), and the creation of performance models in order to predict the performance of the system under various load scenarios.

Today's software systems consist of multiple layers and components. Generally, workload, metrics, tools and analysis techniques used for one problem can not (or only partly) be used for another problem. The expertise on how to manage the various layers and components is spread over various stakeholders. Nevertheless, Jain outlines steps that are common to all performance evaluation projects [7]. The Performance Cockpit is meant to encapsulate the best practices of performance evaluation by integrating them into the framework while abstracting from the concrete system under test. The Performance Cockpit is specialized for the actual system under test via a configuration model and appropriate plug-ins. Thus, the approach helps to avoid common mistakes in performance evaluation and allows for flexible and efficient creation of performance measurement applications.

In the following, we describe how to implement systematic performance evaluations [7] using our Performance Cockpit approach. Figure 1 gives a schematic overview of the involved roles, the necessary assets, and the interactions.

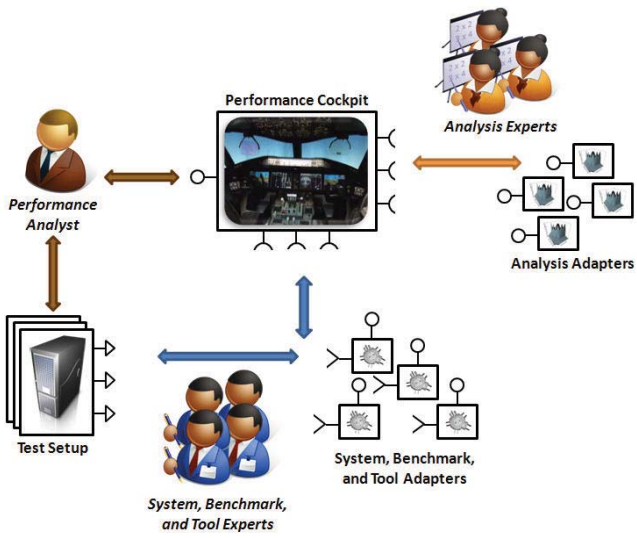


Fig. 1. The Performance Cockpit Approach

1) **Set up the Test System:** This is a necessary preliminary step that has to be done by the *Performance Analyst*. It includes the specification of the goals of the study, the definition of system boundaries, and the listing of services to be evaluated. The information about the setup of the *Test System* is modeled using the configuration meta-model of the framework. Additionally, the *Performance Analyst* selects the adapters which connect the various components of the software systems (e.g. operating system, middleware, and application monitoring tools) with the *Performance Cockpit*. The adapters are plug-ins developed by the corresponding *System, Benchmark, and Tool Experts*. Thus, the *Performance Analyst* does not have to deal with technical details. Moreover, the *Performance Analyst* can leverage from previous configurations of similar set ups.
2) **Select Metrics and List Performance-Relevant Parameters:** In this step, the *Performance Analyst* models the metrics of interest for the intended evaluation as well as all the parameters (system and workload) that affect these metrics. He specifies the metrics and parameters based on the configuration meta-model provided by the framework. Again, the *Performance Analyst* can use the information gathered in previous evaluations.
3) **Model Experiments:** Once the performance-relevant parameters have been identified, the *Performance Analyst* has to specify which of them should be varied during the evaluation and which should not. Furthermore, he has to determine the possible values of the parameters to be varied. After that, he has to decide on a sequence of experiments that provide enough information to get correct and statistically significant results. This information can also be modeled within the *Performance Cockpit*.
4) **Select Workload:** In the next step, the *Performance Analyst* selects a set of adapters that generate load on the

system under test according to the experiment configurations modeled in the previous step. The plug-in based approach of the framework allows to reuse adapters between different performance evaluations which reduces the effort for the *Performance Analyst*. This is especially useful when evaluating service-oriented systems which are characterized by their heterogeneous environment and require knowledge about multiple tools and technologies.

5) **Analyze and Interpret Data:** In this step, the data gathered during the measurements is analyzed using different statistical analysis techniques. The *Performance Cockpit* stores the measured data delivered by the *System, Benchmark, and Tool Adapters* in a central repository. Then, the *Performance Analyst* can run different statistical analyses on the measured data. The analyses are executed using different techniques implemented by *Analysis Experts* and integrated into the framework via *Analysis Adapters*.
6) **Present Results:** The final step is the presentation of the resulting functions. Currently, this is supported by the plotting functionality of the adapted statistic tools.

## III. Related Work

This section introduces current research dealing with measurement-based performance analysis of software systems.

Different frameworks have been proposed to support the automated adaptation and calibration of model-based performance models. Woodside et al. [19] present a methodology for capturing resource functions of software components and systems. They describe the process of building and analyzing performance models. A workbench executes the experiments and stores the resulting functions in a repository. An approach to calibrate existing performance models is presented in [20]. The approach iteratively refines an initial LQN model with respect to parameter estimates. The estimates are calculated by an estimator component based on the results of multiple system tests. However, these approaches are tied to a certain type of performance model or a certain aspect of a software system.

A more general performance evaluation framework is introduced by Meyerhoefer and Neumann in [10]. The TestEJB framework allows to gain performance relevant data from J2EE applications. Mos and Murphy [11] introduce the COMPASS framework for the identification of performance issues in component based systems. COMPASS is based on three modules. A monitoring module captures data by inserting proxy components. The gathered data is then used by a modeling module that builds various UML models. These models are further enhanced by a performance prediction module that allows to simulate and analyze the models. Based on this approach, Parsons and Murphy [12] built a framework for the detection of performance anti-patterns in component based systems. In addition to the COMPASS framework, they use byte code analysis as monitoring technique. However, these approaches assume that a component-based architecture design exists for the system under test which has to be known and extended by the performance analyst in order to insert a proxy-layer or injector component. Moreover, they are not applicable in heterogeneous and dynamic systems due to their limited flexibility and extendability.

Thakkar et al. [17] describe a framework that targets the same problem as our approach. The framework runs a series of tests and uses the obtained results to create software performance models. The proposed framework is extendable by application specific scripts that allow to control the test system. In order to reduce the required number of actually needed test runs the authors suggest to use domain knowledge or statistical analyses techniques such as Main Screen Analysis [21] and two-way ANOVA [16]. The authors also estimate the effort necessary to customize the framework to for other applications. However, the authors remain open how to design such a framework and how their existing solution can be actually customized to other applications. Moreover, the approach does not consider the use of a single-point of configuration and thus does not provide the detailed encapsulation of knowledge as it is proposed in this paper.

The PLASTIC framework for testing service-oriented architectures aims at enabling online and offline testing of networked applications, encompassing functional and non-functional properties [2]. The framework integrates a set of tools for generating and executing test scenarios as well as for monitoring different metrics in service-oriented applications. However, the framework does not aim at the automated integration of different tools and thus could leverage from the approach suggested in this paper.

## IV. Architecture

In this section, we present the architecture of the Performance Cockpit which implements the ideas described earlier (see Section II). The architecture is based on two major design principles: separation of concern and abstraction. This allows us offering a single application that implements best practices in software performance evaluation but is highly extendable. The framework hides the complexity of benchmarks, system administration, tool configuration, etc. from the performance analyst providing a single point of configuration. Figure 2 depicts the overall structure of the Performance Cockpit framework. The framework architecture consists of five main components:

- ***Experiment Series Controller:*** The *Experiment Series Controller* is the heart of the Performance Cockpit. The component is responsible for the management of experiment runs and thus, it is the link between user configuration, benchmark execution and data analysis. An experiment series is a set of experiment runs necessary to derive the required results.

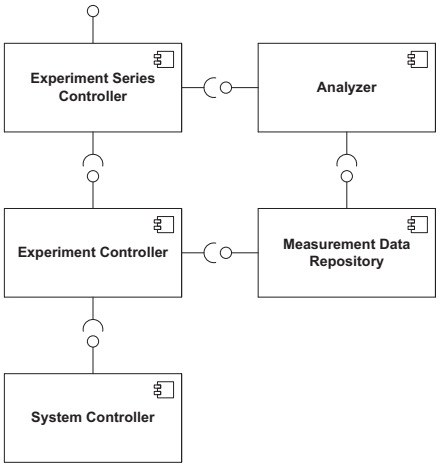


Fig. 2. Component Overview

- ***Experiment Controller:*** This component is responsible for the communication with the various *System Controller* components of the test system. It executes single experiment runs including configuration, starting, stopping, and cleaning up of components in the test system. Moreover, the *Experiment Controller* gathers the measured data after an experiment run.
- ***System Controller:*** The *System Controller* is responsible for the management of a certain system in the overall test setup. For each node in the test setup at least one *System Controller* instance exists. The component handles the communication with the *Experiment Controller* and consist of multiple system specific plug-ins that include adapters to the different components of the managed system.
- ***Measurement Data Repository:*** The Measurement Data Repository is the central data store of the Performance Cockpit responsible for the persistence of the measured data. It provides models and interfaces to store and query measurement data and analysis results.
- ***Analyzer:*** This component is responsible for analyzing the measured data and delivering the results configured by the performance analyst. It provides various statistical analysis methods. The selection of an appropriate method depends on the measurement goals and is based on heuristics and best practices. The automated intelligent selection of an appropriate analysis method is part of our current research and will be implemented in a future release of the framework.

The fundamental communication between the different components is depicted in Figure 3.

After the *Experiment Series Controller* has initially loaded the configuration modeled by the performance analyst, it starts a series of experiment runs based on the configuration. Thereby, the configuration describes which parameters should be varied, in which range, and in which step size (e.g. the number of concurrent users should vary from 1 to 1000 with a step width of 100). For each experiment run, the *Experiment Controller* executes the following steps:

- First, it calls the *initSystem* method on each *System Controller* instance in the test system. The concrete action performed by this method depends on the node-specific implementation of the *System Controller* instance. Possible actions are setting of tool configurations, system clean up from previous runs, generating test data for the current experiment run, etc.
- In the second step, the *Experiment Controller* starts the actual benchmark on the test system. Therefore, it again calls the corresponding method on each node-specific implementation of the *System Controller* instance.
- When the benchmark has finished, the *Experiment Controller* gathers the monitored data from each node and saves the result in the *Measurement Data Repository*

.

After all experiment runs are completed, the *Experiment Series Controller* starts analyzing the measured data based on the configured measurement goal. The analysis is executed by the corresponding plug-in within the *Analyzer* component.

Figure 4 illustrates the example deployment diagram used in the case study described in Section V-A. The basic Performance Cockpit components are deployed on one node while for each node in the test system one instance of the *System Controller* component is deployed.

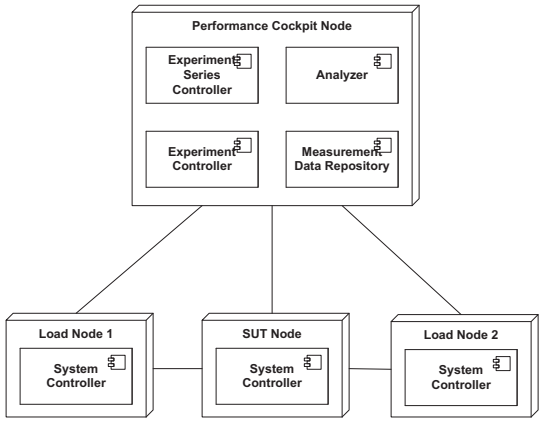


Fig. 4. Sample Deployment

In the following, we take a deeper look in the structure of the *System Controller* component. As stated above, the *System Controller* component contains the various adapters to the system management, monitoring and benchmark applications. The plug-in based architecture of the component is illustrated in Figure 5.

The creation of *System Controller* instances is encapsulated using the patterns builder [5] and dependency injection [3]. The combination of these patterns allows to construct different representations of the *System Controller* at runtime. The class *Abstract System Controller* encapsulates those parts that

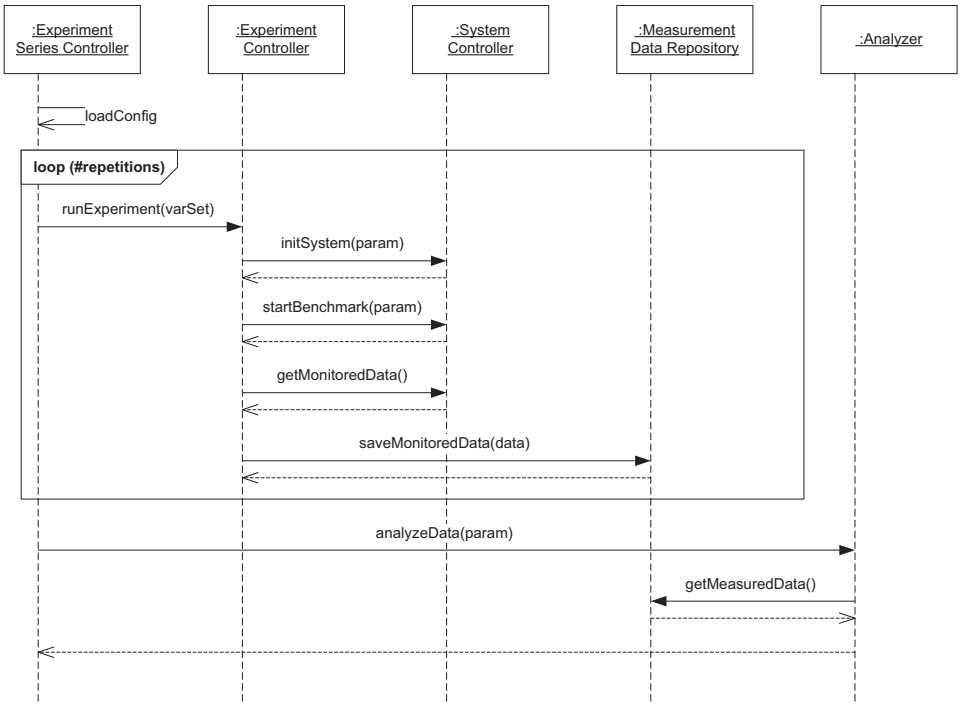


Fig. 3. Component Interaction

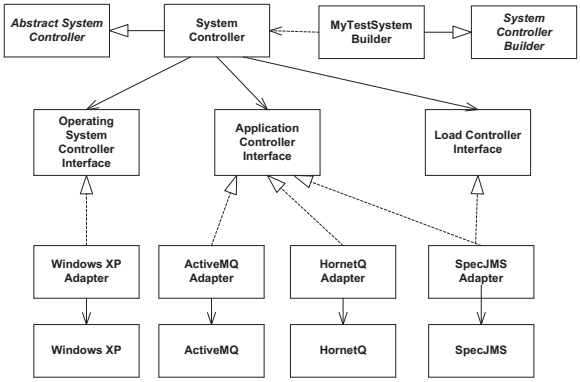


Fig. 5. The System Controller Component

are common to all *System Controller* instances such as the information how to connect to the *Experiment Controller*. The concrete *System Controller* holds three different interfaces:

- **Operating System Controller Interface:** Contains operations necessary to manage the operating system (e.g. reboot, kill processes, query monitoring information). The interface is implemented by different adapters for the operating systems used in the test environment. So far, we implemented adapters for Debian Linux, Ubuntu Linux, Windows Server 2003 and Windows XP.
- **Application Controller Interface:** This interface covers the management of all applications above the operating system layer. This includes middleware, monitoring tools, benchmark applications, and applications under test. The interface contains operations to start, stop, initialize, clean up, and monitor the applications. In our current research, we evaluate the performance of message-oriented middleware and therefore wrote adapters for e.g. ActiveMQ, HornetQ, and the SPECjms Benchmark.
- **Load Controller Interface:** The *Load Controller Interface* is used to configure and manage the tools that generate load on the system under test. This could be the load generating part of a benchmark (such as SpecJMS) or a standalone load generator tool (such as LoadRunner or JMeter).

## V. Case Study

In the following, we present two case studies that we conducted using a first prototype of the framework. In the first case study, we automatically inferred a performance model for Message-oriented Middleware [6]. The second case study uses the framework to derive the properties of multi-core schedulers.

### A. Performance Evaluation of Message-oriented Middleware Platforms

In our first case study, we captured the performance of Message-oriented Middleware (MOM) using systematic measurements and statistical inference. Therefore, we measured the influence of different parameters (e.g., message size and arrival rate) on the performance (timing behavior, throughput, and resource utilization) of the MOM.

The strategy of sampling the effect of different parameter combinations on performance is a critical point as it has to

be detailed enough to achieve accurate predictions but must also be kept feasible at the same time (i.e., measurements must not last too long). For this purpose, we separate the measurements into three phases (see Figure 6).

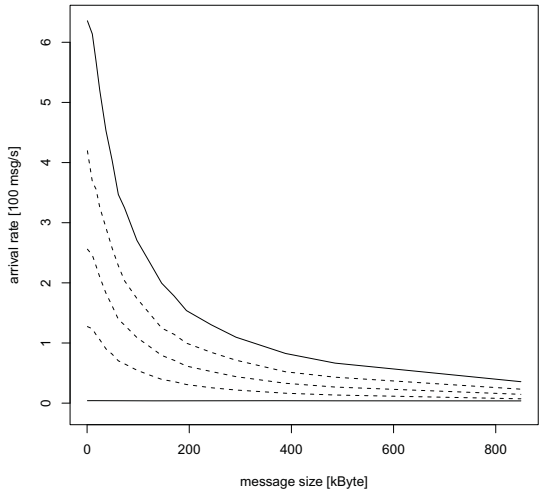


Fig. 6. Measurement Method

In the first phase, we determine the maximal throughput of the system for each message size (upper solid line). After that, we measure the influence of individual parameters without resource contention in phase two, e.g. the effect of different message sizes on the delivery time when only one message is processed at a time (solid line at the bottom). The final phase samples the performance of the system under study between the best case and the worst case. Therefore, we separate the arrival rate into $N$ equidistant steps (dashed lines). To estimate the dependency between different system characteristics (configuration and usage) and performance metrics of interest we use Multivariate Adaptive Regression Splines (MARS) and Genetic Optimization (GO). MARS [4] is a statistical method that requires no prior assumption as to the form of the data. GO is used to fit a function against measured data when the functional dependency between multiple parameters is already known. In our case study, the basis of the GO is an error function that describes the difference between the observations and the measurements's predictions. The goal of the GO is to minimize this error function.

The test system of this case study comprises three nodes as depicted in Figure 4. The actual system under test is the ActiveMQ Messaging Middleware which was deployed on an IBM x3850 Server with Debian Linux. To conduct our measurements we used the SPECjms benchmark [14]. Besides this server system the test setup also includes two load generator nodes, both Windows Server 2003 Systems. The Performance Cockpit runs on a separate node and uses the statistic tool R [13] to execute the analyze the measured data. The benchmark was executed 288 times and required 6 minutes for each run, leading to a total measurement time of approximately 31 hours. For each run, we measured the utilization of CPUs and hard disk, network traffic and throughput as well as the delivery time of messages. Figure 7 and Figure 8 illustrate the results of the statistical analyses.

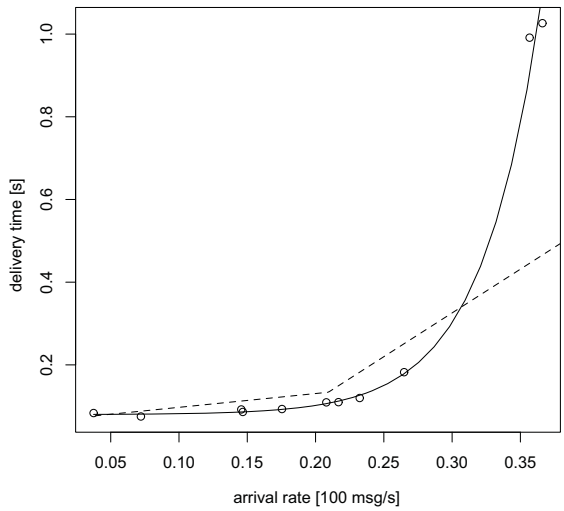


Fig. 7. Comparison of MARS and Genetic Optimization

Figure 7 shows the averages of the measured delivery time (circles) compared to predictions of MARS (dashed line) and genetic optimization (solid line). In this case, regression splines do not capture the steep increase in delivery times while the genetic optimization provides much better results.

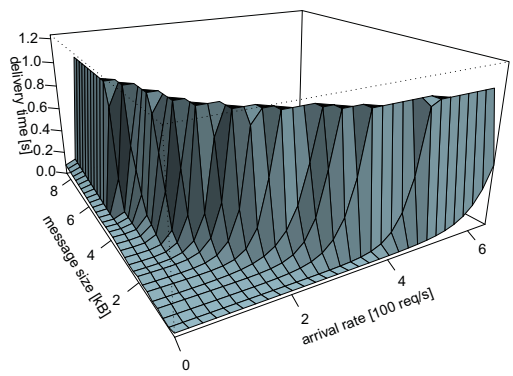


Fig. 8. Functional Dependency of Delivery Times on Arrival Rate and Message Size

Figure 8 shows the result of the genetic optimization for different message sizes and different arrival rates. The exponential function is determined separately for each

message size. The resulting terms are combined by linear interpolation. The detailed results of the case study are presented in [6].

The case study showed that a generic framework for performance measurements is very valuable for our research. To conduct our measurements, we developed adapter for Windows 2003 Server, Debian Linux (e.g. the sar tool), SPECjms (e.g. configure, start, and stop), ActiveMQ (e.g. configure, start, and monitor), and the statistic tool R (e.g. MARS analysis). The plug-in based architecture allows us to easily repeat the study, e.g. for other message-oriented middleware such as HornetQ, while reusing the infrastructure and most of the adapters. The study also showed that it is very helpful to have a single-point of configuration which captures best practices and can be evaluated before the measurements are actually started.

### *B. Automated Analysis of Operating System and Virtualization Properties*

In a second case study, we use the Performance Cockpit to perform automated measurements of operating system and virtualization layer performance properties in order to create detailed performance models.

The performance of a software heavily depends on the infrastructure the software is running on. This includes operating system scheduling, but also additional infrastructure properties, such as virtualization. Obtaining an accurate performance model of the infrastructure is a cumbersome task if it is done manually. It involves getting familiar with the specification and/or the implementation as well as performing a lot of measurements. Furthermore, the performance model is probably not valid for future releases. To use a performance model for different versions, measuring and studying the infrastructure has to be repeated. To simplify the development of an infrastructure performance model, we defined a feature model of performance-relevant properties. For these properties, we define measurements that can be performed automatically using the Performance Cockpit. Thereby, the measurement results are analyzed to derive a feature model instance which is then transformed into a model for performance analysis.

In the following, we describe how the Performance Cockpit has been applied for this case study. To perform measurements on the platform, we implemented a plug-in for the system adapter which is used to put different kinds of synthetic load (e.g. CPU-bound load or I/O load) on the platform. The load is created using a load-generation library described in [1]. For this purpose, the cockpit is used to specify a detailed description of how the load should look like. This includes the amount of load, i. e. how much CPU-bound load or I/O-bound load should be produced, and the chronology of load, i. e. if certain load should occur in sequence or in parallel (e. g. in different threads or processes). For the analysis, we developed an analysis adapter which derives certain performance properties from the experiment results. The measurements are completely automated. For instance, the experiment series controller initiates experiment runs in order to detect certain feature model properties. Based on the results that are retrieved from the analyzer, the experiment series controller iteratively completes the feature instance by running additional experiments that detect missing properties.

## VI. Conclusions and Future Work

In this paper, we presented a framework for measurement-based performance evaluation called Performance Cockpit. The idea of the Performance Cockpit is to encapsulate the knowledge necessary to run various types of performance evaluations in a single application. Thereby, different domain experts (e.g. system administration, monitoring, analysis) contribute to the resulting Performance Cockpit by providing corresponding plug-ins. We described the flexible and extendable architecture of the framework and demonstrated the application of our Performance Cockpit prototype in two different case studies.

The Performance Cockpit allows performance analysts to apply best practices in performance engineering even in large enterprise applications. Using the Performance Cockpit, the performance analyst does not need to know all details about system administration, tools, and statistical analyses as this knowledge is encapsulated in different plug-ins. These plug-ins are provided by various experts of the respective domains. Depending on the chosen configuration, the Performance Cockpit can be used to answer sizing questions, create performance predictions (e.g., to provide guarantees in SLAs), support decisions for adaptation scenarios (e.g., moving an image from one node to another), run regression benchmarks on nightly builds, and define resource demands for performance models.

Based on the general architecture presented in this paper and the already implemented prototype of the Performance Cockpit, we plan the following enhancements. For our ongoing research [18], we are currently working on a module that automatically identifies those parameters of a system that influence performance. The automated determination of experiments that offer maximum information with minimal effort is another research question which once answered will further enhance the framework. Concerning the statistical analyses, the framework so far supports multivariate linear regressions, genetic optimization, and bayesian networks. However, in the future we will include additional analysis techniques such as the principal component analysis for the automated determination of performance-relevant parameters. Another research question we are currently investigating is how we can automatically interpret the data during the measurement phase and use this information to control the following measurements. Finally, we we want to provide a further plug-in interface that allows to add various export

functions like HTML pages or instances of performance models (such as surveyed in [9]).

**Acknowledgement:** This work is partially supported by the German Federal Ministry of Education and Research under promotional reference 01|G09004 (ValueGrids) and by the European Community's Seventh Framework Programme (FP7/2001-2013) under grant agreement no.216556.

## REFERENCES

[1] S. Becker, T. Dencker, and J. Happe. Model-Driven Generation of Performance Prototypes. In *SIPEW 2008: SPEC International Performance Evaluation Workshop*, volume 5119, pages 79–98, 2008.

[2] A. Bertolino, G. Angelis, L. Frantzen, and A. Polini. The plastic framework and tools for testing service-oriented applications. pages 106–139, 2009.

[3] M. Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html, Jan. 2004.

[4] J. H. Friedman. Multivariate adaptive regression splines. *Annals of Statistics*, 19(1):1–141, 1991.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[6] J. Happe, D. Westermann, K. Sachs, and L. Kapova. Statistical inference of software performance models for parametric performance completions. In *6th International Conference on the Quality of Software Architectures, QoSA 2010, Prague, Czech Republic, , June 23-25, 2010, Proceedings. To Appear.*

[7] R. Jain. *The art of computer systems performance analysis*. Wiley Interscience, New York, 1991.

[8] T. Kalibera, L. Bulej, and P. Tuma. Generic environment for full automation of benchmarking. In S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, editors, *SOQUA/TECOS*, volume 58 of *LNI*, pages 125–132. GI, 2004.

[9] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, In Press, Corrected Proof, 2009.

[10] M. Meyerhöfer. Testejb: response time measurement and call dependency tracing for ejbs. In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 55–60, New York, NY, USA, 2007. ACM.

[11] A. Mos and J. Murphy. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 235–236, New York, NY, USA, 2002. ACM.

[12] T. Parsons. A framework for detecting performance design and deployment antipatterns in component based enterprise systems. In *DSM '05: Proceedings of the 2nd international doctoral symposium on Middleware*, pages 1–5, New York, NY, USA, 2005. ACM.

[13] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2010. ISBN 3-900051-07-0.

[14] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410–434, Aug 2009.

[15] J. Sankarasetty, K. Mobley, L. Foster, T. Hammer, and T. Calderone. Software performance in the real world: personal lessons from the performance trauma team. In V. Cortellessa, S. Uchitel, and D. Yankelevich, editors, *WOSP*, pages 201–208. ACM, 2007.

[16] M. Sopitkamol and D. A. Menascé. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *WOSP*, pages 53–64. ACM, 2005.

[17] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora. A framework for measurement based performance modeling. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 55–66, New York, NY, USA, 2008. ACM.

[18] D. Westermann and J. Happe. Towards performance prediction of large enterprise applications: An approach based on systematic measurements. In R. H. Reussner, C. Szyperski, W. Weck, and B. Buhnova, editors, *Proceedings of the 15th International Workshop on Component Oriented Programming (WCOP 2010)*, Interne Berichte, Karlsruhe, Germany, 2010. Karlsruhe Institute of Technology (KIT). To appear.

[19] C. 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*, pages 239–256, London, UK, 2001. Springer-Verlag.

[20] X. Wu and M. Woodside. A calibration framework for capturing and calibrating software performance models. In *EPEW '08: Proceedings of the 5th European Performance Engineering Workshop on Computer Performance Engineering*, pages 32–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] C. Yilmaz, A. S. Krishna, A. M. Memon, A. A. Porter, D. C. Schmidt, A. S. Gokhale, and B. Natarajan. Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 293–302. ACM, 2005.