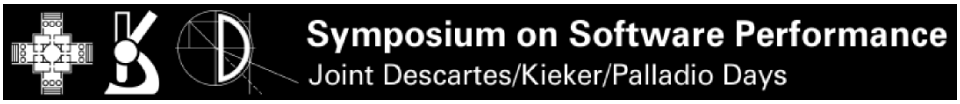


SOSP '14

Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days 2014

Stuttgart, Germany, November 26–28, 2014

Proceedings

Editors:
Steffen Becker, Wilhelm Hasselbring, André van Hoorn,
Samuel Kounev, and Ralf Reussner

Published as:
University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Technical Report Computer Science No. 2014/05

Editors' addresses:

Steffen Becker
Chemnitz University of Technology
Faculty of Computer Science
Straße der Nationen 62
09111 Chemnitz, Germany

Wilhelm Hasselbring
Kiel University
Department of Computer Science
Christian-Albrechts-Platz 4
24118 Kiel, Germany

André van Hoorn
University of Stuttgart
Institute of Software Technology
Universitätsstraße 38
70569 Stuttgart, Germany

Samuel Kounev
University of Würzburg
Department of Computer Science
Am Hubland
97074 Würzburg, Germany

Ralf Reussner
Karlsruhe Institute of Technology (KIT)
Institute for Program Structures and Data Organization
Am Fasanengarten 5
76131 Karlsruhe, Germany

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

Preface

Performance is one of the most relevant quality attributes of any IT system. While good performance leads to high user satisfaction, poor performance lead to loss of users, perceivable unavailability of the system, or unnecessarily high costs of network or compute resources. Therefore, various techniques to evaluate, control, and improve the performance of IT systems have been developed, ranging from online monitoring and benchmarking to modeling and prediction. Experience shows, that for system design or later optimization, such techniques need to be applied in smart combination.

Therefore, the Symposium on Software Performance brings together researchers and practitioners interested in all facets of software performance, ranging from modeling and prediction to monitoring and runtime management. The symposium is organized by three already established research groups, namely Descartes, Kieker, and Palladio, who use this symposium also as a joint developer and community meeting. Descartes' focus are techniques and tools for engineering self-aware computing systems designed for maximum dependability and efficiency. Kieker is a well-established tool and approach for monitoring software performance of complex, large, and distributed IT systems. Palladio is a likewise-established tool and approach for modeling software architectures of IT systems and for simulating their performance.

The two-and-a-half day program features developer meetings, 19 talks (including two invited industrial talks), seven tutorials, and a dedicated poster session with almost ten posters. In the first industrial talk, Heiko Koziolek (ABB Corporate Research) reports about six years of Performance Modeling at ABB Corporate Research. In the second industrial talk, Stefan Fütterling and Michael Großmann (Capgemini) report about performance challenges in a large mainframe system.

In addition to invited contributions from practitioners and researchers, we welcomed contributions from academic, scientific, or industrial contexts in the field of software performance, including but not limited to approaches employing Descartes, Kieker, and/or Palladio. We solicited the following types of contributions: presentation, tool demonstration/tutorial, poster. Submitted proposals were evaluated based on a submission form, asking for the proposed contribution's list of authors, title, type, summary, relation to Descartes/Kieker/Palladio, as well as a list of previous events/publications where the work has bes been presented before. Authors of accepted contributions have had the opportunity to submit a paper to be published in this symposium proceedings. This proceedings volume includes the abstracts of all accepted contributions as well as 13 papers, describing a subset of the contributions in more detail.

We would like to thank all participants that contribute to the event, including the authors and presenters, as well as the NovaTec GmbH who sponsors this event by hosting and catering.

November 2014

Steffen Becker, Wilhelm Hasselbring
André van Hoorn, Samuel Kounev, Ralf Reussner

Organization Committee

Steffen Becker, University of Technology Chemnitz
Wilhelm Hasselbring, Kiel University
André van Hoorn, University of Stuttgart
Samuel Kounev, University of Würzburg
Ralf Reussner, KIT/FZI

Local Organizers

André van Hoorn, University of Stuttgart
Stefan Siegl, NovaTec GmbH

Contents

1 Abstracts of all SOSP 2014 Contributions

1.1 The Descartes Modeling Language: Status Quo
(*Samuel Kounev, Fabian Brosig, and Nikolaus Huber*)

This presentation will present a summary of the latest developments on the Descartes Modeling Language and related tools. The Descartes Modeling Language (DML: http://www.descartes.tools), also referred to as Descartes Meta-Model (DMM), is a new architecture-level language for modeling performance and resource management related aspects of modern dynamic software systems and IT infrastructures. DML provides appropriate modeling abstractions to describe the resource landscape, the application architecture, the adaptation space, and the adaptation processes of a software system and its IT infrastructure. Technically, DML is comprised of several sub-languages, each of them specified using OMGs Meta-Object Facility (MOF) and referred to as meta-model in OMGs terminology. The various sub-languages can be used both in offline and online settings for application scenarios like system sizing, capacity planning and trade-off analysis as well as for self-aware resource management during operation. The talk will first introduce the various sub-languages and then present a summary of several case studies showing how DML can be applied in real-life scenarios. Finally, an overview of related tools in the Descartes Tool Chain (http://www.descartes.tools) will be presented.

1.2 Evaluating the Prediction Accuracy of Generated Performance Models in Up- and Downscaling Scenarios
(*Andreas Brunnert, Stefan Neubig, and Helmut Krcmar*)

This paper evaluates an improved performance model generation approach for Java Enterprise Edition (EE) applications. Performance models are generated for a Java EE application deployment and are used as input for a simulation engine to predict performance (i.e. response time, throughput, resource utilization) in up- and downscaling scenarios. Performance is predicted for increased and reduced numbers of CPU cores as well as for different workload scenarios. Simulation results are compared with measurements for corresponding scenarios using average values and measures of dispersion to evaluate the prediction accuracy of the models. The results show that these models predict mean response time, CPU utilization and throughput in all scenarios with a relative error of mostly below 20

1.3 Investigating the Use of Bayesian Networks in the Hora Approach for Component-based Online Failure Prediction
(*Teerat Pitakrat and André van Hoorn*)

Online failure prediction is an approach that aims to predict potential failures that can occur in the near future. There are a number of techniques that have been used, e.g., time

series forecasting, machine learning, and anomaly detection. These techniques are applied to the data that can be collected from the system and that contain information regarding the current state of the system, such as, response time, log files, and resource utilization.

The existing works which employ these prediction techniques to predict failures can be grouped into two categories. The first category approaches the task by using the technique to predict failures at specific locations in the system. For example, time series forecasting may be used to predict the response time at the system boundary. Once it is predicted that the response time tend to go beyond a certain value in the near future, a warning is then issued. On the other hand, the second category applies the prediction technique to the whole system, i.e., using the data collected from all locations and creating a model that can analyze and conclude from these data whether a failure is expected on the system level.

In our work, we take another direction by combining techniques in the first category with the architectural model of the system to predict not only the failures but also their consequences. In other words, the existing prediction techniques provide prediction results of each component while the architectural model allows the predicted failures to be extrapolated and further predicts whether they will affect other components in the system.

We have developed the prediction framework based on Kieker's pipe-and-filter architecture and employed its monitoring capability to obtain the data at runtime. The architectural model of the system is extracted from the monitoring data and used to create a Bayesian network that can represent the failure dependency between components. The prediction results obtained from each component failure predictors are forwarded to the Bayesian network to predict the failure propagation.

1.4 Predicting Energy Consumption by Extending the Palladio Component Model (*Felix Willnecker, Andreas Brunnert, and Helmut Krcmar*)

The rising energy demand in data centers and the limited battery lifetime of mobile devices introduces new challenges for the software engineering community. Addressing these challenges requires ways to measure and predict the energy consumption of software systems. Energy consumption is influenced by the resource demands of a software system, the hardware on which it is running, and its workload. Trade-off decisions between performance and energy can occur. To support these decisions, we propose an extension of the meta-model of the Palladio Component Model (PCM) that allows for energy consumption predictions. Energy consumption is defined as power demand integrated over time. The PCM meta-model is thus extended with a power consumption model element in order to predict the power demand of a software system over time. This paper covers two evaluations for this meta-model extension: one for a Java-based enterprise application (SPECjEnterprise2010) and another one for a mobile application (Runtastic). Predictions using an extended PCM meta-model for two SPECjEnterprise2010 deployments match energy consumption measurements with an error below 13 %. Energy consumption predictions for a mobile application match corresponding measurements on the Android operating system with an error of below 17.2 %.

1.5 Towards Modeling and Analysis of Power Consumption of Self-Adaptive Software Systems in Palladio
(*Christian Stier, Henning Groenda, and Anne Koziolek*)

Architecture-level evaluations of Palladio currently lack support for the analysis of the power efficiency of software systems and the effect of power management techniques on other quality attributes. This neglects that the power consumption of software systems constitutes a substantial proportion of their total cost of ownership. Currently, reasoning on the influence of design decisions on power consumption and making trade-off decisions with other Quality of Service (QoS) attributes is deferred until a system is in operation. Reasoning approaches that evaluate a system's energy efficiency have not reached an abstraction suited for architecture-level analyses. Palladio and its extension SimuLizar for self-adaptive systems lack support for specifying and reasoning on power efficiency under changing user load. In this paper, we (i) show our ideas on how power efficiency and trade-off decisions with other QoS attributes can be evaluated for static and self-adaptive systems and (ii) propose additions to the Palladio Component Model (PCM) taking into account the power provisioning infrastructure and constraints.

1.6 Integrating Workload Specification and Extraction for Model-Based and Measurement-Based Performance Evaluation: An Approach for Session-Based Software Systems
(*André van Hoorn, Christian Vögele, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar*)

Workload modeling and generation/simulation are essential to systematically and accurately evaluate performance properties of software systems for both measurement-based (e.g., load testing and benchmarking) and model-based performance evaluation (e.g., prediction). For measurement-based evaluation, established load generation tools exist that take workload specifications as input and generate corresponding requests to the system under test (SUT). Specifications of workloads are also tightly integrated in formalisms and approaches for model-based performance evaluation, including analytic and simulative techniques. The definition of workload specifications that represent the real workload as accurately as possible is one of the biggest challenges in both areas. Only few approaches for sharing workload specifications between the model-based and the measurement-based worlds exist.

In this talk, we will present our approach that aims to interchange and automate the extraction and transformation of workload specifications for measurement-based and model-based performance evaluation of session-based software systems. The approach comprises three main components: First, a domain specific language (DSL) allows the layered modeling of probabilistic workload specifications of session-based systems. Second, we automatically extract instances of this DSL from recorded session logs of production systems, e.g., employing Kieker. During this extraction process, different groups of customers with similar navigational patterns are identified using configurable clustering algorithms. Third,

instances of the DSL are transformed to workload specifications for load generation tools and model-based performance evaluation tools. We will present a transformation to the common load testing tool Apache JMeter, including the Markov4JMeter extension developed in our previous work. Moreover, we will present our work in progress on transforming instances of the DSL to workload models in the Palladio Component Model. We evaluate our approach by comparing workload-specific characteristics (e.g., session lengths and arrival rates) from the recorded and the extracted/generated workload of the industry benchmark SPECjEnterprise2010.

1.7 6 years of Performance Modeling at ABB Corporate Research (*Heiko Koziolek*)

Despite significant scientific research, systematic performance engineering techniques are still hardly used in industry, as many practitioners rely on ad-hoc performance firefighting. It is still not well understood where more sophisticated performance modeling approaches are appropriate and the maturity of the existing tools and processes can be improved. While there have been several industrial case studies on performance modeling in the last few years, more experience is needed to better understand the constraints in practice and to optimize existing tool-chains. This talk summarizes six years of performance modeling at ABB, a multi-national corporation operating mainly in the power and automation industries. In three projects, different approaches to performance modeling were taken, and experiences on the capabilities and limitations of existing tools were gathered. The talk reports on several lessons learned from these projects, for example the need for more efficient performance modeling and the integration of measurement and modeling tools.

1.8 Performance Challenges in a Mainframe System (*Stefan Fütterling and Michael Großmann*)

The global ordering system of a car manufacturer processes several hundred thousand car orders each year. It is used by thousands of dialog-users at car-dealerships, market systems, production plants and several other third party systems which are also supplied with all necessary data. The system is running on an IBM mainframe and uses several technology stacks such as CICS/Cobol, Websphere Application Server/Java, Messaging and DB2. The system is subject to a constant change and growth due to enhanced functionalities, the addition of new markets and market systems, as well as the rise in car sales and the increasing complexity of the cars themselves.

The general growth has lead to an increase in mainframe costs, where today no one can answer exactly the questions on how much CPU consumption a user (or user-class) is triggering or in which parts of the system the most CPU consumption is caused. Additionally the goal for 2016 is, to charge-back CPU cost on a user-class basis. Here, a detailed reporting is necessary, where CPU consumption is evaluated on a per user-class and per business

transaction level. The new reporting will help to improve the operation, the optimization and the analysis of the system.

The general questions to be answered are:

- Who is calling which business transaction and how much cpu-time is consumed?
- How expensive is a business transaction?
- How can the cpu-consumption of secondary transactions be evaluated in an event based system and how can it be attributed to user-classes and/or business transactions?
- Which technology (i.e. CICS/Cobol or WAS/Java) causes the growth? Where are the cost drivers?
- How large is the future cpu-consumption in the context of a forecast for the predicted growth in car sales and additional functionality due to new releases.

The presentation describes the gathering of CPU consumption data, its aggregation on basis of a user-class mapping and the creation of performance reports and performance predictions.

1.9 Using the Free Application Performance Diagnosis Tool "inspectIT" (*Stefan Siegl*)

inspectIT (http://www.inspectit.de) is a free Java performance diagnosis solution developed at NovaTec Consulting. In this live presentation I will guide you through the features of inspectIT. We will be analyzing a sample application (DVDStore), that NovaTec extended to suffer from certain performance problem patterns, which we often see at our customers.

- *<phone rings>*
- *Customer Support:* "We once again had a complaint about hang ups and slow response times when our customer wanted to buy a DVD on our store."
- *Development Guy:* "We set up tests for this situation already; we do not see any problems. Did the customer tell you what he did specifically? We are at a loss here"

...this is where we will come in and try to figure out what is actually going on in the DVDStore. Is there really a problem in the code? Is it the user not using the application correctly? We will find out.

1.10 LibReDE: A Library for Resource Demand Estimation (*Simon Spinner and Jürgen Walter*)

When creating a performance model, it is necessary to quantify the amount of resources consumed by an application serving individual requests. In distributed enterprise systems, these resource demands often cannot be observed directly, their estimation is a major challenge. Different statistical approaches to resource demand estimation based on monitoring data have been proposed in the literature, e.g., using linear regression, Kalman filtering, or optimization techniques. However, the lack of publicly available implementations of these estimation approaches hinders their adoption performance engineers. LibReDE provides a set of ready-to-use implementations of approaches to resource demand estimation. It is the first publicly available tool for this task and aims at supporting performance engineers during performance model construction. The library can be used for offline and online performance model parameterization. LibReDE helps performance engineers to select an estimation approach for a system under study by automatically selecting suitable estimation approaches based on their pre-conditions and by offering cross-validation techniques for the resulting resource demands. Based on the cross-validation results, the performance engineer is able to compare the accuracy of different estimation approaches for a system under study.

1.11 Approaching the Cloud: Using Palladio for Scalability, Elasticity, and Efficiency Analyses (*Sebastian Lehrig and Matthias Becker*)

In cloud computing, software architects develop systems for virtually unlimited resources that cloud providers account on a pay-per-use basis. Elasticity management systems provision these resource autonomously to deal with changing workload. Such changing workloads call for new objective metrics allowing architects to quantify quality properties like scalability, elasticity, and efficiency, e.g., for requirements/SLO engineering and software design analysis. However, Palladio so far did not support such novel metrics. To cope with this problem, we extended Palladios simulation approach SimuLizar by additional metrics for scalability, elasticity, and efficiency. In this tool paper, we focus on illustrating these new capabilities. For this illustration, we analyze a simple, self-adaptive system.

1.12 Static Spotter for Scalability Anti-Patterns Detection (*Jinying Yu and Goran Piskachev*)

Understanding large and old legacy systems is tedious and error-prone task. Design patterns (and anti-patterns) detection is a reverse engineering technique which allows analysis of such systems. Software architects use this technique to recover bad design decisions. Particularly, web-based applications face scalability issues. Scalability is especially impor-

tant for cloud-based systems which need to handle different environments (e.g. change of workload over time). We want to automatically detect scalability anti-patterns of existing component-based applications on the source code level. In literature, there are different ways to define the scalability anti-patterns. Moreover, there exist several general pattern detection engines, but none of them, examines the detection of scalability anti-patterns. In our work, first we define scalability anti-patterns and specify them using our domain specific language, and second we use our model-driven static spotter to detect the specified anti-patterns. There are two inputs required for our Eclipse based implementation of the static spotter: (1) the catalog of specified patterns (in our case the scalability anti-patterns) and (2) tree model representation of the source code of our application. To demonstrate our work, we use an example web-based application where we detect the specified scalability anti-patterns.

1.13 CactoSim — Optimisation-Aware Data Centre Prediction Toolkit (*Sergej Svorobej, Henning Groenda, Christian Stier, James Byrne, and Pj Byrne*)

Virtualisation is a common technique in todays data centres for running Cloud-scale applications, providing isolation between diverse systems and customers as well as comparably lightweight relocation and consolidation of virtual on physical machines. Currently topology optimisation algorithms are in use in data centres, their configuration typically being trial-and-error based instead of using sound reasoning. Simulation can be used to enable the evaluation of such optimisation algorithms towards better reasoning at decision time. This paper describes work towards the development of a simulation-based prediction prototype (CactoSim) and its integration with the live data centre topology optimisation prototype (CactoOpt). This coupling enables the evaluation of runtime optimisations at design time. This paper describes CactoSim and how it provides extended architecture models at its interfaces but internally employs Palladio. It showcases a method for using extended architecture models with the base Palladio Component Model (PCM), including the required model transformations. In this specific case, the supported infrastructure model for data centres is described, which goes beyond PCM and takes into account the complexity of the real-world data centres virtualisation infrastructure. Finally, an integration method for simulation and live data centres is described, which allows CactoSim to pull live models and utilise them directly for simulation experiments.

1.14 Benchmarking Workflow Management Systems (*Marigianna Skouradaki, Vincenzo Ferme, Cesare Pautasso, Dieter Roller, and Frank Leymann*)

The goal of the BenchFlow project is to design the first benchmark for assessing and comparing the performance of BPMN 2.0 Workflow Management Systems (WfMSs). WfMSs have become the platform to build composite service-oriented applications, whose performance depends on two factors: the performance of the workflow system itself and the

performance of the composed services (which could lie outside of the control of the workflow).

Despite the rapid evolution of benchmarks, there is only a recent appearance of them targeting to Service Oriented Architecture (SOA) middleware. In particular for WfMS there is not yet a currently accepted benchmark, despite the recent appearance of standard workflow modeling languages such as BPMN 2.0. Our main goal is to present to the community the open challenges of a complex industry-relevant benchmark.

1.15 Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability
(*Jan Waller, Florian Fittkau, and Wilhelm Hasselbring*)

Monitoring of a software system provides insights into its runtime behavior, improving system analysis and comprehension. System-level monitoring approaches focus, e.g., on network monitoring, providing information on externally visible system behavior. Application-level performance monitoring frameworks, such as Kieker or Dapper, allow to observe the internal application behavior, but introduce runtime overhead depending on the number of instrumentation probes.

We report on how we were able to significantly reduce the runtime overhead of the Kieker monitoring framework. For achieving this optimization, we employed micro-benchmarks with a structured performance engineering approach. During optimization, we kept track of the impact on maintainability of the framework. In this paper, we discuss the emerged trade-off between performance and maintainability in this context.

To the best of our knowledge, publications on monitoring frameworks provide none or only weak performance evaluations, making comparisons cumbersome. However, our micro-benchmark, presented in this paper, provides a basis for such comparisons. Our experiment code and data are available as open source software such that interested researchers may repeat or extend our experiments for comparison on other hardware platforms or with other monitoring frameworks.

1.16 The DIN/ISO Definition and a Measurement Procedure of Software Efficiency
(*Werner Dirlewanger*)

Colloquially people used to assign the attribute speed (i. e. performance) not only to hardware but also to software. But software does not have an attribute speed. It defines—for each user initiated task—a sequence of information processing steps (machine instructions) to be performed. To find a suitable term for the aspect under construction we use two implementations (Imp1 and Imp2) of the application on the same hardware and perform two performance measurements yielding the performance values P1 and P2. To find a term describing the the behaviour of the application software we compare P1 to P2. This yields a measure of how good Imp2 transforms the task submitted into a sequence of

machine instructions compared to Imp1: Imp2 is less or more efficient.

This procedure requires a fitting definition of P and a measurement method which simulates in detail the properties of the user entity and delivers detailed performance values. This is done by the ISO 14656 method. It measures the timely throughput, the mean execution times and the total throughput, each as a Vektor of m values, where m ist the total number of task types. The software efficiency values are: The timely throughput efficiency I_{TI}, the mean execution time efficiency I_{ME} and the total throughput efficiency I_{TH}, each being a vector of m values. Additionally there is the quotient of maximal timely served users I_{MAXUSR} which is a scalar.

In the talk the ISO method will be explained. All is supported by examples of real measurement projects. History: DIN 66273 defines a very detailed performance measurement method, but the software efficiency aspect was not yet part of this Standard. The use of the DIN performance measurement method for getting software efficiency values was firstly published in the author's DIN-textbook. ISO took over the idea in ISO 14756. Comprehensive description: The author's textbook on ISO 14756.

1.17 Enabling Assembly of Systems and its Implications within the Palladio Component Model
(*Misha Strittmatter*)

The scope of Palladio models has always been software systems and their inner structure and behavior. That is why the ability to assemble structures into structure of higher order stops at the system level. This makes it impossible to simulate a system in interplay with other systems without using workarounds. This contribution proposes meta-model changes, which enable the composition of systems. The implications of these changes onto the submodels of Palladio are discussed. Systems and their interfaces will then be defined within repositories. Only one composed structure diagram will be needed. Usage Models will be able to call the interfaces of one outer structure, which may be a component or system. If a model should be simulated, an allocation model for this outer structure has to be provided. As a side effect, it will then be possible to apply load directly onto a single component, which may be beneficial in some cases. Further implications onto Palladios tooling and developer role concept are also discussed. The modification will be performed in the scope of the PCM refactoring process, so it will not cause much additional development effort nor will it break functionality, as this is first mitigated by a backwards transformation.

1.18 Parallel Simulation of Queueing Petri Nets
(*Jürgen Walter, Simon Spinner, and Samuel Kounev*)

Queueing Petri Nets (QPNs) show high accuracy to model and analyze various computer systems. To apply these models at runtime scenarios a speedup of simulation is desirable.

Prediction speed at design time is of importance as well, as we can see at the transformation from Palladio Component Model (PCM) to QPNs. SimQPN is a simulation engine to analyze QPNs. At a time where multi-core processors are standard, parallelization is one way to speedup simulation. Decades of intensive research showed no general solution for parallel discrete event simulation, which provides reasonable speedups. In this context, Queueing Networks (QNs) and Petri Nets (PNs) have been extensively studied. We research the application of parallel simulation for QPNs. We developed a parallel simulation engine that employs application level and event level parallelism. The simulation engine parallelizes by the use of a conservative barrier-based synchronization algorithm. Here we optimized for common model structures and applied active waiting. The speedup for a single run depends on the capability of the model. Hence, we performed three case studies which all show speedups throughout parallelization.

1.19 Adaptive Instrumentation of Java Applications for Experiment-Based Performance Analysis

(***Henning Schulz, Albert Flaig, Alexander Wert, and André van Hoorn***)

Running load tests, instrumentation of selected application parts is a common practice in order to measure performance metrics. Instrumentation means to extend the target application by measurement probes while executing measurements. For instance, in Java bytecode instrumentation, additional commands are inserted into the bytecode of the target application. Since data generation is time-consuming, it may affect the target application and thus the measurement. Countering this problem, stepwise approaches execute several measurements while using only few measurement probes per measurement. Utilizing existing approaches, the target application has to be restarted in order to change the instrumentation. The resulting measurement overhead can cause the execution of stepwise measurements to be impracticable or bound to high manual effort. In this presentation, we introduce the Adaptable Instrumentation and Monitoring (AIM) framework enabling stepwise measurements without system restarts. Furthermore, we show the advantages of selective instrumentation with AIM over excessive instrumentations. For instance, we introduce an approach to highly precise and fully automated performance-model calibration. Thereby, the relative error is smaller than 4%, whereas excessive instrumentations introduce an error of up to 50%. Last not least, we present the embedding of AIM in the Kieker framework, merging the adaptability of AIM with the comprehensive monitoring and analysis capabilities of Kieker.

1.20 Using and Extending LIMBO for the Descriptive Modeling of Arrival Behaviors

(***Jóakim v. Kistowski, Nikolas Roman Herbst, and Samuel Kounev***)

LIMBO is a tool for the creation of load profiles with variable intensity over time both from scratch and from existing data. Primarily, LIMBO's intended use is the description

of load intensity variations in open workloads. Specifically, LIMBO can be used for the creation of custom request or user arrival time-stamps or for the re-parameterization of existing traces.

LIMBO uses the Descartes Load Intensity Model (DLIM) internally for the formalized description of its load intensity profiles. The DLIM formalism can be understood as a structure for piece-wise defined and combined mathematical functions. We will outline DLIM and its elements and demonstrate its integration within LIMBO.

LIMBO is capable of generating request or user arrival time stamps from DLIM instances. In a next step, these generated time-stamps can be used for both open workload based benchmarking and simulations. The TimestampTimer plug-in for JMeter already allows the former. LIMBO also offers a range of tools for easy load intensity modeling and analysis, including, but not limited to, a visual decomposition of load intensity time-series into seasonal and trend parts, a simplified load intensity model as part of a model creation wizard, and an automated model instance extractor.

As part of our LIMBO tutorial, we explain these features in detail. We demonstrate common use cases for LIMBO and show how they are supported. We also focus on LIMBOs extensible architecture and discuss how to use LIMBO as part of another tool-chain.

1.21 Using Java EE ProtoCom for SAP HANA Cloud (*Christian Klaussner and Sebastian Lehrig*)

Performance engineers analyze the performance of software architectures before their actual implementation to resolve performance bottlenecks in early development phases. Performance prototyping is such an approach where software architecture models are transformed to runnable performance prototypes that can provide analysis data for a specific target operation platform. This coupling to the operation platform comprises new challenges for performance prototyping in the context of cloud computing because a variety of different cloud platforms exists. Because the choice of platform impacts performance, this variety of platforms induces the need for prototype transformations that either support several platforms directly or that are easily extensible. However, current performance prototyping approaches are tied to only a small set of concrete platforms and lack an investigation of their extensibility for new platforms, thus rendering performance protoyping ineffective for cloud computing. To cope with this problem, we extended Palladios performance prototyping approach ProtoCom by an additional target platform, namely the SAP HANA Cloud, and analyzed its extensibility during the extension process. In this tool paper, we focus on illustrating the capabilities of our extension of ProtoCom. For this illustration, we use a simple example system for which we create a ProtoCom performance prototype. We particularly run this prototype in the SAP HANA Cloud.

1.22 Experience with Continuous Integration for Kieker (*Nils Christian Ehmke, Christian Wulf, and Wilhelm Hasselbring*)

Developing new features and extensions for the Kieker framework is very often an agile process. We are successfully using CI (Continuous Integration) as an assistance during the development for some years now, extending it more and more. Unit tests and performance benchmarks are executed regularly, static analysis tools ensure a constant code quality, and nightly builds allow an easy access to snapshot versions of Kieker. All of this is not only a support for more experienced Kieker developers, but also for new members joining the project.

Our presentation is an experience report about the success of CI in Kieker. We detail the structure, the used tools and the history of our CI infrastructure. We report on advantages and disadvantages recognized during the usage. Furthermore, the presentation includes an outlook for further activities regarding our CI infrastructure.

1.23 Towards Performance Awareness in Java EE Development Environments (*Alexandru Danciu, Andreas Brunnert, and Helmut Krcmar*)

This paper presents an approach to introduce performance awareness in integrated development environments (IDE) for Java Enterprise Edition (EE) applications. The approach automatically predicts the response time of EE component operations during implementation time and presents these predictions within an IDE. Source code is parsed and represented as an abstract syntax tree (AST). This structure is converted into a Palladio Component Model (PCM). Calls to other Java EE component operations are represented as external service calls in the model. These calls are parameterized with monitoring data acquired by Kieker from Java EE servers. Immediate predictions derived using analytical techniques are provided each time changes to the code are saved. The prediction results are always visible within the source code editor, to guide developers during the component development process. In addition to this immediate feedback mechanism, developers can explicitly trigger a more extensive response time prediction for the whole component using simulation. The talk will cover the conceptional approach, the current state of the implementation and sketches for the user interface.

1.24 Identifying Semantically Cohesive Modules within the Palladio Meta-Model (*Misha Strittmatter and Michael Langhammer*)

The Palladio meta-model is currently contained within one file and is subdivided into packages. This structure formed through the years, while the PCM was developed and extended. The top-level packages are partly aligned with the containment structure of the PCM submodels (e.g. repository, system). There are other top-level packages, which contain crosscutting or general concepts (e.g. parameter, composition, entity). Further

ones are concerned with quality dimensions (e.g. reliability, QoS Annotations). Some extensions distributed their new classes across packages, which were semantically fitting. Within the scope of the Palladio refactoring, this structure is being transformed into a modular structure. The goal is to divide the Palladio meta-model into smaller, semantically cohesive meta-models (meta-model modules). This has several advantages. The meta-model becomes better maintainable and easier to extend and understand. When a Palladio model is created, the model developer can choose the meta-model modules which are relevant to him and is not confused by the full amount of content from all extensions. Within this contribution, first, the current structure of the Palladio meta-model is briefly explained. Next, the semantically cohesive modules, which are currently contained or even scattered across the current structure, are presented and their interdependencies explained.

1.25 Evolution of the Palladio Component Model: Process and Modeling Methods (*Reiner Jung, Misha Strittmatter, Philipp Merkle, and Robert Heinrich*)

Palladio and its component meta-model have grown over time to support a wide variety of models for application performance prediction. The present meta-model is a large interwoven structure. Specific additions, e.g., for reliability and other quality properties, have been developed separately and resulted often in derived meta-models which cannot be used together. Furthermore, as discussed on KPDAYS 2013, certain views, typing and instance definitions lack a precise definition. Therefore, the Palladio Component Model (PCM) must evolve into a modular and easy to extend meta-model, and the Palladio tools must support this modularity to allow users to combine those Palladio functionality they need for their purposes. To achieve this modularity, we must solve two challenges. First, the modularization well be a long running task. Therefore, we need a process and road map to guide us from the present Palladio to the modular future Palladio. And second, the complexity of the existing metamodel and the need to keep the resulting meta-models maintainable, we need guidance for the modularization of the PCM and its tooling. Furthermore, the use methods must support future evolution steps. In this presentation, we first provide a brief summary on our method to construct modular meta-models based on aspect and view based modeling. This general distinction is supported by collected construction advice and contextual meta-model patterns. Furthermore, we illustrate how tools should be constructed around such metamodels. And second, we lay out a plan, how this evolution of the PCM can be realized in a stepwise approach which could also be distributed among different volunteers. As the tooling is an integral part of Palladio, the evolution of the meta-model will cause alterations to the tooling. Therefore, the plan encompasses both tool and meta-model evolution.

1.26 Toward a Generic and Concurrency-Aware Pipes & Filters Framework (*Christian Wulf, Nils Christian Ehmke, and Wilhelm Hasselbring*)

The Pipes-and-Filters design pattern is a well-known pattern to organize and execute components with sequential dependencies. The pattern is therefore often used to perform several tasks consecutively on large data streams, e.g., during image processing or dynamic analyses. In contrast to the pattern's familiarity and application, almost each common programming language lacks of flexible, feature-rich, fast, and concurrency-aware Pipes-and-Filters frameworks. So far, it is common practice that most developers write their own implementation tailored to their specific use cases and demands hampering any effective re-use. In this paper, we discuss Pipes-and-Filters architectures of several Java-based applications and point out their drawbacks concerning their applicability and efficiency. Moreover, we propose a generic and concurrency-aware Pipes-and-Filters framework and provide a reference implementation for Java called TeeTime.

1.27 Evaluation of Alternative Instrumentation Frameworks (*Dušan Okanović and Milan Vidaković*)

When monitoring application performance parameters under production workload, during continuous monitoring process, everything must be done to reduce the overhead generated by monitoring system. This overhead is highly unwanted because it can have negative effect on the end user's experience. While this overhead is inevitable, certain steps can be taken to minimize it.

Our system for continuous monitoring - DProf - uses instrumentation, and sends gathered data to the remote server for further analysis. After this data is analyzed, new monitoring configuration can be created in order to find the root cause of the performance problem. New parameters turn monitoring off, where performance data is within expected, and on, where data shows problems. This way, the overhead is reduced, because only problematic parts of software are monitored. The first version of our tool used AspectJ for instrumentation. The downside of this approach was the fact that the resulting bytecode is not fully optimized, generating even higher overhead than expected. Also, the monitored system had to be restarted each time monitoring configuration changed in order to apply new monitoring parameters.

We have explored the use of other AOP or AOP-like tools, mainly DiSL framework, with our system. The goal is to find a tool that has lower overhead than AspectJ. Support for runtime changes of monitoring configuration is also considered.

1.28 Cloud Application Design Support for Performance Optimization and Cloud Service Selection
(*Santiago Gómez Sáez, Vasilios Andrikopoulos, and Frank Leymann*)

The introduction of the Cloud computing paradigm and the significant increase of available Cloud providers and services have contributed in the last years to increase the number of application developers strongly supporting to partially or completely run their applications in the Cloud. However, application developers nowadays face application design challenges related to the efficient selection among a wide variety of Cloud services towards efficiently distributing their applications in one or multiple Cloud infrastructures. Standards like TOSCA allow for the modelling and management of application topology models, further supporting the application distribution capabilities, potentially in a multi-Cloud environment. However, such approaches focus on enabling portability among Cloud providers, rather than assisting the application developers in the Cloud services selection decision tasks. In this work we focus on the challenges associated with the application performance requirements and evolution, and therefore aim to define the means to support an efficient application (re-)distribution in order to efficiently handle fluctuating over time workloads. There are two fundamental aspects which must be taken into consideration when partially or completely running the application in the Cloud. Firstly, the distribution of the application in the Cloud has a severe effect on the application performance however it is not always evident whether it is beneficial or detrimental. Secondly, an application workload, and therefore its resources demands, fluctuates over time, and its topology may have to be adapted to address the workload evolution.

2 Papers

Using Java EE ProtoCom for SAP HANA Cloud*

Christian Klaussner
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
cfk@mail.uni-paderborn.de

Sebastian Lehrig
Software Engineering Chair
Chemnitz University of Technology
Straße der Nationen 62
09107 Chemnitz, Germany
sebastian.lehrig@informatik.tu-chemnitz.de

Abstract: Performance engineers analyze the performance of software architectures before their actual implementation to resolve performance bottlenecks in early development phases. Performance prototyping is such an approach where software architecture models are transformed to runnable performance prototypes that can provide analysis data for a specific target operation platform. This coupling to the operation platform comprises new challenges for performance prototyping in the context of cloud computing because a variety of different cloud platforms exists. Because the choice of platform impacts performance, this variety of platforms induces the need for prototype transformations that either support several platforms directly or that are easily extensible. However, current performance prototyping approaches are tied to only a small set of concrete platforms and lack an investigation of their extensibility for new platforms, thus rendering performance protoyping ineffective for cloud computing.

To cope with this problem, we extended Palladio's performance prototyping approach ProtoCom by an additional target platform, namely the SAP HANA Cloud, and analyzed its extensibility during the extension process. In this tool paper, we focus on illustrating the capabilities of our extension of ProtoCom. For this illustration, we use a simple example system for which we create a ProtoCom performance prototype. We particularly run this prototype in the SAP HANA Cloud, thereby showing that our extension can efficiently be applied within a practical context.

1 Introduction

Performance engineers analyze the performance of software architectures before their actual implementation to resolve performance bottlenecks in early development phases. This early detection and resolving of performance problems reduces fix-it-later costs and promises a high customer satisfaction, due to a high quality of service right from the beginning of system operation. Performance prototyping is such an approach to performance engineering. In performance prototyping, engineers transform software architecture models to runnable performance prototypes. These prototypes can provide analysis data for a specific target operation platform.

The coupling of performance prototypes to the operation platform comprises new challenges in the context of cloud computing. In cloud computing, a variety of different cloud

*The research leading to these results has received funding from the EU Seventh Framework Programme (FP7/2007-2013) under grant no 317704 (CloudScale).

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

platforms exists. Because the choice of platform impacts performance, this variety of platforms needs to be covered by transformations to performance prototypes [LLK13]. Therefore, prototype transformations either need to support several platforms directly or have to be easily extensible. However, current performance prototyping approaches are tied to only a small set of concrete platforms and lack an investigation of their extensibility for new platforms, thus rendering performance prototyping ineffective for cloud computing.

In related work, initial attempts have been made to support a larger quantity of platforms [Bec08, GL13]. However, such attempts remain on a conceptual level and do not provide transformation implementations that would make performance prototyping more efficient. Other works provide such implementations but are tied to single platforms only. For example, the implementation by Lehrig and Zolynski [LZ11] only has support for Java SE as a target platform. To the best of our knowledge, no related approach investigates the extensibility of performance prototypes.

To cope with this problem, we extended Palladio's Java SE performance prototyping approach ProtoCom [Bec08] by an additional target platform and analyzed its extensibility during the extension process. We decided to enrich ProtoCom by Java EE capabilities such that we could reuse first conceptual ideas of our previous work [GL13]. Moreover, we use the SAP HANA Cloud as concrete target platform because it represents a practically used cloud computing environment with Java EE support.

The contribution of this tool paper is an illustration of the capabilities of our ProtoCom extension. For this illustration, we use a simple example system for which we create a Java EE ProtoCom performance prototype. We particularly run this prototype in the SAP HANA Cloud, thereby showing that our extension can efficiently be applied within a practical context. For our results regarding the extensibility of ProtoCom, we refer to Klaussner's Bachelor's thesis [Kla14] (we identify both, implementation parts that are easily extensible and parts that can be improved regarding extensibility). His thesis particularly gives a complete technical overview of our ProtoCom extension.

This paper is structured as follows. We introduce our example system in Sec. 2. We use this system throughout our paper as running example. In Sec. 3, we describe the fundamentals of our work (performance prototyping with ProtoCom, Java EE, and the SAP HANA Cloud). Afterwards, we describe our Java EE extension to ProtoCom in Sec. 4. This extension allows us to use ProtoCom within SAP HANA Cloud in Sec. 5. In Sec. 6, we discuss related work in the area of performance prototyping. We close this paper by giving concluding remarks and an outlook on future work in Sec. 7.

2 Running Example: The Alice&Bob System

As running example, we use the *Alice&Bob* system as illustrated in Fig. 1. This system consists of two Java EE servers: JavaEE-Server-A allocates the Alice component that provides the interface IAlice with the operation callBob() and JavaEE-Server-B allocates the Bob component that provides the interface IBob with the operation sayHello(). The IAlice interface is provided to a user who can invoke its operation through a client-side technol-

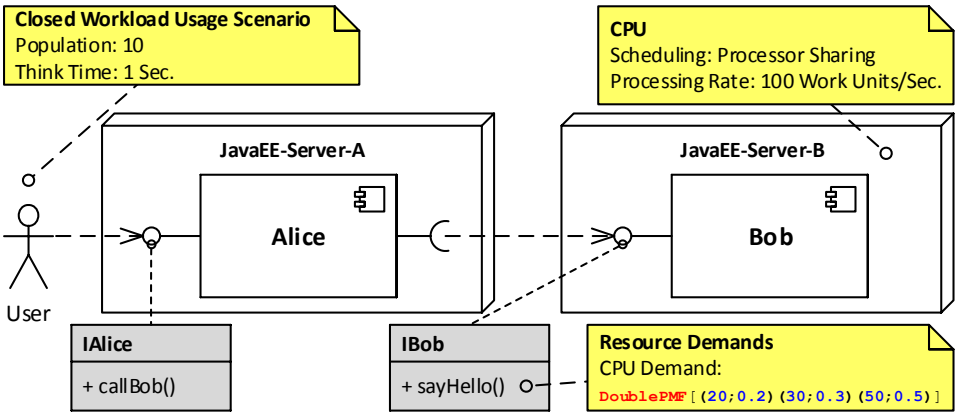


Figure 1: Alice&Bob System

ogy like a browser. This invocation can be received by a component on the server side – in our case by the Alice component implementing the IAlice interface.

Furthermore, we annotated performance-relevant information to the *Alice&Bob* system using yellow sticky notes. These sticky notes state that (1) the user population is 10 with a think time of 1 second within a closed workload, (2) the CPU of JavaEE-Server-B follows a processor sharing (round-robin) scheduling strategy while processing with 100 work units per seconds, and (3) calls to sayHello() cause a CPU demand as specified by the given probability mass function. The latter specifies that 20 CPU work units are demanded in 20% of the cases, 30 CPU work units are demanded in 30% of the cases, and 50 CPU work units are demanded in 50% of the cases.

3 Fundamentals

In this section, we describe the fundamentals needed to understand our extension to the performance prototyping approach ProtoCom. We accordingly describe the state-of-the art in performance prototyping with ProtoCom in Sec. 3.1. Afterwards, we describe the target platform for our extension – Java EE and the SAP HANA Cloud – in Sec. 3.2.

3.1 Performance Prototyping with ProtoCom

Software engineers and architects aim to analyze quality attributes of a software system, e.g., performance, early in the design process in order to avoid costs for subsequent adjustments. Performance prototyping is an approach that facilitates such analyses by simulating the performance of a software system in a realistic execution environment and under different load levels.

Palladio supports engineers in efficiently constructing such performance prototypes. For this support, the Palladio Component Model (PCM) provides a component-based Architecture Description Language (ADL). The PCM allows engineers to create a formalized model of the components and performance-relevant properties of a software architecture, similar to the model of our Alice&Bob system illustrated in Fig. 1.

After such a PCM model is created by performance engineers, it serves as input for the code generator ProtoCom, which transforms it into a runnable performance prototype for the desired target technology. Currently, ProtoCom supports the generation of performance prototypes for three target technologies: Java SE with RMI, Java EE with EJBs (Enterprise Java Beans), and Java EE with Servlets for the SAP HANA Cloud, which we introduce in this paper.

Our implementation transforms components from the model to Java EE Servlets. These Servlets can communicate via HTTP (cf. Sec. 4), while all previous implementations were based on RMI communication for distributing components. When running a generated performance prototype, an external HTTP load generator, e.g., JMeter, is used to simulate users interacting with the system according to the usage scenario specified in the model. Meanwhile, the performance prototype takes several mesasurements, e.g., response times, which allow a subsequent examination of the software architecture's performance. We give a more detailed description of this workflow in Sec. 5.

3.2 Java EE and SAP HANA Cloud

The Java Enterprise Edition (Java EE) is a set of specifications and APIs for Java that facilitate the development of enterprise software. This type of software is usually run on one or more servers (as illustrated in Fig. 1) and makes use of web technologies. There exist several implementations of the Java EE specifications, e.g., GlassFish and WildFly (formerly JBoss). Additionally, some projects implement only a subset of the specifications. For example, Apache Tomcat[1] implements Servlets, which are Java classes that can respond to requests. A common use case for Servlets is the HTTP request-response model for typical websites.

The SAP HANA Cloud is a Platform-as-a-Service (PaaS) that provides a Java EE environment and a cloud infrastructure for running enterprise applications. It is based on Apache Tomcat and includes additional services for applications, e.g., a document service that can be used to store unstructured or semi-structured data.

4 Java EE ProtoCom

Adding Java EE for the SAP HANA Cloud as target technology for ProtoCom required the implementation of new transformations and a new framework (the ProtoCom "runtime").

[1] `http://tomcat.apache.org`

When generating code from PCM instances, the entities and concepts of the source model have to be mapped to constructs of the target language. Although both the PCM and Java have similar constructs, e.g., interfaces and components/classes, they differ in expressiveness. Therefore, a one-to-one mapping is sometimes impossible. Tab. 1 lists the PCM entities and concepts regarded in our extension together with their respective Java mapping. Especially the provided and required roles of components need specific patterns to be transformed correctly [Kla14].

PCM Entity/Concept	**Java**
Interface	Interface
Component	Component class
Provided role	Port class
Required role	Context class
System	System class
Assembly context	Component class instance
Call action	RPC over HTTP
Control flow	Control flow
Resource environment	Resource environment class
Allocation	Allocation class
Usage scenario	[External]

Table 1: Mapping of PCM entities and concepts to Java, cf. [GL13, Kla14]

The addititions we made can be grouped intro three categories: inter-component communication, user interface, and load generator.

4.1 Inter-Component Communication

For the communication between components across resource container boundaries (i.e., ExternalCallAction entities in PCM), we use a custom, lightweight RPC protocol based on JSON and HTTP. Compared with other RPC protocols like SOAP, our custom protocol is easier to process in intermediary components and tools involved in the execution of the performance prototype. For example, when using Apache JMeter for the simulation of usage scenarios, data has to be exchanged between JMeter and the performance prototype. Thanks to JMeter's scripting capabilities (including JavaScript) the data received from the performance prototype can be processed without any manual parsing as would be the case with SOAP and XML.

Conceptually, our RPC method works similar to Java RMI. Components register themselves with a unique name at a central registry that is accessible via a Servlet. Other components can then contact this registry and obtain references to other named components. After a connection is established, method calls to remote components are initiated by sending the method name, parameter types, and arguments (serialized to JSON) to the target component, as shown in List. 1.

The formalParameters array consists of the type names of the formal parameters and is used – together with the name – to find the correct method to invoke at the destination, whereas the actualParameters array specifies the actual type names of the serialized arguments. These type names are required during deserialization in order to recreate the serialized objects. For example, the argument in List. 1 is deserialized as StackContext object and passed to the callBob0 method which expects a StackContext argument. In this case, the serialized argument consists of an empty JSON object, instructing the deserializer to create a new StackContext object to be passed to the method.

```
{
  "name":"callBob0",
  "formalTypes":["de.uka.ipd.sdq.simucomframework.variables.StackContext"],
  "actualTypes":["de.uka.ipd.sdq.simucomframework.variables.StackContext"],
  "arguments":[{}]
}
```

Listing 1: RPC protocol representation of a call to Alice's callBob method

4.2 User Interface

Since the Java EE performance prototypes run on the SAP HANA Cloud, they are inaccessible through a console. Therefore, we developed an HTML user interface that can be accessed through a web browser. Fig. 2 shows a screenshot of the user interface used to operate the performance prototype generated from the *Alice&Bob* system introduced in Sec. 2. It allows performance engineers to specify the location of the central component registry and to start particular modules, i.e., startable entities of the performance prototype like resource containers (that start all allocated components) and systems. Additionally, it provides downloads for the transformed usage scenarios (JMeter files) and analysis results.

4.3 Load Generator

In order to generate load on the performance prototype, we provide an interface for external load generators, e.g., Apache JMeter. The ProtoCom transformations automatically generate a JMeter test plan for each usage scenario in the model. ExternalCallActions are realized by sending HTTP requests to the respective components according to the RPC protocol described in Sec. 4.1.

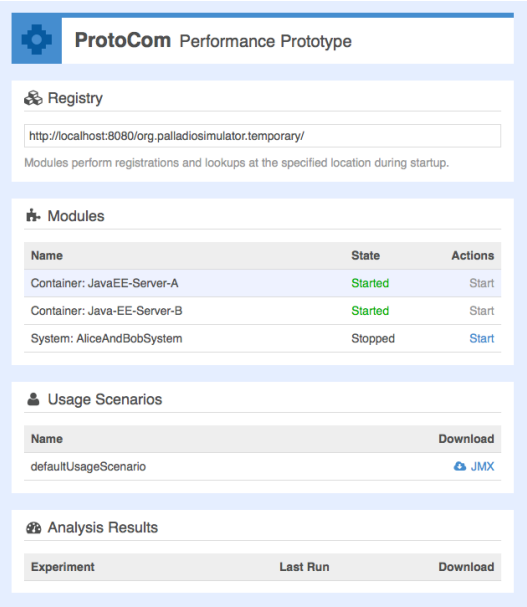


Figure 2: The user interface of the performance prototype

5 Using Java EE ProtoCom for SAP HANA Cloud

This section describes how ProtoCom is used in the SAP HANA Cloud. First, it explains how the abstract resource demands from the PCM are translated to actual demands on the hardware of the cloud platform. Afterwards, it shows the workflow of transforming and executing performance prototypes.

5.1 Hardware Calibration

Before running a performance prototype, the abstract resource demands for CPU and HDD specified in the SEFFs of the PCM components have to be translated to demands for the hardware that the performance prototype is running on. For that purpose, ProtoCom provides a set of calibration strategies, e.g., Fibonacci, which simulates CPU intensive tasks with minimized RAM access [LZ11]. The calibration computes several iterations – in this case the n^{th} Fibonacci numbers – and measures the time required for these computations. The results are stored in a calibration table for later lookup, as shown in Tab. 2. When a demand from the model has to be translated to a real hardware demand, the calibration table is used to find the iteration count n for the given demand. For example, a CPU demand of 0.036 seconds would be achieved by computing the 40^{th} Fibonacci number.

[2]Results taken from `http://www.cs.utsa.edu/~wagner/CS3343/recursion/fibrecurs2.html`

n	Time
10	0.032 Sec.
20	0.033 Sec.
30	0.035 Sec.
40	0.036 Sec.
50	0.038 Sec.
...	...

Table 2: A calibration table for the Fibonacci strategy[2]

Lehrig and Zolynski [LZ11] validate such calibration tables by creating a model with a processing resource that has a processing rate of 1000 units and a closed workload usage scenario in which a single user repeatedly invokes a task that consumes 1000 units. Ideally, the response time of this usage scenario should consume a time of 1 second. The results for the Large Chunks HDD calibration show that the mean response time of the usage scenario is indeed around 1 second, with deviations resulting from external factors, e.g., process scheduling behavior controlled by the operating system.

The calibration of the hardware resources is a time-consuming task and has to be performed once for each hardware configuration. Regarding the SAP HANA Cloud, all calibration strategies of ProtoCom work for both local installations (development and testing) and the actual cloud platform.

5.2 Prototyping Workflow for SAP HANA Cloud

This section describes the typical workflow of performance prototyping with ProtoCom for SAP HANA Cloud. The activity diagram in Fig. 3 shows all steps involved.

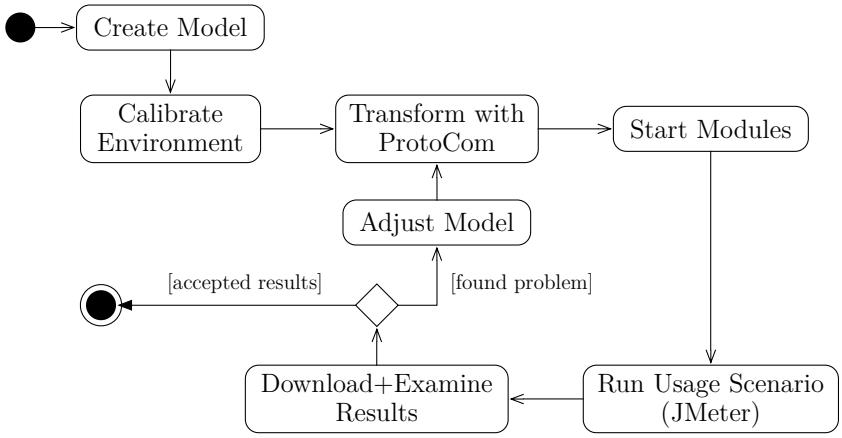


Figure 3: Performance prototyping workflow with ProtoCom for SAP HANA Cloud

Performance engineers start with creating a model of the software architecture to examine. Afterwards, they calibrate the environment that the performance prototype will run on. This can either be the environment of the cloud infrastructure or a local installation of the SAP HANA Cloud runtime. Subseqently, the model is transformed to a performance prototype and deployed on the target platform. At this point, performance engineers interact with the user interface illustrated in Fig. 2. They start the modules, i.e., the resource containers and the system, and download the generated JMeter test plan. Loading this test plan into JMeter allows them to configure and run the measurements.

Finally, the results of the analysis are provided as a download in the user interface. Performance engineers examine these results in the Palladio-Bench and decide if the model needs further improvements, e.g., caused by a performance bottleneck, or if the results are satisfying. In this case, software engineers can start with the implementation of the architecture.

Following this workflow for our Alice&Bob system shows that the performance prototype generated from the model in Fig. 1 can be deployed in the SAP HANA Cloud to take performance measurements. Evaluating the results of the prototype and comparing them to the results of ProtoCom for Java SE with RMI is part of our future work.

6 Related Work

Besides works on ProtoCom, there are only a few approaches on performance prototyping. Becker et al. [BDH08] describe these approaches and their limitations compared to ProtoCom in their related work. In this section, we therefore only focus on related work on ProtoCom directly.

In his PhD thesis, Becker [Bec08] provides the groundwork for ProtoCom and for transforming PCM instances to performance prototypes using Java EE EJBs (Enterprise Java Beans). However, the performance prototypes generated by these transformations suffer from several usability issues. For example, the generated code requires manual adjustments after the transformations. Furthermore, the deployment process of the generated performance prototypes proved to be inefficient.

Building on this work, Lehrig and Zolynski [LZ11] present an improved version of ProtoCom that aims to resolve these shortcomings. They extend the transformations and the framework of ProtoCom such that the generated performance prototypes target the Java SE technology and use Java RMI for inter-process communication. However, the transformations in this version of ProtoCom proved to be slow and inextensible due to the use of Xpand[3] templates as a means for model-to-text transformations. Therefore, Lehrig and Zolynski provide a reimplemented version of ProtoCom ("ProtoCom 3") [Kar] that replaces the Xpand templates with templates for the programming language Xtend 2[4]. This version of ProtoCom served us as a good basis for analyzing the extensibility of ProtoCom. As Klaussner describes in his Bachelor's thesis [Kla14], the new transformation parts in-

[3] `http://www.eclipse.org/modeling/m2t/?project=xpand`
[4] `http://www.eclipse.org/xtend`

deed provide a good extensibility. However, some parts of ProtoCom's generic framework still provide room for improvement regarding extensibility [Kla14].

Since the release of Becker's PhD thesis, a new version of Java EE EJBs was published, providing features that simplify the transformations. Furthermore, there is a need for performance prototype transformations targeting multiple platforms [LLK13]. These developments led to the addition of new Java EE EJB transformations by Giacinto and Lehrig [GL13]. However, these transformations are provided only on a conceptual level and also lack an implementation. In our work, we provide such a Java EE implementation and use it within the SAP HANA Cloud, thus making our approach more efficient.

7 Conclusions

In this tool paper, we introduce a novel ProtoCom version that is capable of automatically generating Java EE performance prototypes that can directly operate within the SAP HANA Cloud (out-of-the-box). In this context, we give an overview of novel Java EE features in ProtoCom and describe how generated performance prototypes are used within the SAP HANA Cloud.

Performance engineers can now efficiently generate performance prototypes based on Java EE. Engineers achieve best efficiency within the SAP HANA Cloud because our ProtoCom version is capable of using dedicated SAP HANA Cloud features such as its document service (e.g., to store calibration tables). However, engineers can now also analyze other Java EE platforms (Glassfish, Tomcat, etc.) more efficiently because our ProtoCom version introduces features shared among all of such platforms (e.g., RPC over HTTP communication). Here, we profit from the standardization process behind Java EE. Furthermore, we showed that engineers can easily extend ProtoCom if new requirements arise [Kla14].

In future work, we want to conduct several case studies within the SAP HANA Cloud using our novel ProtoCom version. We plan to reuse the case studies from our earlier work with ProtoCom in virtualized environments [LZ11]. This reuse will allow us to compare our previous results with new results gained within a less-controlled environment (the SAP HANA Cloud), eventually leading to assessing the predictability of cloud computing environments.

References

[BDH08] Steffen Becker, Tobias Dencker, and Jens Happe. Model-Driven Generation of Performance Prototypes. In Samuel Kounev, Ian Gorton, and Kai Sachs, editors, *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 79–98. Springer Berlin Heidelberg, 2008.

[Bec08] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. PhD thesis, University of Oldenburg, Germany, January 2008.

[GL13] Daria Giacinto and Sebastian Lehrig. Towards Integrating Java EE into ProtoCom. In *KPDAYS*, pages 69–78, 2013.

[Kar] Karlsruhe Institute of Technology. ProtoCom - SDQ Wiki. `http://sdqweb.ipd.kit.edu/wiki/ProtoCom`. Retrieved: 07/12/2014.

[Kla14] Christian Klaussner. Extensible Performance Prototype Transformations for Multiple Platforms. Bachelor thesis, Software Engineering Group, University of Paderborn, Software Engineering Group, Paderborn, Germany, July 2014.

[LLK13] Michael Langhammer, Sebastian Lehrig, and Max E. Kramer. Reuse and Configuration for Code Generating Architectural Refinement Transformations. In *VAO '13*. ACM, 2013.

[LZ11] Sebastian Lehrig and Thomas Zolynski. Performance Prototyping with ProtoCom in a Virtualised Environment: A Case Study. In *Proceedings to Palladio Days 2011, 17-18 November 2011, FZI Forschungszentrum Informatik, Karlsruhe, Germany*, November 2011.

Towards Modeling and Analysis of Power Consumption of Self-Adaptive Software Systems in Palladio

Christian Stier, Henning Groenda
FZI Research Center for Information Technology
Karlsruhe, Germany
{stier|groenda}@fzi.de

Anne Koziolek
Karlsruhe Institute of Technology
Karlsruhe, Germany
koziolek@kit.edu

Abstract: Architecture-level evaluations of Palladio currently lack support for the analysis of the power efficiency of software systems and the effect of power management techniques on other quality characteristics. This neglects that the power consumption of software systems constitutes a substantial proportion of their total cost of ownership. Currently, reasoning on the influence of design decisions on power consumption and making trade-off decisions with other Quality of Service (QoS) characteristics is deferred until a system is in operation. Reasoning approaches that evaluate a system's energy efficiency have not reached a suitable abstraction for architecture-level analyses. Palladio and its extension SimuLizar for self-adaptive systems lack support for specifying and reasoning on power efficiency under changing user load. In this paper, we (i) show our ideas on how power efficiency and trade-off decisions with other QoS characteristics can be evaluated for static and self-adaptive systems and (ii) propose additions to the Palladio Component Model (PCM) taking into account the power provisioning infrastructure and constraints.

1 Introduction

Palladio [BKR09] enables the evaluation of quality characteristics such as performance, cost or reliability for component-based software systems at early design stages. By predicting the performance and reliability of software systems it is possible to reason on design alternatives and infer whether agreed upon Service Level Agreements (SLAs) can be maintained. While Palladio accounts for the execution environment of software, its focus is on software-centric design decisions. Currently Palladio has limited support for data center design and sizing decisions. In particular, power consumption or provisioning are not considered although they are a major cost-factor in data centers. Power consumption accounts for roughly 15% of a data center's Total Cost of Ownership (TCO) [GHMP08]. A data center needs to be equipped with suitably sized power provisioning infrastructure to avoid risking blackouts under peak load. Cost of an additional Watt of provisioned peak power is estimated around $\$11$ to $\$12.5$ [FWB07, GHLK$^+$12]. When power provisioning

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

and cooling costs are considered, the cost associated with power consumption is responsible for 40% of a data center's TCO [GHMP08].

Even though power consumption decisively determines the TCO of a software system it is currently not sufficiently considered on an architectural level. Tradeoff decisions between power consumption and other quality dimensions are deferred to deployment time. The impact of design decisions on the energy efficiency of a software system can only be determined by accounting for the power consumption characteristics of the deployment environment.

Previous work on energy consumption analysis of software architectures focuses on specific architectural styles [SEMM08]. As it focuses on power consumption induced by communication it cannot be applied to predict the effect of individual design decisions on power consumption. Other approaches make limiting assumptions regarding the usage context [GWCA12] or application characteristics [MBAG10] which restrict their applicability outside of their specific problem domain. Brunnert et al. [BWK14] introduce the specification and evaluation of power consumption characteristics to Palladio. The authors assume power consumption to follow the same pattern for all physical machines. Even though a fixed model may accurately capture power consumption of current machines, it likely will become inaccurate in the future when considering recent trends in energy proportionality of physical machines [HP13]. Brunnert et al. perform an average case power consumption analysis. Consequently, it is not possible to identify phases in which the power consumption surpasses critical limits. As electricity pricing schemes usually factor in both peak power consumption and total energy consumption it is critical to take into account both average and peak power consumption for the modeled system [ZWW12].

This paper proposes (i) an approach enabling trade-off decisions between multiple quality characteristics of static and self-adaptive software systems. Our approach accounts for the operation of software systems in a data center. Our second contribution is (ii) an explicit model of the power consumption of software systems and the power provisioning infrastructure integrated into Palladio. The paper describes the *Power Consumption Analyzer (PCA)* approach leveraging the model and allowing continuous power consumption analysis. It supports reasoning on maintaining power consumption thresholds leveraging Palladio simulations and analysis of power-conscious *self-adaptation tactics* [PLL14] on an architectural level. Power consumption properties are evaluated in a post-simulation analysis that requires no modification of the simulation. In order to enable the analysis of power-conscious self-adaptation tactics, we extend the Palladio-based SimuLizar approach [BLB13] to support power consumption evaluations at intra-simulation using our PCA.

We show that our approach can be applied to evaluate whether limits in power consumption are adhered to for a given usage context. We are able to identify peaks in power consumption and violations not only for individual physical machines but also for Power Distribution Units (PDUs). By accounting for peak power consumption we improve the prediction of a data center's TCO when compared to previous work [BWK14].

This paper is structured as follows. Section 2 outlines foundations of our approach. Section 3 discusses related work. Section 4 introduces our model extensions to PCM. Section 5 sketches how the PCA interprets these model extensions to support power consumption

analysis. Finally, Section 6 summarizes the work presented in this paper and outlines our plans for future work.

2 Foundations

The *Palladio* [BKR09] approach enables software architects to predict quality characteristics of a component-based software architecture. The architecture is specified in the *Palladio Component Model (PCM)*. Software systems specified in PCM are assumed to be static: PCM does not include a specification of architectural runtime adaptations. Quality characteristics of a system defined in PCM can be predicted using analytical solvers [KBH07, KR08] or simulators [BKR09, MH11, BBM13].

Individual modeling concerns in PCM are separated into specialized submodels or views. PCM encompasses an explicit structural view on the hardware deployment environment onto which software components are deployed. The deployment environment is specified in the *Resource Environment* model. The *Resource Environment* model consists of a set of nested *Resource Containers*. Outer containers represent physical parts of a data center such as racks and compute nodes, whereas inner containers serve as operating system or virtualization layers. Resource Containers host a set of *ProcessingResourceSpecifications* where each specification represents a resource to be used by components, e.g. CPU or HDD.

SimuLizar by Becker et al. [BBM13, BLB13] extends Palladio to enable a systematic design of self-adaptive software systems. Self-adaptive software systems adapt their configuration at run-time to cope with changing environmental properties such as varying user load. One example for such a reconfiguration is the migration of a virtual machine from an over- to an under-utilized node. Reconfigurations are the outcome of self-adaptation tactics. Becker et al. subdivide tactics into "a condition (input) and a self-adaptation action (output)" [BLB13]. The condition specifies when a self-adaptation action is triggered depending on measurements taken via system probes. Probes are attached to PCM in the Palladio Measurement Specification (PMS) model. Currently, the PMS only allows to capture performance-centric measurements such as the response time or utilization. Analysis of the self-adaptive system is carried out with the SimuLizar simulator. The simulator allows predicting quality characteristics of a self-adaptive software system specified according to the SimuLizar approach. Software architects can use the provided simulative analysis to reason on the impact of self-adaptation mechanisms on quality characteristics of the simulated system.

Energy-conscious self-adaptation tactics form a subset of self-adaptation tactics that "modify runtime software configuration for the specific purpose of lowering energy consumption" [PLL14]. They are aimed at reducing consumption over a period of time. *Power-conscious self-adaptation tactics* perform reconfigurations to reduce the power draw of a software system at a specific point time. Both energy- and power-conscious self-adaptation tactics rely on consumption measurements or estimates when reasoning on the benefit of performing an adaptation.

Power models are used to estimate power consumption in absence of actual power measurements [RRK08]. A *power model* describes power consumption of hardware and software components, individual services or complete server nodes based on a set of system metrics. The fundamental assumption of power models is that power consumption correlates with the values of a set of system metrics, e.g. CPU utilization or I/O throughput. Examples include linear regression models correlating CPU utilization [FWB07] or software performance counters [ERKR06] with power consumption. Power models evaluate power consumption at a stationary point. In order to determine energy consumption over a time-frame, numerical integration algorithms can be applied.

3 Related Work

Seo et al. investigate the energy consumption of different architectural communication styles [SEMM08]. The authors present a set of evaluation models for specific styles. Among the analyzed communication styles are client-server and publish-subscribe. Their approach focuses on energy consumption caused by communication and disregards all other aspects of a software system. While their approach allows to compare energy efficiency of employing specific communication styles against each other it can not be applied to reason on power consumption of other architectural design decisions.

An approach for multi-objective architecture optimization for embedded systems is proposed by Meedenyia et al. [MBAG10]. In their paper the authors focus on the tradeoff between reliability and energy consumption. A model that evaluates the energy consumed by service calls is applied to predict workload-dependent energy consumption. In order to account for idle consumption the authors additionally include a static consumption offset. All services offered by a component are assumed to cause the same energy consumption. This is an acceptable abstraction for the embedded system domain where one component handles a homogeneous task, e.g. controlling the brakes based on sensor signals. The parameter space of these sensors are restricted by physical constraints such as the maximum speed of a car. This is not the case for other other domains like enterprise software. Business components offer an array of services with a large input parameter space. Depending on parameter values, business components require largely varying resource demands. Assuming that all services of a component consume the same amount of energy consequently would result in imprecise consumption predictions.

The design of energy efficient self-adaptive software systems on an architectural level has already been investigated by Götz et al. [GWCA12, GWR+13]. Main goal of the proposed architectural design framework is to find optimal system configurations for single users. This is achieved by adapting the runtime configuration of the system to the QoS requirements of a single user. Multi-user scenarios are not considered by the approach. Hence it cannot be applied to the design of self-adaptive systems that are used by multiple users. Götz argues that QoS requirements could be checked against an architectural model of the self-adaptive systems using simulation [GÏ3, p. 103f.]. The developer leaves the development of a concept for this evaluation open to be addressed in future work.

Brunnert et al. [BWK14] outline an approach for capturing application profiles that subsume the core characteristics of a static software system. These profiles are used to make QoS-driven deployment decisions for different software architectures. Runtime reconfigurations are not considered by the authors. Brunnert et al. enhance power consumption characteristics as part of PCM's Resource Environment specifications to enable design-time power consumption analysis for software systems. Reasoning on power consumption is limited to an average-case analysis of static software systems. It is thus not possible to identify whether restrictions on peak power consumption are violated at specific points in time. Consequently, sizing decisions for the power provisioning infrastructure are therefore limited.

CloudSim by Calheiros et al. [CRB+11] is a simulator for Infrastructure-as-a-Service (IaaS) data centers. It focuses on the operation and optimization of running virtual machines (VMs) in a data center. All information is provided as Java source code extending the CloudSim simulation environment. There is no model abstraction of system entities and their relation. Unlike Palladio, CloudSim does not consider parametric dependencies and has no explicit usage model. Rather it simulates VMs as isolated entities. Power consumption predictions per node are carried out based on the CPU utilization aggregated over all VMs that are deployed on the same node. VM load is modeled directly as an utilization function over time. Reasoning on the power consumption of a software system requires accurate utilization descriptions.

4 Modeling the Power Consumption Characteristics of Software Systems

In order to reason on the power consumption of a software system on architectural level its consumption properties need to be captured as part of its architectural description. This section outlines our extension of PCM with power consumption characteristics.

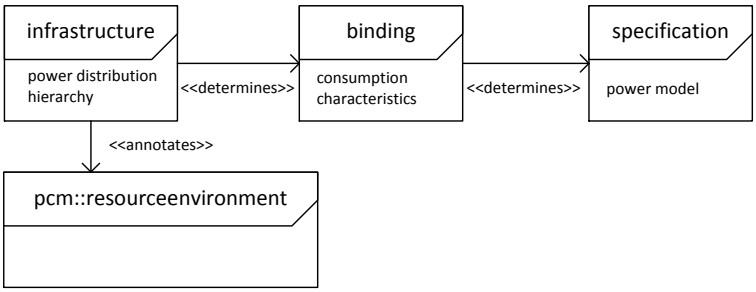


Figure 1: Relation between models used for specifying power consumption characteristics of a software system

PCM's Resource Environment models the hardware environment strictly by their performance and reliability characteristics. Our model annotates the hardware components in the

Resource Environment with their power consumption characteristics. Figure 1 depicts the relation between the model extensions and PCM. Our model introduces a model abstraction of the system's power provisioning infrastructure (*Power Infrastructure*). *Power Specification* describes models for evaluating the power consumption of hardware components and the power provisioning infrastructure. The *Power Binding* model links both models by binding each element in the infrastructure to the model used for evaluating its power consumption.

This section is organized as follows. Section 4.1 discusses advantages and disadvantages of different extension mechanisms that were considered for introducing power consumption characteristics to PCM. Section 4.2 presents the power provisioning model. In section 4.3 an overview is given on the model for specifying power models. Section 4.4 outlines the model used to specify consumption characteristics of hardware components.

4.1 Extending the Palladio Component Model

There are three approaches to extend PCM by another quality dimension. They are presented in the following including their advantages and drawbacks.

First, PCM can be invasively modified to include additional quality characteristics. This approach was taken for reliability [BKBR12] and is proposed by Brunnert et al. for power consumption predictions [BWK14]. The main disadvantage of an invasive extension is that it breaks support of existing tooling.

Second, PCM's profile extension mechanism [KDH+12] or EMF's child creation extenders [Mer08] can be used to introduce new properties and elements to PCM. These approaches are best taken when invasive changes to PCM are required that should not be propagated to all PCM-based tooling.

Third, it is possible to introduce a new meta-model which annotates or references existing PCM elements. As this alternative annotates existing model elements, it should only be chosen when no backwards navigation is needed or indicated.

We chose the last option where a separate model represents power consumption characteristics of the modeled software system. Model elements that have a counterpart in PCM's Resource Environment are annotated with their consumption characteristics. The reasons for choosing this extension mechanism are as follows. Wile power consumption is an important quality characteristic, it is not essential to other characteristics such as performance. Hence a non-invasive extension is indicated. Power consumption properties of hardware have no direct implications on their performance. It consequently suffices to introduce power consumption characteristics to PCM in an annotation model.

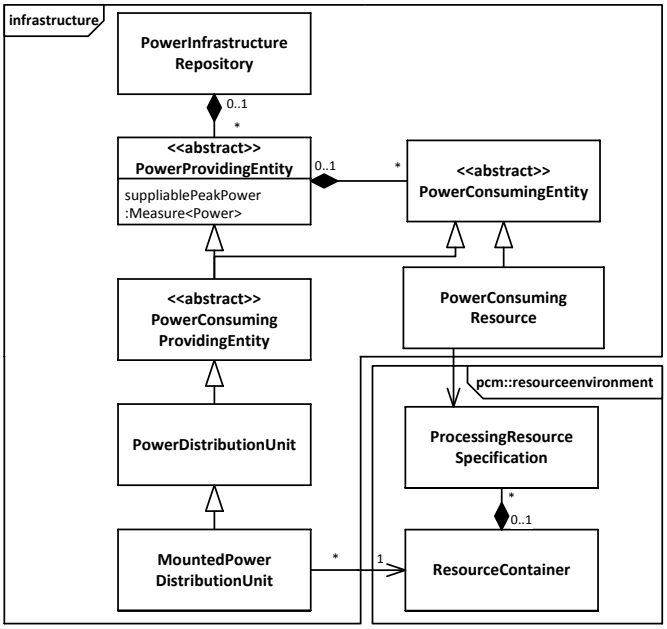


Figure 2: Power Infrastructure meta-model

4.2 Power Provisioning Infrastructure

The power provisioning infrastructure of data centers is typically structured hierarchically [FWB07]. It consists of a hierarchy of entities that provide and distribute power, e.g. PDUs, and entities that consume power, such as the physical components of a node. Figure 2 depicts our proposed model for the power provisioning infrastructure.

PowerInfrastructureRepository hosts a set of power provisioning infrastructure model descriptions. Typically only one infrastructure model is considered. However, the repository enables an easier modeling of alternative power provisioning infrastructures.

PowerConsumingEntity subsumes all components in the system that consume power.

PowerConsumingResource represents a physical resource that consumes power. It annotates a *ProcessingResourceSpecification* in a PCM instance with its consumption properties. Every PowerConsumingResource is a PowerConsumingEntity. An example for PowerConsumingEntity is the CPU of a compute node.

A *PowerProvidingEntity* distributes power to a set of *PowerConsumingEntities* nested below them. *suppliablePeakPower* defines the peak power that all PowerConsumingEntities connected to it can draw in total at any point in time.

Aside from entities that only provide or consume power there are also components that take on both providing and consuming roles, e.g. PDUs. *PowerConsumingProvidingEntity*

subsumes these entities.

PowerDistributionUnit extends *PowerConsumingProvidingEntity*. A PDU is connected to a power source (PowerProvidingEntity) from which it draws power. The PDU then further distributes power to connected PowerConsumingEntities.

MountedPowerDistributionUnit further specializes PowerDistributionUnit. It links the PDU to a ResourceContainer in PCM's Resource Environment. In essence, a mounted PDU corresponds with a Power Supply Unit of a node or a rack-mounted PDU. The relationship between a mounted PDU and its ResourceContainer is explicitly modeled so that consumption properties of nodes and racks can individually be traced.

4.3 Power Model Specifications

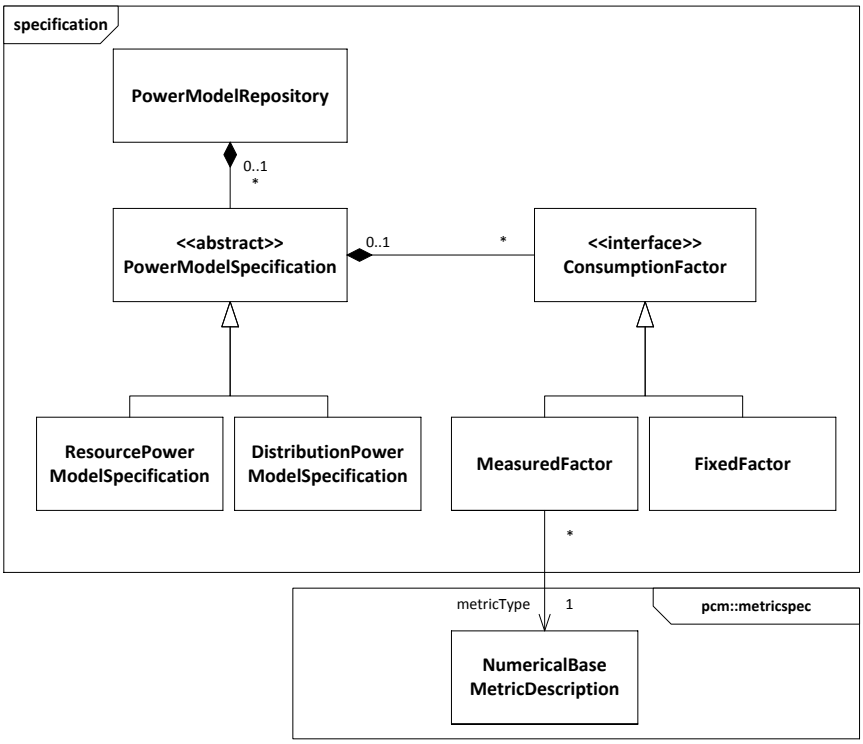


Figure 3: Power Specification meta-model

The *Power Specification* model shown in Figure 3 enables an explicit specification of power models and their input parameters. The calculation method is not specified in the Power Specification model instance. Rather, a calculator is implemented for each instance as part

of PCA as is explained further in Section 5.

The Power Specification Model is designed for unit support, e.g watt and ampere. Palladio's stochastical expression language *StoEx* [Koz08] was considered for specifying power models but ruled out due to lacking support for metric units.

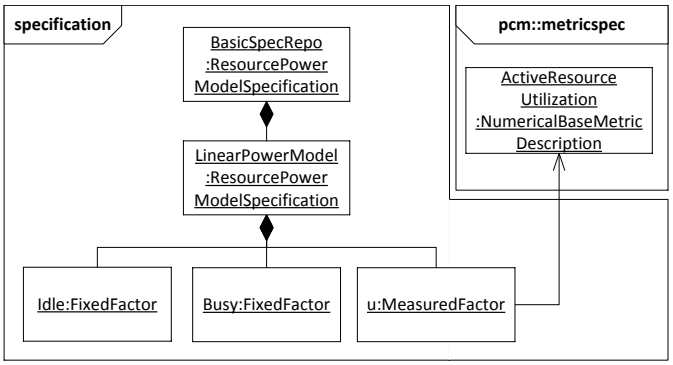


Figure 4: Linear power model represented as an instance of the PowerModelSpecification

A *PowerModelRepository* contains a set of *PowerModelSpecifications*. PowerModelSpecification represents a certain type of power consumption model (c.f. Figure 3). In the following, we will explain the concepts of the model based on the exemplary *LinearPowerModel* instance displayed in Figure 4. The depicted *LinearPowerModel* represents a simple linear regression-based power model for compute nodes as proposed by Fan et al [FWB07]. Fan et al. identify a close-to-linear relation between power consumption of a whole node and its CPU utilization . The linear power model predicts the power consumption $P(u)$ under utilization u by interpolating between the node's power consumption in an *Idle* and *Busy* state: $P(u) = P_{Idle} + (P_{Busy} - P_{Idle}) \cdot u$.

A *ConsumptionFactor* specifies an input parameter of a power model. In the example, *Idle*, *Busy* and *u* are consumption factors. ConsumptionFactors are further distinguished into *Fixed* and *Measured Factors*.

Fixed Factors are independent from measurements and describe static power consumption characteristics of a system. For the linear power consumption model the fixed factors are made up of the CPU's power consumption under *Idle* (P_{Idle}) and *Busy* (P_{Busy}) load.

As their name implies, *Measured Factors* are extracted from the simulated system through measurements. A Measured Factor's metric is specified in direct reference to Palladio's *Metric Specification Framework* [Leh14]. For the linear power model the measured CPU utilization *u* is included as a Measured Factor.

Similar to Palladio's Resource Repository, instances of the Power Specification model are intended for reuse. Linear and other power models allow to predict the power consumption of a wide range of systems. Hence they are defined separately from the Power Infrastructure model.

This section discussed the specification of power models as a set of input parameters.

Consumption parameters alone do not suffice in evaluating the power consumption. It is necessary to define how the predictions are calculated from these parameters. We opted for a code-based realization of the calculation methods of power models. Section 5 provides details on the calculation.

4.4 Specifying Consumption Characteristics of the Infrastructure

Power consumption characteristics of the infrastructure are not directly specified in the infrastructure model. They are maintained in the separate *Binding* model and referenced from the Infrastructure model. The main rationale behind separating these two models lies in increased reuse and ease of variation compared to a direct specification of consumption characteristics in the infrastructure model. E.g., power consumption characteristics of an IBM System X3350M3 server can be described in terms of its consumption in idle (230 W) and busy states (410 W) as is done by Brunnert et al. [BWK14].

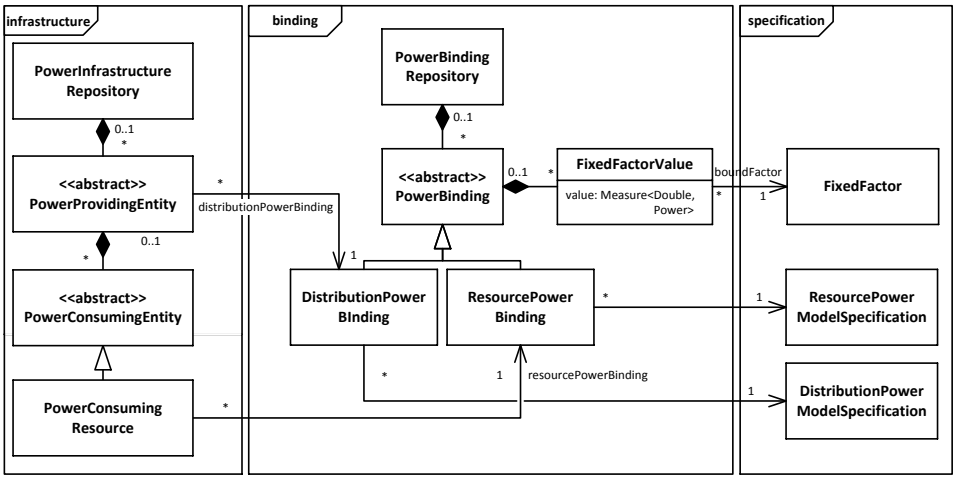


Figure 5: Power Binding meta-model

Figure 5 depicts our proposed binding model. It specifies the relation between the entities in the provisioning infrastructure and power models. Both *Resource* and *Distribution-PowerBinding* specialize *PowerBinding*.

A PowerBinding comes with a set of *FixedFactorValues*. Every FixedFactorValue determines the value of a FixedFactor. In case a server's power consumption can be described by a linear power model, the FixedFactorValues of the respective PowerBinding would specify the consumption of the node in a busy and idle state in Watt.

A *ResourcePowerBinding* defines the consumption characteristics of a type of PowerConsumingResource. Continuing with the previous example, all X3350M3 nodes in a system can reference the same binding. Should a CPU-based linear power consumption model not

sufficiently capture consumption characteristics of a node, e.g. because it is exclusively used as a file server, the PowerConsumingResource can reference a more suitable binding.

Every PowerProvidingEntity references a *DistributionPowerBinding*, which defines with what values the input parameters of a DistributionPowerModelSpecification are instantiated. As for ResourcePowerBinding, a DistributionPowerBinding can be reused across multiple PowerProvidingEntities.

5 Power Consumption Analyzer

The *Power Consumption Analyzer (PCA)* evaluates the power consumption of a software system. The analysis requires the specification of the system as a PCM instance and corresponding consumption annotations in the shape of a Power Consumption model (c.f. Section 4) instance. As part of a needs analysis for architecture-level power consumption, we identified two use cases covering static and self-adaptive software systems. PCA support both use cases, which are as follows: The *Power Consumption Analyzer (PCA)* evaluates the power consumption of a software system. The analysis requires the specification of the system as a PCM instance and corresponding consumption annotations in the shape of a Power Consumption model (c.f. Section 4) instance. As part of a needs analysis for architecture-level power consumption, we identified two use cases covering static and self-adaptive software systems. PCA support both use cases, which are as follows:

In the first use case, a software architect wants to predict power consumption of the system in a specific usage context. The software architect intends to compare the impact of design decisions on the systems' power consumption. In this case, the structure and consumption characteristics of hardware components have no impact on the system performance. Consequently, power consumption is analyzed as part of a *post-simulation* step. The advantage of analyzing power consumption separate from the simulation is that different power provisioning infrastructures can be compared without requiring additional time-intensive simulation runs. A new simulation is only necessary when both performance and power consumption characteristics have changed, e.g. because new servers are added to the resource environment.

In the second case, the software architect is interested in evaluating the impact a power-conscious self-adaptation tactic has on multiple quality characteristics. Power-conscious self-adaptation tactics use power consumption measurements to reason on the power efficiency of the system and issue system reconfigurations. The mutual dependency between current system architecture and power consumption requires an intra-simulation analysis.

The implementation of PCA is designed for both intra- as well as post-simulation power consumption analysis. Figure 6 sketches the integration of the PCA with the Quality Analysis Lab developed by Lehrig [Leh14] that is part of the upcoming Palladio release 3.5. The PCA evaluates the power consumption for a given *EvaluationContext*. This context consists of an instance of our Power Consumption model, the PCM instance it annotates, and a set of relevant measurements taken from the simulated system-under-analysis.

PCA computes the power consumption of individual PowerConsumingResources using

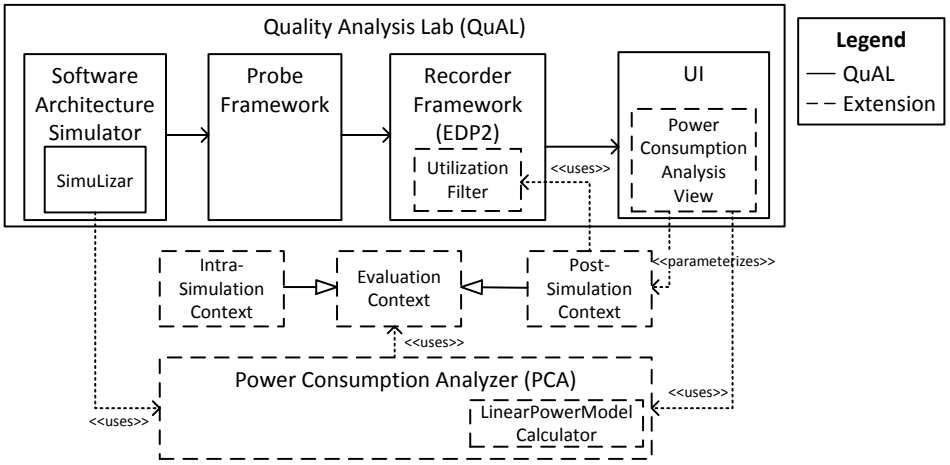


Figure 6: Integration of the Power Consumption Analyzer with Palladio's Quality Analysis Lab (QuAL) [Leh14]

calculators. A calculator programmatically evaluates power consumption of infrastructure elements. For example, the *LinearPowerModelCalculator* for the linear power model depicted in Figure 6 determines the power consumption of the node based on its current utilization by evaluating the factors of $P(u) = P_{Idle} + (P_{Busy} - P_{Idle}) \cdot u$. Every calculator offers a `calculate` method that takes exactly one measurement for every *metricType* of a MeasuredFactor specified in the corresponding PowerModelSpecification instance. FixedFactors do not change and are thus passed as part of the calculator's constructor parameters.

After determining power consumption of all nested PowerConsumingEntities the analyzer aggregates the power consumption of a PowerProvidingEntity. We apply the power model concept to individual elements in the power provisioning infrastructure. This design enables modeling of arbitrary conversion losses [PKA07] for each individual element in the provisioning infrastructure hierarchy.

The following sections briefly sketch how the PCA is used as part of post- and intra-simulation power consumption analysis to support software architects in designing power-efficient software systems.

5.1 Post-Simulation Power Consumption Analysis

Post-simulation power consumption analysis allows software architects to analyze the power consumption of a software system after a previous simulation run. It is possible to analyze the power consumption of any element in the power provisioning infrastructure at any given point in time. Consumption characteristics of the power provisioning infrastructure

can be changed without requiring repeated simulation. Post-simulation analysis leverages measurements that were extracted from the simulation for a set of system metrics, e.g. resource utilization for the linear power model presented in Section 4.3. Multiple power provisioning systems can be compared on the basis of the same simulation run as changes in the power consumption characteristics do not induce changes in the simulation logic. Software architects can configure the analysis, e.g. by setting the size of the interval for utilization-based metrics.

PCA calculates power consumption measurements of a PowerProvidingEntity by aggregating the power consumption of all connected elements. In case of post-simulation analysis the measurements are taken from the Recorder Framework (c.f. Figure 6), which stores all measurements of a simulation run. Power consumption is always calculated at a discrete point in time. Yet, measured data from the experiment does not necessarily contain measurements for said specific point in time. This is resolved by passing a derived calculated value from the post-simulation context to the analyzer.

5.2 Intra-Simulation Power Consumption Analysis

Power-conscious self-adaptation tactics as proposed in [JHJ+10, DSG+12, VAN08, RRT+08] adapt the system based on power consumption measurements. While each of the tactics has been evaluated for a set of specific software systems it is difficult to estimate how they would impact quality characteristics of other systems. Estimating the effect of using multiple energy-conscious self-adaptation tactics concurrently is even more difficult. Furthermore, power-conscious and other self-adaptation tactics can influence each other depending on specific thresholds for activation. By making power consumption measurements available at intra-simulation time software architects are enabled to evaluate combinations of all tactics and reason on QoS implications on an architectural level.

Intra-simulation power consumption analysis uses the same technical infrastructure as post-simulation power consumption analysis. The PCA derives power consumption metrics from measurements collected during the simulation by aggregating the power consumption of individual elements in the Power Infrastructure model instance. The only difference is that the PCA processes constantly updated measurements from a running simulation instead of pre-recorded measurements. Calculators provide power consumption estimates of hardware components to power consumption probes, which in turn provide power consumption measurements as a basis for decisions of power-conscious self-adaptation tactics.

We are currently implementing the presented concepts as an extension of SimuLizar.

6 Conclusion and Outlook

This paper presented a model-driven approach for analyzing the power consumption of static and self-adaptive software systems. By accounting for different power models and temporary peaks our analysis improves the precision of power consumption estimations on

an architectural level when compared to previous work [BWK14, MBAG10].

Our proposed Power Consumption model captures the consumption characteristics of a software system and its power provisioning infrastructure. It is designed to support design-time reasoning on the power consumption of power-conscious software systems. We presented how the three major aspects provisioning infrastructure (Power Infrastructure), consumption characteristics of hardware components (Power Binding) and the power models used to evaluate consumption characteristics of the components (Power Specification) can be abstracted. We showed the advantages of integrating them using a loose coupling concept. It allows software architects to separately define and extend a software system's power provisioning infrastructure and the power models used to predict the consumption of different types of hardware components.

The paper outlines a methodology and implementation for post- and intra-simulation power consumption analysis of software systems. Post-simulation analysis allows software architects to evaluate the power consumption of a software system. Thereby, multiple design alternatives can be compared against each other for their effect on power consumption. Not only is it possible to track peaks in power consumption for individual nodes but also for all other elements in the power provisioning infrastructure, e.g. PDUs. As a significant portion of large-scale software systems' TCO is determined by their power efficiency, this ultimately allows software architects to more accurately reason on the TCO. No repeated simulations are required when power consumption properties of the system-under-analysis are changed. Our approach further supports software architects in selecting a set of energy-conscious self-adaptation tactics that are best suited to meet QoS goals in multiple quality dimensions. We showed the integration of intra-simulation power consumption analysis with the SimuLizar approach and how our extension supports reasoning on energy-conscious self-adaptation tactics.

In future work we will increase the precision of power consumption predictions and enable architecture-level cost projections based on power consumption.

The Power Infrastructure model includes power sources as explicit entities. As of now, constant restrictions on the peak power provided by these PowerProvidingEntities are captured. We plan to introduce temporal constraints on power consumption that are driven by electricity price and availability. We expect a significant improvement of operational cost projections for software systems that are operated in dynamic environments.

Power efficiency of software systems can be controlled through hardware and middleware techniques such as ACPI [Hew13]. ACPI allows to control power efficiency by transitioning resources between states with different power consumption and performance properties. Palladio currently does not distinguish operational states of hardware. By introducing the modeling of operational states to Palladio's hardware resources, the evaluation of techniques such as power capping [RRT+08] will be supported. We expect this to improve accuracy of both power consumption and performance predictions.

We plan to account for reconfigurations to the execution environment of software systems, e.g. turning nodes off and on. This allows focusing on the optimization for whole data centres instead of a set of applications. Their inclusion should contribute further in achieving more precise power consumption predictions for self-adaptive software systems.

Further ideas for future work include an explicit modeling of the power consumption caused by network equipment, as well as cooling and ventilation. Our approach currently does not explicitly account for cooling and ventilation, as both are strongly correlated with power consumption of the computing infrastructure [FWB07]. When modeling a system's total power consumption we thus follow Fan et al.'s proposition to include them into our power models "as a fixed tax over the critical power" [FWB07]. However, the approach presented in this paper allows to include detailed power models for cooling by including them as PowerConsumingEntities. Dependencies between the heat generated by the compute hardware and their power consumption can be represented as MeasuredFactors to consider load variations in a data center. We currently account for network power consumption as a fixed factor since it is reported to be mostly static [FWB07, NPI+08]. Our model can be extended to include network power consumption in a similar way as for cooling and ventilation. The necessary steps comprise the introduction of network equipment as a specialized PowerConsumingEntity and the specification of their consumption properties in the Power Binding and Power Specification model.

7 Acknowledgements

This work is funded by the European Union's Seventh Framework Programme under grant agreement 610711.

References

[BBM13] Matthias Becker, Steffen Becker, and Joachim Meyer. SimuLizar: Design-Time Modelling and Performance Analysis of Self-Adaptive Systems. In *Proceedings of Software Engineering 2013*, SE2013, February 2013.

[BKBR12] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. Architecture-Based Reliability Prediction with the Palladio Component Model. *IEEE Transactions on Software Engineering*, 38(6):1319–1339, Nov 2012.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

[BLB13] Matthias Becker, Markus Luckey, and Steffen Becker. Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time. In *Proceedings of the 9th ACM SigSoft International Conference on Quality of Software Architectures*, QoSA '13. ACM, June 2013.

[BWK14] Andreas Brunnert, Kilian Wischer, and Helmut Krcmar. Using Architecture-level Performance Models As Resource Profiles for Enterprise Applications. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 53–62, New York, NY, USA, 2014. ACM.

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud

Computing Environments and Evaluation of Resource Provisioning Algorithms. *Softw. Pract. Exper.*, 41(1):23–50, January 2011.

[DSG+12] Corentin Dupont, Thomas Schulze, Giovanni Giuliani, Andrey Somov, and Fabien Hermenier. An Energy Aware Framework for Virtual Machine Placement in Cloud Federated Data Centres. In *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*, e-Energy '12, pages 4:1–4:10, New York, NY, USA, 2012. ACM.

[ERKR06] Dimitris Economou, Suzanne Rivoire, Christos Kozyrakis, and Partha Ranganathan. Full-System Power Analysis and Modeling for Server Environments. In *Workshop on Modeling Benchmarking and Simulation (MOBS)*, 2006.

[FWB07] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power Provisioning for a Warehouse-sized Computer. *SIGARCH Computer Architecture News*, 35(2):13–23, June 2007.

[GÏ3] Sebastian Götz. *Multi-Quality Auto-Tuning by Contract Negotiation*. PhD thesis, Technische Universität Dresden, Dresden, Germany, February 2013.

[GHLK+12] B. Grot, D. Hardy, P. Lotfi-Kamran, B. Falsafi, C. Nicopoulos, and Y. Sazeides. Optimizing Data-Center TCO with Scale-Out Processors. *Micro, IEEE*, 32(5):52–63, Sept 2012.

[GHMP08] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, December 2008.

[GWCA12] Sebastian Götz, Claas Wilke, Sebastian Cech, and Uwe Aßmann. *Sustainable ICTs and Management Systems for Green Computing*, chapter Architecture and Mechanisms for Energy Auto Tuning, pages 45–73. IGI Global, June 2012.

[GWR+13] Sebastian Götz, Claas Wilke, Sebastian Richly, Christian Piechnick, Georg Püschel, and Uwe Aßmann. Model-driven Self-optimization Using Integer Linear Programming and Pseudo-Boolean Optimization. In *The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2013)*, pages 55–64. IARIA, 2013.

[Hew13] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation. Advanced Configuration and Power Interface Specification, 2013.

[HP13] Chung-Hsing Hsu and S.W. Poole. Revisiting Server Energy Proportionality. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 834–840, Oct 2013.

[JHJ+10] Gueyoung Jung, M.A Hiltunen, K.R. Joshi, R.D. Schlichting, and C. Pu. Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures. In *2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 62–73, June 2010.

[KBH07] Heiko Koziolek, Steffen Becker, and Jens Happe. Predicting the Performance of Component-Based Software Architectures with Different Usage Profiles. In Sven Overhage, Clemens A. Szyperski, Ralf Reussner, and Judith A. Stafford, editors, *Software Architectures, Components, and Applications*, volume 4880 of *Lecture Notes in Computer Science*, pages 145–163. Springer Berlin Heidelberg, 2007.

[KDH+12] Max E. Kramer, Zoya Durdik, Michael Hauck, Jörg Henss, Martin Küster, Philipp Merkle, and Andreas Rentschler. Extending the Palladio Component Model using Profiles and Stereotypes. In Steffen Becker, Jens Happe, Anne Koziolek, and Ralf Reussner, editors, *Palladio Days 2012 Proceedings (appeared as technical report)*, Karlsruhe Reports in Informatics ; 2012,21, pages 7–15, Karlsruhe, 2012. KIT, Faculty of Informatics.

[Koz08] Heiko Koziolek. *Parameter dependencies for reusable performance specifications of software components*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2008.

[KR08] Heiko Koziolek and Ralf Reussner. A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In *Proceedings of the SPEC International Workshop on Performance Evaluation: Metrics, Models and Benchmarks*, SIPEW '08, pages 58–78, Berlin, Heidelberg, 2008. Springer-Verlag.

[Leh14] Sebastian Lehrig. Quality Analysis Lab (QuAL): Software Design Description and Developer Guide Version 0.2. Technical report, Universität Paderborn, Faculty of Electrical Engineering - Computer Science - Mathematics, April 2014.

[MBAG10] Indika Meedeniya, Barbora Buhnova, Aldeida Aleti, and Lars Grunske. Architecture-Driven Reliability and Energy Optimization for Complex Embedded Systems. In George T. Heineman, Jan Kofron, and Frantisek Plasil, editors, *Research into Practice - Reality and Gaps*, volume 6093 of *Lecture Notes in Computer Science*, pages 52–67. Springer Berlin Heidelberg, 2010.

[Mer08] Ed Merks. Creating Children You Didn't Know Existed. `http://ed-merks.blogspot.de/2008/01/creating-children-you-didnt-know.html`, January 2008. Online; accessed 31/10/2014.

[MH11] Philipp Merkle and Jörg Henss. EventSim – An Event-driven Palladio Software Architecture Simulator. In Steffen Becker, Jens Happe, and Ralf Reussner, editors, *Palladio Days 2011 Proceedings (appeared as technical report)*, Karlsruhe Reports in Informatics ; 2011,32, pages 15–22, Karlsruhe, 2011. KIT, Fakultät für Informatik.

[NPI+08] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. Reducing Network Energy Consumption via Sleeping and Rate-adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 323–336, Berkeley, CA, USA, 2008. USENIX Association.

[PKA07] A. Pratt, P. Kumar, and T.V. Aldridge. Evaluation of 400V DC distribution in telco and data centers to improve energy efficiency. In *Telecommunications Energy Conference, 2007. INTELEC 2007. 29th International*, pages 32–39, Sept 2007.

[PLL14] Giuseppe Procaccianti, Patricia Lago, and Grace A. Lewis. Green Architectural Tactics for the Cloud. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 41–44, April 2014.

[RRK08] Suzanne Rivoire, Parthasarathy Ranganathan, and Christos Kozyrakis. A Comparison of High-level Full-system Power Models. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, pages 3–3, Berkeley, CA, USA, 2008. USENIX Association.

[RRT+08] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. *SIGARCH Comput. Archit. News*, 36(1):48–59, March 2008.

[SEMM08] Chiyoung Seo, G. Edwards, S. Malek, and N. Medvidovic. A Framework for Estimating the Impact of a Distributed Software System's Architectural Style on its Energy Consumption. In *Seventh Working IEEE/IFIP Conference on Software Architecture*, WICSA 2008, pages 277–280, February 2008.

[VAN08] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In Valrie Issarny and Richard Schantz, editors, *Middleware 2008*, volume 5346 of *Lecture Notes in Computer Science*, pages 243–264. Springer Berlin Heidelberg, 2008.

[ZWW12] Yanwei Zhang, Yefu Wang, and Xiaorui Wang. Electricity Bill Capping for Cloud-Scale Data Centers that Impact the Power Markets. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 440–449, Sept 2012.

Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability

Jan Waller, Florian Fittkau, and Wilhelm Hasselbring
Department of Computer Science, Kiel University, Kiel, Germany
{jwa,ffi,wha}@informatik.uni-kiel.de

Abstract: Monitoring of a software system provides insights into its runtime behavior, improving system analysis and comprehension. System-level monitoring approaches focus, e.g., on network monitoring, providing information on externally visible system behavior. Application-level performance monitoring frameworks, such as Kieker or Dapper, allow to observe the internal application behavior, but introduce runtime overhead depending on the number of instrumentation probes.

We report on how we were able to significantly reduce the runtime overhead of the Kieker monitoring framework. For achieving this optimization, we employed micro-benchmarks with a structured performance engineering approach. During optimization, we kept track of the impact on maintainability of the framework. In this paper, we discuss the emerged trade-off between performance and maintainability in this context. To the best of our knowledge, publications on monitoring frameworks provide none or only weak performance evaluations, making comparisons cumbersome. However, our micro-benchmark, presented in this paper, provides a basis for such comparisons.

Our experiment code and data are available as open source software such that interested researchers may repeat or extend our experiments for comparison on other hardware platforms or with other monitoring frameworks.

1 Introduction

Software systems built on and around internal and external services are complex. Their administration, adaptation, and evolution require a thorough understanding of their structure and behavior at runtime. Monitoring is an established technique to gather data on runtime behavior and allows to analyze and visualize internal processes.

System monitoring approaches, such as Magpie [BIMN03] or X-Trace [FPK+07], are minimal invasive and target only network and operating system parameters. Although these approaches have the advantage of minimal performance impact, they are not able to provide a view of internal application behavior.

A solution to these limitations is application level monitoring, as provided by SPASSmeter [ES14], Dapper [SBB+10], or Kieker [vHWH12]. However, application-level monitoring introduces monitoring overhead depending on the number of monitored operations and the efficiency of the monitoring framework. For a detailed view of a software system's internal behavior, monitoring must be all-embracing, causing significant performance im-

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

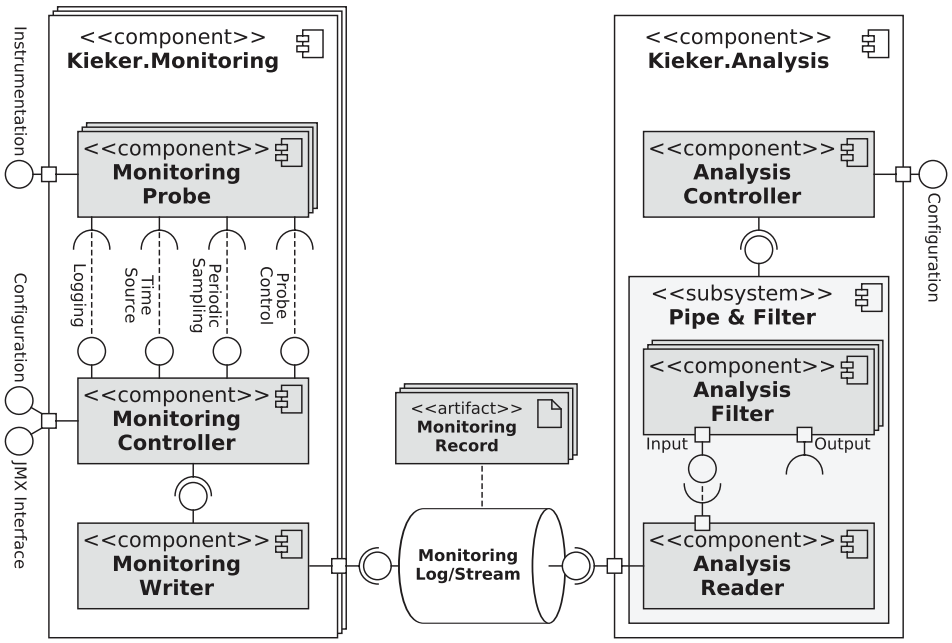


Figure 1: UML component diagram of a top-level view on the Kieker framework architecture

pact. While many monitoring frameworks are claiming to have minimal impact on the performance, these claims are often not backed up with a concise performance evaluation determining the actual cost of monitoring.

Two of our recent projects, ExplorViz [FWWH13] and iObserve [HHJ+13], use monitoring to evaluate distributed cloud systems. They need detailed information on the internal behavior of these systems. Therefore, they rely on a high-throughput monitoring framework which is easy to adapt and maintain. Furthermore, the framework must be able to transport logged data from the observed system components and aggregate it on a remote analysis system.

The Kieker monitoring framework has been developed in our research group over the past years. We evaluated this monitoring framework with a micro-benchmark to assess its capability as high-throughput framework and used a structured performance engineering approach to minimize its overhead while keeping the framework maintainable.

In this paper, we present our monitoring micro-benchmark MooBench and its application in a structured performance engineering approach to reduce the monitoring overhead of the Kieker framework. We report on our exploration of different potential optimization options and our assessment of their impact on the performance as well as the maintainability and usability of the framework. While high-throughput is very important to observe distributed systems, the maintainability trade-off should be minimal. Otherwise the framework may become unusable for a broader audience, effectively rendering the optimization useless.

In summary, our main contributions are:

- a micro-benchmark for monitoring frameworks,
- an example of a structured performance engineering activity for tuning the throughput of monitoring frameworks using a micro-benchmark,
- and an evaluation of the trade-off between performance and maintainability in a high-throughput monitoring framework.

Besides our approach and evaluation, we provide the research community with all sources to repeat and validate our benchmarks and our results in our download area.[1] These downloads include our results in a raw data format, statistical analyses, and generated diagrams.

The rest of the paper is organized as follows. In Section 2, we introduce software quality terms and the Kieker monitoring framework. Our benchmark developed for overhead evaluation of monitoring frameworks is presented in Section 3. Our performance tunings and our evaluations are described in Section 4, Finally, related work is discussed in Section 5, while we draw the conclusions and present future work in Section 6.

2 Foundations

This paper evaluates performance advances realized for the Kieker framework in conjunction with their impact on maintainability. For a better understanding of these two characteristics we provide a brief description of the used software quality attributes (Section 2.1) and an overview of the Kieker framework (Section 2.2) in this section.

2.1 Software Quality

The ISO/IEC 25010 [ISO11] standard, the successor of the well-known ISO/IEC 9126 [ISO01] standard, defines software quality over eight distinct characteristics and 23 subcharacteristics. In this paper we mainly focus on *performance efficiency* and *maintainability* supplemented by *functional completeness*, as these characteristics are used to evaluate different modifications of our monitoring framework.

Performance efficiency, as defined in [ISO11], comprises *time behavior*, *resource utilization*, and *capacity*, as sub-characteristics for the degree of requirement fulfillment. For our evaluation we focus on processing and response time in combination with throughput.

Maintainability is very important for a monitoring framework to be applicable in different software projects. It is characterized by *modularity*, *reusability*, *analyzability*, *modifiability*, and *testability*. However, maintainability, especially if realized by modularization or generalization, can lead to a reduction in performance. Therefore, optimizations for these two characteristics are conflicting requirements.

[1] `http://kieker-monitoring.net/overhead-evaluation/`

2.2 Kieker Monitoring Framework

The Kieker[2] framework [vHWH12, vHRH+09] is an extensible framework for application performance monitoring and dynamic software analysis. The framework includes measurement probes for the instrumentation of software systems and monitoring writers to facilitate the storage or further transport of gathered data. Analysis plug-ins operate on the gathered data, and extract and visualize architectural models, augmented by quantitative observations.

In 2011, the Kieker framework was reviewed, accepted, and published as a recommended tool for quantitative system evaluation and analysis by multiple independent experts of the SPEC RG. Since then, the tool is also distributed as part of SPEC RG's tool repository.[3] Although originally developed as a research tool, Kieker has been evaluated in several industrial systems [vHRH+09].

Kieker Architecture

The Kieker framework provides components for software instrumentation, collection of information, logging of collected data, and analysis of this monitoring data. A top-level view of its architecture is presented in Figure 1. Each Kieker component is extensible and replaceable to support specific project contexts.

The general Kieker architecture is divided into two parts: The Kieker.Monitoring component for monitoring software systems and the Kieker.Analysis component for analysis and visualization of gathered data. These two components are connected by a Monitoring Log or Stream, decoupling the analysis from the monitoring and providing the means to perform the analysis on a separate system resulting in a reduced performance impact on the monitored system.

The focus of this paper is on the Kieker.Monitoring component, see the left side in Figure 1. It is realized by three subcomponents for data collection, monitoring control, and data delivery. Data collection is performed by MonitoringProbes which are technology dependent, as they are integrated into the monitored software system. The MonitoringController plays a central role in the monitoring side of Kieker. To handle record data, it accepts MonitoringRecords from probes and delivers them to the MonitoringWriter. Kieker comes with different MonitoringWriters addressing different needs for monitoring data handling, like logs and streams. Asynchronous MonitoringWriters contain a separate WriterThread to decouple the overhead of writing from the monitored software system. In this case, MonitoringRecords are exchanged and buffered via a configurable (blocking) queue. All interfaces of Kieker.Monitoring are well defined, allowing to add new components or to augment functionality in a transparent way.

[2] http://kieker-monitoring.net
[3] http://research.spec.org/projects/tools.html

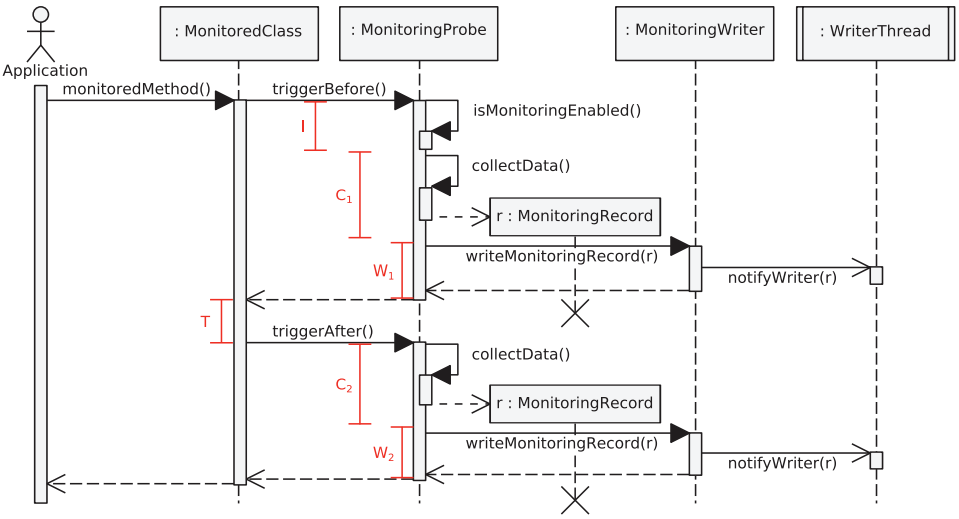


Figure 2: UML sequence diagram for method monitoring with the Kieker framework [WH13]

3 MooBench Benchmark

Benchmarks are used to compare different platforms, tools, or techniques in experiments. They define standardized measurements to provide repeatable, objective, and comparable results. In computer science, benchmarks are used to compare, e. g., the performance of CPU, database management systems, or information retrieval algorithms [SEH03]. In this paper, we use the MooBench benchmark to measure the overhead caused by monitoring a method, to determine relevant performance tuning opportunities, and to finally evaluate the impact of the optimization in the Kieker framework.

In this section, we first introduce the notion of monitoring overhead and provide a partition into three causes of monitoring overhead (Section 3.1). Next, we provide a benchmark to measure these portions of monitoring overhead (Section 3.2).

3.1 Monitoring Overhead

A monitored software system has to share some of its resources with the monitoring framework (e. g., CPU-time or memory), resulting in the *probe effect* [Jai91]. The probe effect is the influence on the behavior of the system by measuring it. This influence includes changes to the probabilities of non-deterministic choices, changes in scheduling, changes in memory consumption, or changes in the timing of measured sections. Here, we take a look at the overhead causing parts of the probe effect.

Monitoring overhead is the amount of additional usage of resources by a monitored execution of a program compared to a normal (not monitored) execution of the program. In

Listing 1: MonitoredClass with monitoredMethod()

```
class MonitoredClass {
  ThreadMXBean threadMXBean = ManagementFactory
    .getThreadMXBean();
  long monitoredMethod(long methodTime, int recDepth) {
    if (recDepth > 1) {
      return this.monitoredMethod(methodTime, recDepth-1);
    } else {
      final long exitTime = this.threadMXBean
        .getCurrentThreadUserTime() + methodTime;
      long currentTime;
      do {
        currentTime = this.threadMXBean
          .getCurrentThreadUserTime();
      } while (currentTime < exitTime);
      return currentTime;
    }
  }
}
```

this case, resource usage encompasses utilization of CPU, memory, I/O systems, and so on, as well as the time spent executing. Monitoring overhead in relation to execution time is the most commonly used definition of overhead. Thus, in the following, any reference to monitoring overhead concerns overhead in time, except when explicitly noted otherwise.

3.1.1 Causes of Monitoring Overhead

In Figure 2, we present a simplified UML sequence diagram representation of the control flow for monitoring a method execution in the Kieker monitoring framework.Although this diagram is tailored to the Kieker framework, other application level monitoring frameworks usually have a similar general behavior. As annotated in red color in the diagram, we propose a separation into three different portions of monitoring overhead while monitoring an application [WH12].

These portions are formed by three common causes of application level monitoring overhead: (1) the instrumentation of the monitored system itself (I), (2) collecting data within the system, e. g., response times or method signatures, (C), and finally (3) either writing the data into a monitoring log or transferring the data to an analysis system (W). These three causes and the normal execution time of a monitored method (T) are further detailed below:

T The actual execution time of the monitoredMethod(), i. e., the time spent executing the actual code of the method if no monitoring is performed, is denoted as T.

In Figure 2 this time is annotated with a red T. Although sequence diagrams provide a general ordering of before and after, the depicted length of an execution carries no meaning. Thus, for reasons of space and clarity, the illustration of the actual execution time T in the figures is small compared to the sum of the three overhead timings. However, note that in actual systems the execution time T is often large compared to the sum of overhead.

I Before the code of the monitoredMethod() in the MonitoredClass is executed, the triggerBefore() part of the MonitoringProbe is executed. Within the probe, isMonitoringEnabled() determines whether monitoring is activated or deactivated for the monitoredMethod(). If monitoring is currently deactivated for the method, no further probe code will be executed and the control flow immediately returns to the monitoredMethod(). Besides these operations of the monitoring framework, I also includes any overhead caused by the used instrumentation. For instance, when performing aspect-oriented instrumentation with AspectJ, similar calls to our triggerBefore() are performed internally.

In Figure 2, I indicates the execution time of the instrumentation of the method including the time required to determine whether monitoring of this method is activated or deactivated.

C If monitoring of the monitoredMethod() is active, the MonitoringProbe will collect some initial data with its collectData() method, such as the current time and the method signature, and create a corresponding MonitoringRecord in memory (duration C_1). After this record is forwarded to the MonitoringWriter, the control flow is returned to the monitoredMethod().

When the execution of the actual code of the monitoredMethod() finished with activated monitoring, the triggerAfter() part of the MonitoringProbe is executed. Again, some additional data, such as the response time or the return values of the method, is collected and another corresponding MonitoringRecord is created in main memory. Finally, this record is forwarded to the MonitoringWriter, too.

In Figure 2, the time needed to collect data of the monitoredMethod() and to create the MonitoringRecords in main memory is $C = C_1 + C_2$.

W Each created and filled MonitoringRecord r is forwarded to a MonitoringWriter with the method writeMonitoringData(r). The MonitoringWriter in turn stores the collected data in an internal buffer, that is processed asynchronously by the WriterThread into the Monitoring Log/Stream.

Depending on the underlying hardware and software infrastructure and the available resources, the actual writing within this additional thread might have more or less influence on the results. For instance, in cases where records are collected faster than they are written, the internal buffer reaches its maximum capacity and the asynchronous thread becomes effectively synchronized with the rest of the monitoring framework. Thus, its execution time is added to the caused runtime overhead of W. In other cases, with sufficient resources available, the additional overhead of the writer might be barely noticeable [WH12].

Listing 2: Benchmark thread calling monitoredMethod()

```
MonitoredClass mc; //initialized before
long start_ns, stop_ns;
for (int i = 0; i < totalCalls; i++) {
  start_ns = System.nanoTime();
  mc.monitoredMethod(methodTime, recDepth);
  stop_ns = System.nanoTime();
  timings[i] = stop_ns - start_ns;
}
```

In Figure 2, $W = W_1 + W_2$ is the amount of overhead caused by placing the monitoring data in an exchange buffer between the MonitoringWriter and the WriterThread as well as possibly the time of actually writing the collected monitoring data into a monitoring log or into a monitoring stream.

3.1.2 Measures of Monitoring Overhead

In this paper, we are focussed on improving the monitoring *throughput*, i. e., the number of MonitoringRecords sent and received per second, instead of the flat monitoring cost imposed per MonitoringRecord, i. e., the actual change in a monitored method's *response time*. This response time and the monitoring throughput are related: improving one measure usually also improves the other one. However, with asynchronous monitoring writers (as in the case of Kieker and our experiments) the relationship between throughput and response time can become less obvious.

In order to measure the maximal monitoring throughput, it is sufficient to minimize T while repeatedly calling the monitoredMethod(). Thus MonitoringRecords are produced and written as fast as possible, resulting in maximal throughput. As long as the actual WriterThread is capable of receiving and writing the records as fast as they are produced (see description of W above), it has no additional influence on the monitored method's response time. When our experiments reach the WriterThread's capacity, the buffer used to exchange records between the MonitoringWriter and the WriterThread blocks, resulting in an increase of the monitored method's *response time*.

3.2 Our Monitoring Overhead Benchmark

The MooBench micro-benchmark has been developed to quantify the three portions of monitoring overhead under controlled and repeatable conditions. It is provided as open source software.[4] Although the benchmark and our experiments are originally designed for the Kieker framework, they can be adapted to other monitoring frameworks by exchanging the used monitoring component and its configuration.

[4] http://kieker-monitoring.net/MooBench

Figure 3: Benchmark engineering phases [WH13]

In order to achieve representative and repeatable performance statistics for a contemporary software system, benchmarks have to eliminate random deviations. For instance, software systems running on managed runtime systems, such as the Java VM (JVM), are hard to evaluate because of additional parameters influencing the performance, such as class loading, just-in-time compilation (JIT), or garbage collection [GBE07]. Therefore, a benchmark engineering process with guidelines to produce good benchmarks is required.

Our benchmark engineering process is partitioned into three phases [WH13] (see Figure 3). For each phase, a good benchmark should adhere to several common guidelines. For instance, it should be designed and implemented to be *representative* and *repeatable* [Gra93, Kou05, Hup09]. Similar guidelines should be followed during the execution and analysis/presentation of the benchmark [GBE07], i. e., using *multiple executions*, a *sufficient warm-up period*, and an *idle environment*. Furthermore, a rigorous *statistical analysis* of the benchmark results and a *comprehensive reporting* of the experimental setup are required in the analysis and presentation phase. Refer to [WH13] for an overview on our employed guidelines.

In the following, we detail the three benchmark engineering phases of our micro-benchmark for monitoring overhead. First, we describe our design and implementation decisions to facilitate representativeness and repeatability. Next, we give guidelines on the execution phase of our benchmark, focussing on reaching a steady state. Finally, we describe the analyses performed by our benchmark in the analysis & presentation phase.

3.2.1 Design & Implementation Phase

The architecture of our benchmark setup is shown in Figure 4. It consists of the Benchmark System running in a JVM, and an External Controller initializing and operating the system. The Benchmark System is divided into two parts: First, the Monitored Application, consisting of the Application instrumented by the Monitoring component. Second, the Benchmark, consisting of the Benchmark Driver with one or more active Benchmark Threads accessing the Monitored Application. For benchmarking the Kieker framework, the Monitoring is realized by the Kieker.Monitoring component (see Figure 1). For benchmarking of other monitoring frameworks, this component would be replaced accordingly.

For our micro-benchmark, the Monitored Application is a basic application core, consisting of a single MonitoredClass with a single monitoredMethod() (see Listing 1). This method has a fixed execution time, specified by the parameter methodTime, and can simulate recDepth nested method calls (forming one *trace*) within this allocated execution time. During the execution of this method, busy waiting is performed, thus fully utiliz-

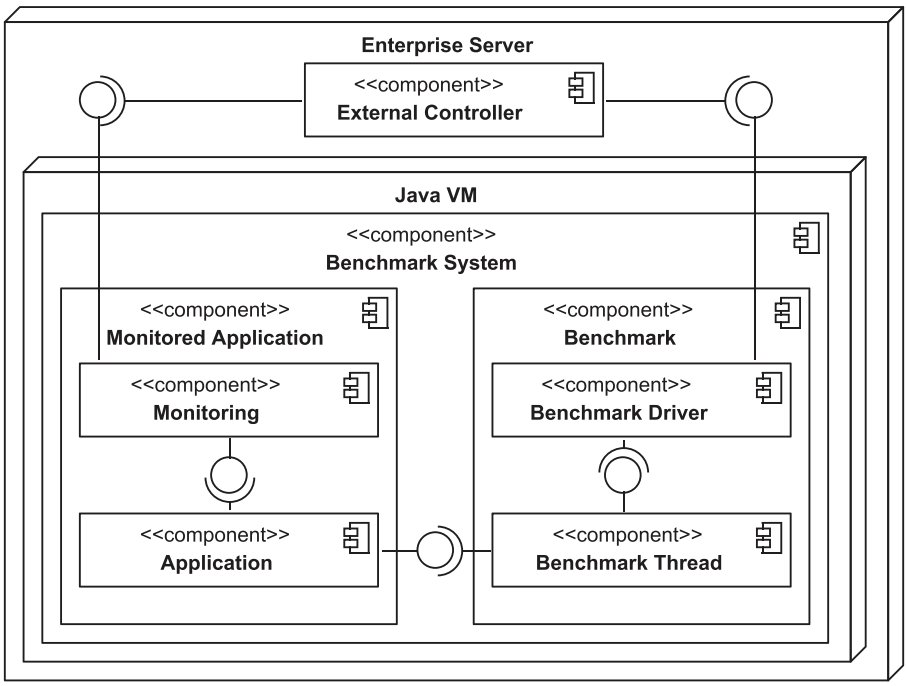


Figure 4: Architecture of the benchmark setup

ing the executing processor core. The loop of the method cannot be eliminated by JIT compiler optimizations, thus avoiding common pitfalls in benchmark systems. In order to correctly simulate a method using the CPU for a period of time despite the activities of other threads in the system, we use getCurrentThreadUserTime() of JMX's ThreadMXBean. The operating system and the underlying hardware of the Benchmark System have to provide a sufficient accuracy of the method in order to get stable and repeatable results. In the case of single threaded benchmarks on an otherwise unoccupied system we could use calls of System.nanoTime() or System.currentTimeMillis() instead.

The Benchmark Driver initializes the Benchmark System, then starts the required number of Benchmark Threads, and collects and persists the recorded performance data.

One or more concurrently executing Benchmark Threads call the monitoredMethod() while recording its response time with calls to System.nanoTime() (see Listing 2). Each thread is parameterized with a *total number of calls*, as well as the *method time* and the *recursion depth* of each call. The total number of calls has to be sufficiently large to include the warm-up period and a sufficient portion of the steady state. Execution time and recursion depth can be utilized to control the number of method calls the monitoring framework will monitor per second.

The External Controller calls the Monitored Application with the desired parameters and ensures that the Monitoring component is correctly initialized and integrated into the Monitored Application.

Each experiment consists of four independent runs, started by the external controller on a fresh JVM invocation. Each individual portion of the execution time is measured by one run (see T, I, C, and W in Figure 2). This way, we can incrementally measure the different portions of monitoring overhead as introduced in the previous Section 3.1. For instance, we can use this information to guide our optimizations.

1. In the first run, only the execution time of the chain of recursive calls to the monitoredMethod() is determined (T).
2. In the second run, the monitoredMethod() is instrumented with a Monitoring Probe, that is deactivated for the monitoredMethod(). Thus, the duration $T + I$ is measured.
3. The third run adds the data collection with an activated Monitoring Probe without writing any collected data ($T + I + C$).
4. The fourth run finally represents the measurement of full monitoring with the addition of an active Monitoring Writer and an active Writer Thread ($T + I + C + W$).

In summary, this configurable benchmark design allows for repeatable measurements of monitoring overhead, that are representative for simple traces.

3.2.2 Execution Phase

The actual benchmark execution is controlled by the provided External Controller. Each independent experiment run to determine a portion of overhead can be repeated multiple times on identically configured JVM instances to minimize the influence of different JIT compilation paths. Furthermore, the number of method executions can be configured to ensure steady state.

In order to determine the steady state of experiment runs, the benchmark user can analyze the resulting data stream as a time series of averaged measured timings. Such a typical time series diagram for experiments with Kieker is presented in Figure 5. To visualize the warm-up phase and the steady state, invocations are bundled into a total of 1,000 bins. The benchmark calculates the mean values of each bin and uses the resulting values to generate the time series diagram.

Our experiments as well as our analyses of JIT compilation and garbage collection logs of benchmark runs with the Kieker framework on our test platform suggest discarding the first half of the executions to ensure a steady state in all cases (the grey-shaded part of Figure 5 illustrates this). We propose similar analyses with other monitoring frameworks, configurations, or hard- and software platforms to determine their respective steady states.

Furthermore, the Benchmark Driver enforces the garbage collection to run at least once at the beginning of the warm-up phase.Our experiments suggest that this initial garbage collection reduces the time until a steady state is reached. The regular spikes in the measured execution times, seen in Figure 5, correspond to additional garbage collections.

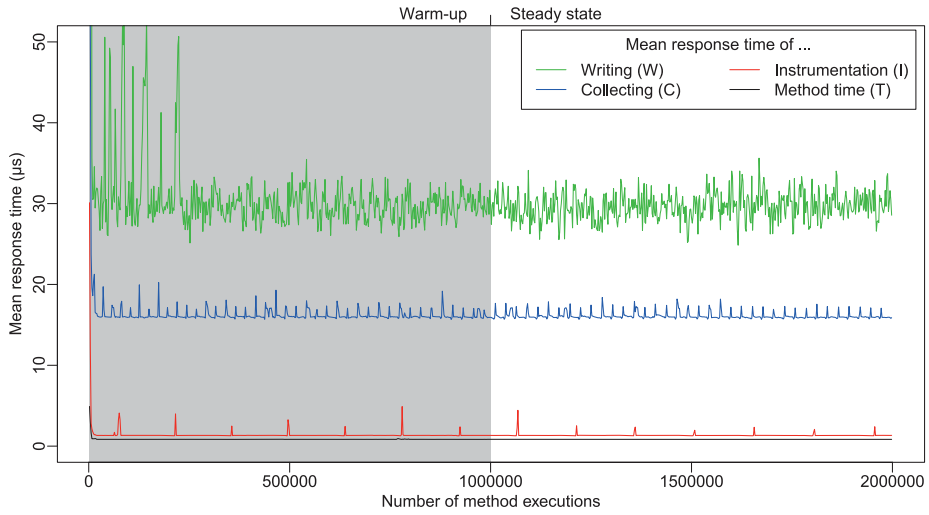


Figure 5: Time series diagram of measured timings

Finally, the benchmark user should ensure that the hard- and software environment, used to execute the experiments, is held idle during the experiments. Thus, there should be no perturbation by the execution of background tasks.

3.2.3 Analysis & Presentation Phase

In accordance with Georges et al. [GBE07], in the statistical analysis of the results of the benchmark runs, our benchmark provides the mean and median values of the measured timings across all runs instead of reporting only best or worst runs. In addition, it includes the lower and upper quartile, as well as the 95% confidence interval of the mean value.

Of note is the calculation of the monitoring throughput within the benchmark. As mentioned in the benchmark design phase, our benchmark collects the response times of the MonitoredMethod(). These response times measurements are collected in a number of bins, each containing one second worth of method executions. The number of response times per bin corresponds to the reported throughput of method executions per second.

To facilitate repetitions and verifications of our experiments, the benchmark user has to provide a detailed description of the used configurations and environments.

3.3 Evaluations of MooBench

Our proposed micro-benchmark has already been evaluated with the Kieker framework in multiple scenarios. In [vHRH+09], we performed initial single-threaded measurements of the Kieker framework and demonstrated the linear scalability of monitoring overhead with

increasing recursion depths. In [WH12], we compared the monitoring overhead of several multi-threaded scenarios, of different writers, and of different hardware architectures with each other. In [vHWH12], we complemented additional results of our micro-benchmark by measurements of the SPECjEnterprise®2010 macro-benchmark. A detailed comparison of several different releases of Kieker as well as of different monitoring techniques and writers has been performed in [WH13]. Finally, in [FWBH13], we extended the benchmark to measure the additional overhead introduced by an online analysis of monitoring data concurrent to its gathering.

In order to broaden our evaluation basis, we intend to perform similar measurements with further available monitoring framework. Furthermore, we plan to validate our results by performing similar measurements with additional, more extensive benchmark suites and macro-benchmarks. Finally, it is of interest to compare further hard- and software configurations, e. g., different heap sizes, JVMs, or platforms as well as, e. g., different instrumentation techniques.

4 Overhead Reduction and its Impact on Maintainability

This section provides an evaluation of the monitoring overhead of Kieker with the MooBench micro-benchmark. The benchmark is used to measure the three individual portions of monitoring overhead. The results of the benchmark are then used to guide our performance tunings of Kieker. The tuned version is again evaluated and compared to the previous one with the help of our benchmark. Thus, we provide an example how micro-benchmarks can be used to steer a structured performance engineering approach.

In the rest of the section, we first provide a description of our experimental setup (Section 4.1) to enable repeatability and verifiability for our experiments. Next, we describe our base evaluation of the Kieker framework without any tunings (Section 4.2). The next four sections (4.3 –4.6) describe our incremental performance tunings (PT). Finally, we discuss threats to the validity of our conducted experiments (Section 4.7).

4.1 Experimental Setup

Our benchmarks are executed on the Java reference implementation by Oracle, specifically an Oracle Java 64-bit Server VM in version 1.7.0_25 running on an X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM with Solaris 10 and up to 4 GiB of available heap space for the Java-VM.

In our experiments, we use modified versions of Kieker 1.8 as the monitoring framework under test. All modifications are available in the public Kieker Git repository with tags starting with 1.8-pt-. Furthermore, access to these modifications and to the prepared experimental configurations and finally to all results of our experiments are available online.[5]

[5] http://kieker-monitoring.net/overhead-evaluation/

Table 1: Throughput for basis (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 176.5k	757.6k	63.2k	16.6k
95% CI	± 25.9k	± 5.5k	± 0.1k	± 0.02k
Q_1	1 189.2k	756.6k	63.0k	16.2k
Median	1 191.2k	765.9k	63.6k	16.8k
Q_3	1 194.6k	769.8k	63.9k	17.2k

AspectJ release 1.7.3 with load-time weaving is used to insert the particular MonitoringProbes into the Java bytecode. Kieker is configured to use a blocking queue with 10,000 entries to synchronize the communication between the MonitoringWriter and the WriterThread. The employed TCP writer uses an additional buffer of 64 KiB to reduce network accesses. Furthermore, Kieker is configured to use event records from the kieker.common.record.flow package and the respective probes.

We use a single benchmark thread and repeat the experiments on ten identically configured JVM instances with a sleep time of 30 seconds between all executions. In all experiments using a disk writer, we call the monitoredMethod() 20,000,000 times on each run with a configured methodTime of 0 µs and a stack depth of ten. We discard the first half of the measured executions as warm-up and use the second half as steady state executions to determine our results.

A total of 21 records are produced and written per method execution: a single TraceMetaData record, containing general information about the trace, e. g., the thread id or the host name, and ten BeforeOperationEvent and AfterOperationEvent records each, containing information on the monitored method, e. g., time stamps and operation signatures. This set of records is named a *trace*.

We perform our benchmarks under controlled conditions on a system exclusively used for the experiments. Aside from this, the server machine is held idle and is not utilized.

To summarize our experimental setup according to the taxonomy provided by Georges et al. [GBE07], it can be classified as using multiple JVM invocations with multiple benchmark iterations, excluding JIT compilation time and trying to ensure that all methods are JIT-compiled before measurement, running on a single hardware platform with a single heap size and on a single JVM implementation.

However, the benchmark can be adapted to other scenarios, such as using replay compilation [GEB08] to avoid JIT compiler influence, a comparison of different JVMs [EGDB03], varying heap sizes [BGH+06], or different hardware combinations.

4.2 Base Evaluation

In this section, we present the results of our base evaluation of Kieker. We use the Kieker 1.8 code without the performance tunings mentioned in the following sections. The results from this base evaluation are used to form a baseline for our tuning experiments.

Table 2: Throughput for PT1 (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 190.5k	746.3k	78.2k	31.6k
95% CI	± 4.1k	± 4.1k	± 0.1k	± 0.1k
Q_1	1 191.0k	728.1k	78.3k	28.1k
Median	1 194.1k	756.6k	78.5k	32.5k
Q_3	1 195.1k	763.7k	78.7k	34.7k

The overhead of writing monitoring data (W) is measured in the fourth experiment run of each experiment. Writing requires the transport of the produced MonitoringRecords out of the probe into a remote site via a network connection.

We use the TCP writer, intended for online analysis of monitoring data, i. e., the MonitoringRecords are transported to a remote system, e. g., a storage cloud, to be analyzed while the monitored system is still running. In the case of our experiments, we used the local loopback device for communication, to avoid further perturbation and capacity limits by the local network. Per method execution 848 bytes are transmitted.

Experimental Results & Discussion

The results of our base evaluation are presented in Table 1 and the response times for the four partitions of monitoring overhead of the TCP writer are shown in Figure 6.

For the uninstrumented benchmark system (first experiment run to measure the method time (T)), we measured an average of 1,176.5 k traces per second. Adding deactivated Kieker probes (second experiment run to measure ($T+I$)) resulted in an average of 757.6 k traces per second.

Activating the probes and collecting the monitoring records without writing them (third experiment run to measure ($T + I + C$)) further reduced the average throughput to 63.2 k traces per second. The fourth experiment run with the addition of an active monitoring writer (measuring ($T+I+C+W$)) exceeds the writer thread's capacity (see Section 3.1.2), thus causing blocking behavior.

The TCP writer for online analysis produces an average of 31.6 k traces per second. Furthermore, the 95% confidence interval and the quartiles suggest very stable results, caused by the static stream of written data.

4.3 PT1: Caching & Cloning

As is evident by our analysis of the base evaluation, i. e., by the response times presented in Figure 6 and by the throughputs in Table 1, the main causes of monitoring overhead are the collection of data (C) and the act of actually writing the gathered data (W). Thus, we first focus on general performance tunings in these areas. We identified four possible performance improvements:

First, our preliminary tests showed that certain Java reflection API calls, like constructor and field lookup, are very expensive. These calls are used by Kieker to provide an extensible framework. Instead of performing these lookups on every access, the results can be cashed in HashMaps.

Second, the signatures of operations are stored in a specific String format. Instead of creating this signature upon each request, the resulting String can be stored and reused.

Third, a common advice when developing a framework is not to expose internal data structures, such as arrays. Instead of directly accessing the array, users of the framework should only be able to access cloned data structures to prevent the risk of accidentally modifying internal data. However, in most cases internal data is only read from a calling component and not modified. For all these cases, copying data structures is only a costly effort without any benefit. We omit this additional step of cloning internal data structures and simply provide a hint in the documentation.

Finally, some internal static fields of classes were marked private and accessed by reflection through a SecurityManager to circumvent this protection. These fields were changed to be public to avoid these problems when accessing the fields.

Experimental Results & Discussion

The resulting throughput is visualized in Table 2 and the response time is shown in Figure 6. As can be expected, the changes in the uninstrumented benchmark and with deactivated probes are not significant. However, when adding data collection (C) we measured an increase of 15 k additional traces per second compared to our previous experiments. Finally, the TCP writer's throughput almost doubled while still providing very stable measurements (small confidence interval).

The improvement discussed in this section will be used in Kieker because of their minimal influence on the maintainability and the great improvements in the area of performance efficiency.

4.4 PT2: Inter-Thread Communication

From PT1 we conclude that the queue is saturated and the monitoring thread waits for a free space in the queue, i. e., the writer thread to finish its work. Otherwise, the monitoring thread should be able to pass the records into the queue and proceed with the method. Therefore, our target is to decrease the writing response time. In this step, we want to achieve this goal by optimizing the communication between monitoring and writer thread.

Internal communication is presently modeled with the Java ArrayBlockingQueue class. It is used to pass monitoring records from the monitoring thread to the writer thread. To improve the inter-thread communication performance, we use the disruptor framework,[6] which provides an efficient implementation for inter-thread communication. It

[6] http://lmax-exchange.github.io/disruptor/

Table 3: Throughput for PT2 (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 190.5k	757.6k	78.2k	56.0k
95% CI	± 3.6k	± 6.2k	± 0.1k	± 0.2k
Q_1	1 190.5k	760.0k	78.1k	52.3k
Median	1 191.6k	766.8k	78.4k	53.9k
Q_3	1 194.2k	771.4k	78.7k	61.0k

utilizes a ringbuffer to provide a higher throughput than, for instance, the Java ArrayBlockingQueue class.

The evaluation uses Kieker 1.8 with the designated disruptor ringbuffer. Again, we measure the record throughput per second and the average response time.

Experimental Results & Discussion

The resulting throughput is visualized in Table 3 and the response time is shown in Figure 6. Again, the no instrumentation and deactivated run is roughly the same from the previous experiments. Since we did not improve the collecting phase of Kieker, the response time and throughput for this part is approximately the same as in PT1. The writing response time decreased from 16.35 µs to 6.18 µs which is a decrease of about 62%. The decrease in the response time is accompanied with an increase in the average throughput rate from 31.6k to 56.0k traces per second (77%).

Therefore, our goal of decreasing the writing response time is achieved. The disruptor framework speeds up the transfer into the buffer and also the reading from the buffer resulting in the decreasing of the response time in the writing phase. However, the response time of the writing phase still suggests that the monitoring thread is waiting for the buffer to get an empty space for record passing.

The maintainability of Kieker is unharmed by this optimization because the disruptor framework can be abstracted to provide a single put method into the buffer and the writer components can be easily rewritten to the designated observer pattern of the disruptor framework. It even improves the maintainability of the code since the thread management and message passing is conducted by the disruptor framework. Therefore, this improvement will be used in Kieker.

4.5 PT3: Flat Record Model

As discussed in PT2, the monitoring thread is waiting for the writer thread to finish its work. Therefore, we want to further decrease the writing response time in this optimization. In contrast to PT2, we aim for reducing the work which has to be conducted by the writer thread.

Table 4: Throughput for PT3 (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 176.5k	729.9k	115.7k	113.2k
95% CI	± 2.1k	± 4.4k	± 0.2k	± 0.5k
Q_1	1 186.0k	726.5k	115.8k	113.1k
Median	1 187.1k	734.5k	116.2k	114.3k
Q_3	1 189.2k	739.7k	116.5k	115.0k

The main work, which has to be conducted by the TCP writer, is the serialization of incoming MonitoringRecord objects into a byte representation and writing this byte representation into a ByteBuffer. The root of the object serialization challenge is the creation of the MonitoringRecords which are only created to pass them to the writer thread which serializes them. No calculation is conducted with those records. Therefore, we can also write the required monitoring information directly into a ByteBuffer and pass it into the writer thread. This optimization imposes several performance advantages. First, the object creation and thus garbage collection for those objects is avoided. Furthermore, the ByteBuffers passed into the disruptor framework are fewer objects than passing thousands of MonitoringRecord objects. Since less work has to be conducted by the disruptor framework, the inter-thread communication should be even faster.

For the evaluation we use Kieker 1.8 with the enhancement of directly writing the monitoring information into a ByteBuffer. Again, we measure the record throughput per second and the average response time.

Experimental Results & Discussion

The resulting throughput is shown in Table 4 and the response time is visualized in Figure 6. The no instrumentation and deactivated phase are the same as in the previous experiments. In the collecting phase, the response time decreased from 11.44 µs to 7.23 µs and the throughput increased from 78.2k to 115.7k traces per second. The writing response time decreased from 6.18 µs to 0.2 µs. The average throughput for the writing phase increased from 56.0k to 113.2k traces per second.

The decrease in the writing response time is rooted in the reduction of the work of the writer thread. This work is reduced to sending the ByteBuffer to the network interface and therefore the ringbuffer does not fill up most of the time. The remaining 0.2 µs overhead for the writing phase is mainly found in the putting of the ByteBuffer into the ringbuffer. The collecting phase also became more efficient. The response time decreased since no MonitoringRecord objects are created and therefore less garbage collection takes place.

This improvement will not be used in Kieker because it is harder for the framework user to add own MonitoringRecord types with this optimization. If she wants to add a MonitoringRecord type, she would be forced to write directly into the ByteBuffer instead of just using object-oriented methods. Thus, the optimization would hinder the modularity and reusability of the code.

Table 5: Throughput for PT4 (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 190.5k	763.3k	145.1k	141.2k
95% CI	± 2.0k	± 4.0k	± 0.2k	± 0.3k
Q_1	1 187.4k	747.0k	144.2k	139.4k
Median	1 191.4k	762.5k	146.1k	142.7k
Q_3	1 195.2k	778.4k	146.8k	144.2k

4.6 PT4: Minimal Monitoring Code

In PT3, we optimized the writing phase such that the monitoring thread does not need to wait for the writer thread. Now, about 80% of the time consumed for monitoring the software execution is spent in the collecting phase. Therefore, we try to optimize this phase. As a further optimization, we deleted anything not directly related to pure monitoring. For instance, we deleted interface definitions, consistence checks, and configurability. Furthermore, we provide only five hard coded types of MonitoringRecords.

For the evaluation we use a newly created project, which only includes minimal code for the monitoring. Again, we measure the record throughput per second and the average response time.

Experimental Results & Discussion

The resulting response time is visualized in Figure 6 and the throughput is shown in Table 5. The response time of the no instrumentation, deactivated, and writing phase are roughly equal to the previous experiments. In the collecting phase, the response time decreased and the throughput increased from 115.7k to 145.1k traces per second.

We attribute the decrease in the response time in the collecting phase mainly to the hard coding of the monitoring since less generic lookups must be made. Furthermore, the consistence checks had also an impact.

This improvement will not be used in Kieker because the monitoring tool now lacks important features for a framework, e.g., configurability and reusability.

4.7 Threats to Validity

In the experiments at least one core was available for the monitoring. If all cores were busy with the monitored application, the results could be different. We investigated this circumstance in [WH12].

Further threats involve our benchmark in general. Common threats to validity of micro-benchmarks are their relevance and systematic errors. Our benchmark bases on repeatedly calling a single method. However, by performing recursive calls, the benchmark is able

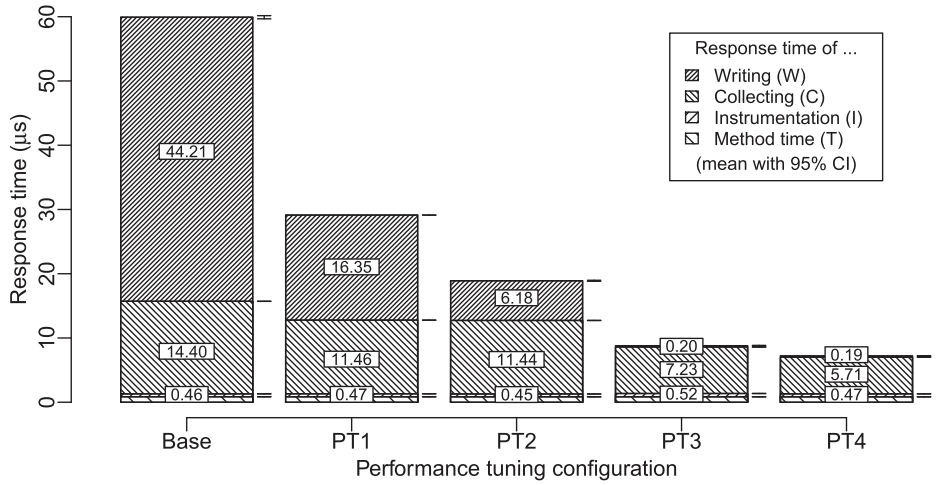


Figure 6: Overview of the tuning results in response time

to simulate larger call stacks. Additional comparisons of our results with more complex benchmarks or applications are required for validation. If the configured method execution time is negligible compared to the overhead, these comparison may be inappropriate. However, our experiments with different method execution times suggest that the results are still valid.

Further threats are inherent to Java benchmarks. For instance, different memory layouts of programs or JIT compilation paths for each execution might influence the results. This is countered by multiple executions of our benchmark. However, the different compilation paths of our four measurement runs to determine the individual portions of monitoring overhead might threaten the validity of our results. This has to be countered by further validation of our benchmark. All benchmark runs include multiple garbage collections which might influence the results. However, this is also true for long running systems that are typically monitored.

5 Related Work

Monitoring is an important part of administration and operation software systems, especially in distributed systems. It allows to determine online or offline bottlenecks or other technical problems in a deployed software system or its neighboring systems which affect the operation. To provide monitoring capabilities, approaches have targeted different levels in the system. Early approaches monitor systems through network communication and services. X-Trace [BIMN03], for example, aggregates such monitoring information and builds system traces on that data. It is minimal invasive for the application, but cannot provide detailed information on machine or application internal behavior. Magpie [FPK+07]

is a framework for distributed performance monitoring and debugging which augments network information with host level monitoring of kernel activities, for example. Both approaches do not provide detailed information on their overhead, but data collection is provided by network infrastructure and the operating system. Therefore only the logging affects the overall system capacity. However, none of these approaches provided a detailed performance impact analyses.

Dapper [SBB+10] and SPASS-meter [ES14] provide, like Kieker, application level monitoring providing insights into internal behavior of applications and its components. Causing a significant impact on the overall performance of those software systems. To reduce the performance impact, Dapper uses sampling, which ignores traces during monitoring. The sampling mechanism of Dapper observes the system load and changes the sampling rate to limit the performance impact.

On systems, like search engines, the jitter of sampling may not have a big impact, as Sigelman et. al. [SBB+10] claim for their applications at Google. In such scenarios sampling can be an approach to balance the monitoring performance trade-off. However, in many scenarios, e.g., failure detection or performance forecasting, detailed monitoring data is necessary to produce a comprehensive view of the monitored system. Therefore, sampling, the reduction of probes, or limiting the scope of the observation to network or host properties is not always an option to limit performance overhead of monitoring. Therefore, we addressed the performance impact of the Kieker framework and minimized it.

Magpie provides a comparison of the impact of different observation methods in their scenario, but no detailed analysis of the overhead. Dapper and X-Trace address the issue of overhead as a characteristic of monitoring, which has to be considered when monitoring. But to the best of our knowledge, no other monitoring framework has been evaluated with a specialized monitoring benchmark targeting the overhead of monitoring itself.

6 Conclusions and Outlook

The paper presents our proposed micro-benchmark for monitoring frameworks. Furthermore, we introduce a structured performance engineering activity for tuning the throughput of the monitoring framework Kieker. The evaluation was conducted by utilizing our monitoring overhead benchmark MooBench. It demonstrates that high-throughput can be combined with maintainability to a certain degree. Furthermore, it shows that our TCP writer is the fastest writer and thus applicable for online analysis, which is required for our upcoming ExplorViz [FWWH13] project.

Our performance tuning shows an upper limit for the monitoring overhead in Kieker. However, in productive environments, monitoring probes are usually configured in a way to stay below the system's capacity. Furthermore, the execution time of most methods (T) is larger than the measured 0.85 µs. Refer to our previous publications for measurement of the lower limit of monitoring overhead [WH12, vHWH12, vHRH+09]. Note, that these previous measurements usually employed a stack depth of one compared to our used stack depth of ten.

As future work, we intend to reduce the impact of deactivated probes by utilizing other instrumentation frameworks than AspectJ, for instance, DiSL [MZA+12]. Furthermore, we will evaluate whether a generator can handle the monitoring record byte serialization and thus PT3 might become applicable in Kieker without loosing maintainability and usability for the framework users. In addition, we plan to measure Kieker with multi-threaded versions of our monitoring benchmark and test further configurations. We also plan to compare the benchmark results for Kieker to the resulting benchmark scores of other monitoring frameworks.

References

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 169–190, 2006.

[BIMN03] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HotOS)*, pages 85–90, 2003.

[EGDB03] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of 18th Conference on OO Programming Systems Languages and Applications (OOPSLA)*, pages 169–186, 2003.

[ES14] Holger Eichelberger and Klaus Schmid. Flexible Resource Monitoring of Java Programs. *Journal of Systems and Software (JSS-9296)*, pages 1–24, 2014.

[FPK+07] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked systems design & implementation (NSDI)*, pages 271–284, 2007.

[FWBH13] Florian Fittkau, Jan Waller, Peer Christoph Brauer, and Wilhelm Hasselbring. Scalable and Live Trace Processing with Kieker Utilizing Cloud Computing. In *Proceedings of the Syposium on Software Performance: Joint Kieker/Palladio Days (KPDays)*, pages 89–98, 2013.

[FWWH13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT)*, 2013.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. of 22nd Conf. on Object-oriented programming systems and applications (OOPSLA)*, pages 57–76, 2007.

[GEB08] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 367–384, 2008.

[Gra93] Jim Gray, editor. *The Benchmark Handbook: For Database and Transaction Systems*. Morgan Kaufmann, 2nd edition, 1993.

[HHJ+13] Wilhelm Hasselbring, Robert Heinrich, Reiner Jung, Andreas Metzger, Klaus Pohl, Ralf Reussner, and Eric Schmieders. iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems. Technical Report 1309, Kiel University, 2013.

[Hup09] Karl Huppler. The Art of Building a Good Benchmark. In *First TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC)*, pages 18–30, 2009.

[ISO01] Software Engineering – Product Quality – Part 1: Quality Model, 2001.

[ISO11] Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[Kou05] Samuel Kounev. *Performance Engineering of Distributed Component-Based Systems – Benchmarking, Modeling and Performance Prediction*. PhD thesis, TU Darmstadt, Germany, 2005.

[MZA+12] Lukas Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Peter Tuma, and Zhengwei Qi. Java Bytecode Instrumentation Made Easy: The DiSL Framework for Dynamic Program Analysis. In *Proceedings of the 10th Asian Syposium on Programming Languages and Systems (APLAS)*, pages 256–263, 2012.

[SBB+10] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[SEH03] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 74–83, 2003.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical Report 921, Kiel University, Germany, 2009.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 247–248, 2012.

[WH12] Jan Waller and Wilhelm Hasselbring. A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring. In *Multicore Software Engineering, Performance, and Tools (MSEPT)*, pages 42–53, 2012.

[WH13] Jan Waller and Wilhelm Hasselbring. A Benchmark Engineering Methodology to Measure the Overhead of Application-Level Monitoring. In *Proceedings of the Syposium on Software Performance: Joint Kieker/Palladio Days (KPDays)*, pages 59–68, 2013.

Toward a Generic and Concurrency-Aware Pipes & Filters Framework

Christian Wulf, Nils Christian Ehmke, and Wilhelm Hasselbring

Software Engineering Group
Kiel University
24098 Kiel, Germany
{chw,nie,wha}@informatik.uni-kiel.de

Abstract: The Pipes-and-Filters design pattern is a well-known pattern to organize and execute components with sequential dependencies. The pattern is therefore often used to perform several tasks consecutively on large data streams, e.g., during image processing or dynamic analyses. In contrast to the pattern's familiarity and application, almost each common programming language lacks of flexible, feature-rich, fast, and concurrency-aware Pipes-and-Filters frameworks. So far, it is common practice that most developers write their own implementation tailored to their specific use cases and demands hampering any effective re-use.

In this paper, we discuss Pipes-and-Filters architectures of several Java-based applications and point out their drawbacks concerning their applicability and efficiency. Moreover, we propose a generic and concurrency-aware Pipes-and-Filters framework and provide a reference implementation for Java called TeeTime.

1 Introduction

Pipes-and-Filters is a common architectural pattern in several projects and applications, often used to process large data streams. Figure 1 shows an example Pipes-and-Filters-oriented processing to visualize program traces, which are reconstructed from serialized method events. Each filter component, also called *stage*, fetches incoming elements from its input ports, processes them, and finally sends the resulting elements via its output ports.

During the development of Kieker [RvHM+08, vHRH+09, vHWH12], an application performance monitoring and architecture discovery framework, we encounter various archi-

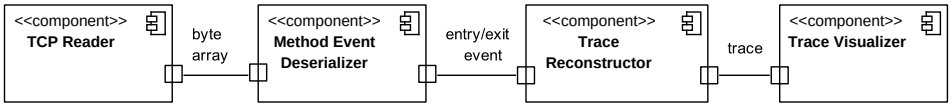


Figure 1: An example Pipes-and-Filters-oriented processing pipeline to visualize program traces reconstructed from serialized method events. Ports serve as interface to connect two stages with each other.

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

tecture and performance issues with Kieker's Pipes-and-Filters-oriented analysis component. Feedback from users confirm these issues. Although suitable for most post-mortem analyses, the framework is not capable of processing the amount of data required for more complex analyses, such as live architecture reconstruction and performance anomaly detection. The underlying analysis implementation does not take advantage of multi processor systems and relies extensively on slow reflection calls and string comparison. Moreover, Kieker's analysis component does not support the composition of multiple stages.

There are only very few Pipes-and-Filters frameworks that are not tailored to a particular use case, but designed for arbitrary pipeline architectures. To the best of our knowledge, none of them are easy to use, extensible, efficient, and address both the usage and the abstraction of multi-core architectures in one single framework.

Hence, we decided to develop a generic and concurrency-aware Pipes-and-Filters framework using our experience and our requirements from Kieker. This includes a fast reference implementation called TeeTime, ports for stages, and a convenient application programming interface which abstracts from the management of concurrency.

Our framework is not limited to Kieker and thus can be used in other projects as well. Furthermore, it can be applied to various programming languages, although TeeTime is written in the Java programming language. A manual and the source code of TeeTime are publicly available at Sourceforge[1].

Hence, our contributions are:

- An approach for a generic and concurrency-aware Pipes-and-Filters framework, and
- a reference implementation for the Java programming language.

The rest of this document is structured as followed. First, we present related work and existing frameworks in Section 2. In the subsequent two sections, we describe our approach focusing on its architecture in Section 3 and its concurrency handling in Section 4. Section 5 finally contains our conclusion and future work.

2 Related Work

Due to the fact that Pipes-and-Filters is a frequently used pattern, various generic solutions exist already. We present and discuss an excerpt of existing frameworks focusing on Java-based implementations. Furthermore, we list work regarding strategies for the concurrent execution of stages.

[1] `https://sourceforge.net/projects/teetime/`

2.1 Current Java-based Pipes-and-Filters Frameworks

A generic Pipes-and-Filters framework is Apache Commons Pipeline[2]. This framework can take advantage of additional threads, but assumes that the stages are implemented in a thread-safe manner. Furthermore, the project has no released version and is not developed any further since 2009.

Apache Camel[3] is a Java framework to configure routing and mediation rules using the enterprise integration patterns. It can also be used to assemble Pipes-and-Filters oriented systems. However, in contrast to TeeTime, it does neither support typed ports nor the concurrent execution of multiple stages. The official recommendation to handle concurrency is, among others, the usage of a database as synchronization point[4].

Ptolemy II [BL10] is a Java framework that supports experimenting with the actor-oriented design. Although it can be used to assemble networks in a Pipes-and-Filters oriented style, it requires additional knowledge to configure and execute a pipeline configuration that goes beyond the Pipes-and-Filters pattern. Furthermore, its use of a scheduler and coarse-grained synchronization mechanisms results in an additional run-time overhead.

The Kieker Monitoring and Dynamic Analysis Framework[5] provides a Pipes-and-Filters-API to create analysis networks. Due to its drawbacks mentioned in Section 1, we will replace it by TeeTime's API.

ExplorViz [FWBH13] is a tool that enables live trace visualization for system and program comprehension in large software landscapes. It comprises a Pipes-and-Filters-based component that is tailored to the processing of program traces. Hence, it is not suited as a generic Pipes-and-Filters framework.

XML Calabash[6] is an implementation of the XML pipeline language XProc. [7] This language can be used to describe pipelines consisting of atomic or compounded operations on XML documents.

Pipes[8] is a Java-based framework using process graphs. So called pipes represent the atomic computing steps and form, once connected, the processing graph. As the pipes implements interfaces with generics, they are type-safe, but have to be arranged in sequential order.

Java 8 introduced a new streams API that allows to successively apply multiple functions on a stream of elements. Besides the lack of reading from and writing to more than one stream at once, its support for executing stages in parallel is limited to particular use cases.

Akka[9] is a framework for both Scala and Java following the actor model [HBS73]. It focuses on scalability (regarding concurrency and remoting) and fault tolerance. Although

[2] `http://commons.apache.org/sandbox/commons-pipeline/`
[3] `https://camel.apache.org/`
[4] see `http://camel.apache.org/parallel-processing-and-ordering.html`
[5] `http://kieker-monitoring.net/`
[6] `http://xmlcalabash.com/`
[7] `http://www.w3.org/TR/xproc/`
[8] `https://github.com/tinkerpop/pipes/wiki`
[9] `http://doc.akka.io`

it is possible to map Akka's actor-based API to a Pipes-and-Filters-based one, Akka is not optimized for the execution of pipeline architectures.

2.2 Concurrent Execution of Stages

There are at least two common strategies to execute a pipeline configuration concurrently [SLY+11, SQKP10].

The first strategy (S1) distributes the given threads over a distinct subset of the declared stages, e.g., each thread executes a single stage. Depending on whether two connected stages are executed by the same thread or by different threads, the corresponding pipe is synchronized or unsynchronized, respectively. Stages are typically not synchronized as each of them is executed by only a single thread.

The major challenge of this strategy lies in finding the optimal assignment of threads to stages. While static assignment approaches are usually more efficient for stable and predictable pipeline configurations, dynamic assignment approaches can additionally handle imbalanced stages.

The second strategy (S2) assigns a copy of the whole pipeline structure to each thread. Each thread maintains a sorted list of all available pipes. Moreover, each thread uses a scheduler that iteratively takes one of the last stages of the pipeline whose input pipes are non-empty. This strategy does not require pipes to be synchronized, but rather stateful stages since the copies of a stage usually share one single state.

Some approaches [SLY+11, NATC09] use work-stealing pipes to balance the workload across all available threads. Such pipes then require single-producer-multiple-consumers data structures that cause additional run-time overhead due to further synchronization and management effort.

3 Toward the TeeTime Framework

Our framework targets software engineers that need to process data in a stream-oriented, throughput-optimized, and type-safe fashion. Although our reference implementation is written in Java, our framework's software architecture can be easily adapted to other object-oriented programming languages that are aware of threads and type parameters, e.g., C#.

We base our framework on the Tee-and-Join-Pipeline design pattern [BMR+96], a generalized version of the Pipes-and-Filters design pattern. It allows a stage to be connected by more than one input and output pipe. For this reason, we call our reference implementation TeeTime.

After giving an overview of our framework's architecture, we describe the most important features in more detail.

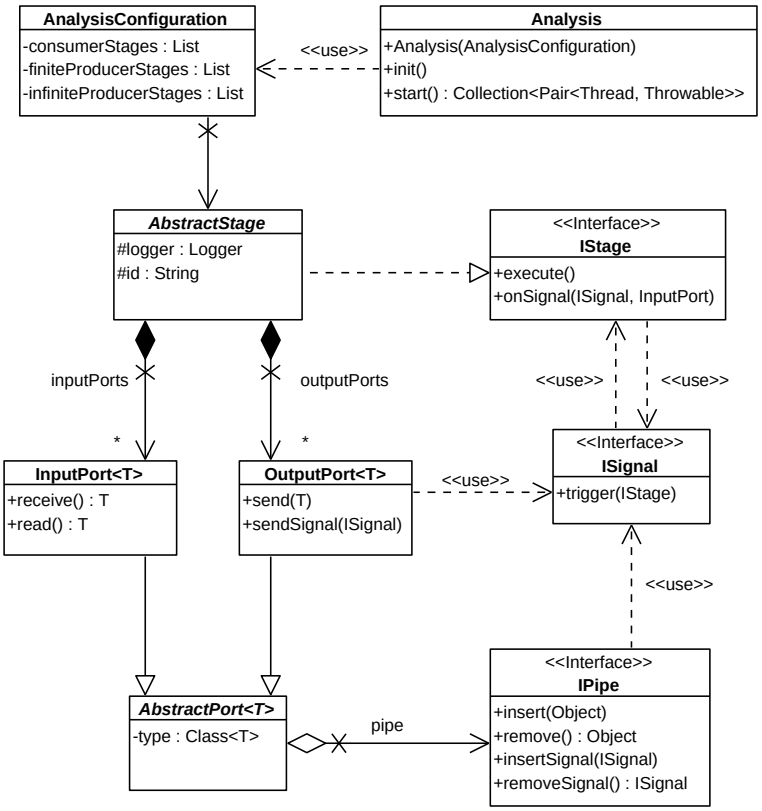


Figure 2: Overview of the basic entities in our Pipes-and-Filters framework architecture

3.1 Overview of the Software Architecture

The core of our framework consists of four basic entities: `Stages`, `Ports`, `Pipes`, and `AnalysisConfigurations`.

A stage represents a processing unit within a pipeline that takes elements from its *input ports*, uses them to compute a result, and puts this result to its *output ports*. Therefore, the concept of ports allows a stage to communicate with multiple predecessors and successors. Additionally, a stage is assigned a unique identifier and a logger to ease in debugging.

A pipe connects an output port of one stage with an input port of another stage by means of a queue. However, a pipe can also be used to realize self-loops by connecting an output port with an input port of the same stage.

An analysis configuration represents a concrete pipeline setup. It defines the stages that should be used and connects their ports with each other.

In Figure 2 we show an overview of the basic entities in our Pipes-and-Filters framework architecture.

3.2 Stage Scheduling

Our framework does not make use of an explicit stage scheduler to decide what stage should be executed next. Instead the pipes between the stages undertake this task.

Each time a stage sends an element to one of its output port, the corresponding connected pipe stores the element in its internal queue. Depending on the pipe's implementation, it then either triggers the execution of the receiver stage or it immediately returns to the sender stage.

The former implementation is used to realize intra-thread communication, i.e., to connect two stages that are executed by the same thread. It ensures progress and reduces the memory usage since passing an element through the whole pipeline has the highest priority. This strategy is also known as the *backpressure* technique.

The latter implementation is used to realize inter-thread communication, i.e., to connect two stages that are executed by two different threads. It causes only a minimal delay in the execution of the sending stage.

In this way, we gain the highest possible throughput with respect to the framework's flexible architecture. For example, we require a throughput of ten thousands to millions of elements per second when analyzing monitoring logs with Kieker. In such situations, an explicit scheduler causes too much run-time overhead because it needs to switch too often between all stages.

The avoidance of a dedicated scheduler additionally reduces the overall complexity of the framework itself and allows the (JIT-)compiler to perform further optimizations, e.g., utilizing cache locality due to back-pressuring [SLY+11].

3.3 Type-safe Connection of Stages

We extend our framework with typed ports to provide a fully type-safe environment. Therefore, it is impossible to connect two incompatible ports which dramatically reduces the need for debugging.

The framework checks for type safety at compile-time and at run-time. The former is realized by specifying a type parameter for each port. In this way, the compiler is able to verify that a pipe connects two ports of the same type.

The latter is realized by checking whether each two ports have the same type attribute (cp. Figure 2) just before a pipeline is being executed. It is not possible to check the type parameter at run-time because some object-oriented programming languages, such as Java, perform type-erasure on the type parameters.

3.4 Reacting to Signals

We further extend our framework by a generic signal concept. A signal represents an event that is triggered within one stage and passed to all other stages with respect to the given pipeline architecture.

Similar to transmitting an element, a signal is sent from an output port to an input port using a pipe. For this purpose, both the intra-thread and the inter-thread pipe implementation already mentioned above have another internal queue that is used for signals only.

For example, a `StartingSignal` is triggered and processed by the first stage when the execution of a pipeline begins. Afterwards the signal is passed to all successor stages of the first stage where the same procedure is repeated until the last stage has processed the signal.

Besides the `StartingSignal`, our framework implementation currently provides a `ValidatingSignal` that occurs when the connections of a stage are being validated and a `TerminatingSignal` that occurs when the execution of a pipeline is being terminated.

Moreover, our framework allows to add further signal types. A new signal type must conform to the `ISignal` interface and needs to be triggered once. The correct signal passing is then handled by our framework automatically.

3.5 Composition of Stages

Stage composition is a key concept that allows to build on top of already available stages. It effectively hides complexity and also improves the usability. For this reason, our framework offers direct support for composing stages.

3.6 Basic Stages

While developing and using Kieker [RvHM+08, vHRH+09, vHWH12], we have identified various atomic stages that are often used either directly or indirectly to build more complex, high-level stages. Below, we describe some of the most important ones in more detail.

Distributor The distributor is characterized by one input port and a customizable number of output ports. It takes an element from its input port and distributes it to its output ports. The distributor uses the `DistributionStrategy` interface to decide how the input element should be distributed among the output ports. We currently provide implementations that forwards the input element either to one output port according to the round-robin style, or to each output port. Instead of simply forwarding the element, a

solution would be conceivable where the distributor delivers a deep clone of the input element.

Merger The merger is characterized by a customizable number of input ports and a single output port. It takes elements from a subset of its input ports and merges them to its output port. Similar to the distributor, the merge step is done according to a particular implementation of the `MergeStrategy` interface. We currently provide a round-robin implementation that forwards the element of the next non-empty input port to the output port.

File system stages The directory reader outputs all available files within a given directory. The text line file reader reads in a text file from its input port and successively sends each text line via its output port. The file writer takes a byte array from its input port and writes it to a pre-configured file.

Generic stages Besides the stages mentioned above, there are also generic stages that process elements independent of the elements' types. The repeater stage takes one element from its input port and outputs a multiple of it to its output port. The delay stage delays the passing of an element to its successor stage by a customizable amount of time. The throughput stage counts the number of passing elements and outputs the corresponding throughput if it is triggered by another stage to do so. The InstanceOfFilter stage checks whether an incoming element matches a configurable particular type and passes it to one of its two output ports representing a valid or invalid match.

Further stages We continuously extend the set of provided stages. Besides the basic ones mentioned above, we currently offer stages that, e.g., count the words contained in a string, encrypts and decrypts a byte array, and outputs the current time in a configurable regular interval.

We also provide stages that interact with I/O devices, such as the file system and the network (via TCP, JMX, JMS, and JDBC). In this way, it is also possible to build a pipeline distributed over several computing nodes.

3.7 Example stage

We now consider the TeeTime implementation of the directory reader described above (see Listing 1) to show (1) that the API only requires few additional knowledge beyond the Pipes-and-Filters pattern itself and (2) that it is able to abstract from any concurrent data structure or directive.

The stage extends the `ConsumerStage` that, among others, provides methods for the declaration of ports (see line 3) and encapsulates the handling of the first input port by

means of the parametrized `execute` method (see line 11). If a stage requires more than one input port, it simply declares it and checks the ports on available input.

Line 5 and 6 contain the declarations of a file filter and a file comparator, respectively. Both are used in the `execute` method to list only relevant files (see line 12) in the desired order (see line 20). Finally, the read files are successively send via the declared output port (see lines 23-25).

Hence, the source code only contains those statements that are necessary to declare a directory reader. In particular, it does not make use of any concurrent data structure, e.g., a concurrent blocking queue, and any concurrent directive, such as `volatile` or `synchronized` to interact with its predecessor or successor stage.

Listing 1: The TeeTime implementation of the directory reader

```
1  public class Directory2FilesFilter extends ConsumerStage<File> {
2
3    private final OutputPort<File> outputPort = this.createOutputPort();
4
5    private FileFilter filter;
6    private Comparator<File> fileComparator;
7
8    // omitted costructors
9
10   @Override
11   protected void execute(final File inputDir) {
12     final File[] inputFiles = inputDir.listFiles(this.filter);
13
14     if (inputFiles == null) {
15       this.logger.error("Directory '" + inputDir + "' does not exist or
             an I/O error occured.");
16       return;
17     }
18
19     if (this.fileComparator != null) {
20       Arrays.sort(inputFiles, this.fileComparator);
21     }
22
23     for (final File file : inputFiles) {
24       this.send(this.outputPort, file);
25     }
26   }
27
28   // omitted getter and setter
29
30   public OutputPort<File> getOutputPort() {
31     return outputPort;
32   }
33
34 }
```

4 Concurrency Handling

To the best of our knowledge there is no similar Pipes-and-Filters framework that manages and handles a concurrent execution in such a transparent and efficient way. In the following, we describe how our approach concurrently executes the stages, how it handles synchronization issues, and how it does everything in a highly efficient and automatic way.

4.1 Concurrent Execution of Stages

Since there are several different use cases for using a Pipes-and-Filters-oriented architecture, we do not commit ourselves to one of the two strategies mentioned in Section 2. Instead, we recommend a hybrid approach that makes use of both approaches.

Consider the sample pipeline consisting of a producer and a consumer whereby elements are produced faster than consumed. In this scenario, duplicating and executing the producer in more than one thread does not increase the throughput since the consumer is the bottleneck. Furthermore, if the producer reads from an I/O device, instantiating more than one producer is often not advisable since most I/O devices can be accessed only in a sequential way excluding any concurrent access. However, duplicating and executing the consumer stage increases the performance provided the internal state can be shared efficiently. Hence, a combination of S1 and S2 can perform better than each of the strategies in isolation.

4.2 Automatic Thread Instantiation and Management

To enable concurrent execution of stages, a number of threads and a suitable management handling their life-cycle is required. Instead of manually instantiating and managing these threads for each individual pipeline, we propose a more abstract concept that encapsulates such technical issues by our framework.

For this purpose, each stage is declared either as consumer or producer where a producer is further divided into finite or infinite.

A **consumer** changes its internal state and/or produces one or more output elements if and only if it receives one or more input elements. Thus, a consumer stage provides at least one input port. It terminates when it receives a termination signal.

A **producer** changes its internal state and/or produces one or more output elements according to its semantics. It provides no input ports and is responsible for its termination.

A producer is **finite** if it terminates itself after a finite number of executions. A producer is **infinite** if it does not terminate autonomously. In this case, the framework terminates the producer stage after all finite producers are terminated.

Based on this categorization, the framework is able to determine the amount of threads

necessary for the execution and the assignment of the stages to the threads. For instance, since a producer is independent of other stages, it can be executed concurrently. However, for the sake of fine-grained optimizations, it is also possible to manually assign stages to threads.

4.3 Synchronization of Stages

4.3.1 Synchronization by Pipe

In our framework, a pipe is responsible for the transmission of elements between two stages.

When connecting two stages within the same thread, the framework chooses an unsynchronized pipe implementation holding a single element. Once a stage sends an element, the corresponding pipe stores it and executes the receiver stage. The receiver stage finally pulls the element from the pipe.

When connecting two stages executed within two different threads, the framework chooses a pipe implementation consisting of a synchronized queue. Once a stage sends an element, the corresponding pipe adds it in a non-blocking fashion to the queue. The receiver stage can either use a busy-waiting or a blocking strategy in order to pull the element from the pipe.

The synchronized queue is a highly optimized implementation[10] for the single-producer/ single-consumer scenario. It is lock-free, cache-aware, and uses only a few well-placed load/load and store/store barriers for the synchronization. In particular, it does not make use of any types of the Java Concurrency API and of any volatile declaration. We have also tested a pipe implementation that bases on the work-stealing paradigm, however it performs significantly slower due to the heavy use of volatile fields.

4.3.2 Synchronization by Shared State

We call a stage stateful if it contains attributes and uses them to compute its outputs. Otherwise we call a stage stateless. It is recommended to implement stages in a stateless way, as this avoids synchronization overhead and issues between stages. However, in the Kieker project we identified various scenarios relying on stateful stages.

For instance, consider an aggregation stage that outputs a result not until a particular number of executions. Before reaching such a threshold, this stage needs to memorize the result of the current and all previous executions.

Our framework does not yet help in synchronizing a shared state although we plan to abstract from particular scenarios (see Section 5). Currently, a developer must be aware of concurrency issues that could arise due to a shared state and has to ensure a proper synchronization.

[10] `https://github.com/JCTools/JCTools/`

5 Conclusions and Outlook

In this paper, we indicated the drawbacks of current Pipes-and-Filters frameworks and proposed a more flexible and concurrency-aware Pipes-and-Filters framework architecture. It focuses on a type-safe application programming interface that makes complex decisions for the programmer and abstracts from technical issues to avoid building an incorrect and inefficient pipeline. We also provide a fast Java-based reference implementation called TeeTime.

We constantly improve our framework by reducing the management overhead, by adding more functionality, and by simplifying the API. Our next step is to give the responsibility of choosing and instantiating a pipe completely to the framework. Afterwards, we plan to automatize the thread assignment as much as possible so that ideally the programmer only needs to describe the pipeline structure and to specify the number of processor cores to use. For this purpose, we follow the idea to utilize further stage information available at compile-time, such as the ratio of the input and output ports or the properties of a stage's state. Moreover, we work on an automatic exception handling by the framework and a way to save and load arbitrary pipeline configurations that were created at run-time.

References

[BL10] Christopher Brooks and Edward A. Lee. Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java, February 2010. Poster presented at the 2010 Berkeley EECS Annual Research Symposium (BEARS).

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1. edition, 1996.

[FWBH13] Florian Fittkau, Jan Waller, Peer Christoph Brauer, and Wilhelm Hasselbring. Scalable and Live Trace Processing with Kieker Utilizing Cloud Computing. In *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*, volume 1083, pages 89–98. CEUR Workshop Proceedings, November 2013.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[NATC09] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical Modeling of Pipeline Parallelism. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 281–290, September 2009.

[RvHM+08] Matthias Rohr, André van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke, and Wilhelm Hasselbring. Kieker: Continuous Monitoring and on demand Visualization of Java Software Behavior. In Claus Pahl, editor, *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08)*, pages 80–85, Anaheim, CA, USA, Februar 2008. ACTA Press.

[SLY+11] D. Sanchez, D. Lo, R.M. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *Proceedings of the 20th International Con-*

ference on Parallel Architectures and Compilation Techniques (PACT), pages 22–32, October 2011.

[SQKP10] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. Feedback-directed Pipeline Parallelism. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, PACT '10, pages 147–156, New York, NY, USA, 2010. ACM.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Forschungsbericht, Kiel University, November 2009.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, April 2012.

Evaluation of Alternative Instrumentation Frameworks

Dušan Okanović, Milan Vidaković

Faculty of Technical Sciences
University of Novi Sad
Fruškogorska 11
Novi Sad, Serbia
oki@uns.ac.rs
minja@uns.ac.rs

Abstract: Our previous research focused on reducing continuous monitoring overhead by employing architectural design that performs adaptive monitoring. In this paper we explore the use of other AOP or AOP-like tools for instrumentation. The goal is to find a tool that has lower overhead than AspectJ. Support for runtime changes of monitoring configuration is also considered. Our main topic of interest is DiSL framework, because it has similar syntax as, already used AspectJ, but allows for more instrumentation options.

1 Introduction

When monitoring application performance parameters under production workload, during continuous monitoring process, everything must be done to reduce the overhead generated by monitoring system. This overhead is highly unwanted because it can have negative effect on the end user's experience. Although inevitable, this overhead can be minimized.

The DProf system [OvHVK13] for continuous monitoring uses instrumentation, and sends gathered data to a remote server for further analysis. After this data is analyzed, new monitoring configuration can be created in order to find the root cause of the performance problem. New parameters turn monitoring off, where performance data is within expected, and on, where data shows problems. This way, the overhead is reduced, because only problematic parts of software are monitored. First implementations of our tool used AspectJ[1] for instrumentation. The downside of this approach was the fact that the bytecode with weaved aspects is not fully optimized. The result was higher overhead than expected. Also, the monitored system had to be restarted each time new monitoring parameters are set.

We have explored the use of other aspect-oriented or similar tools, mainly DiSL framework [MVZ+12], with our system. The goal is to find a tool that has lower overhead than AspectJ. Support for runtime changes of monitoring configuration is also considered.

[1] http://www.eclipse.org/aspectj (October 2014)

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

The remainder of the paper is structured as follows. In section 2. we provide an overview of the DProf system, current implementation problems, and possible AspectJ alternatives. How to replace AspectJ with DiSL is shown is section 3. In the last section we draw conclusions and outlines for future work.

2 DProf system

The DProf system proposed in [OvHVK13] has been developed for adaptive monitoring of distributed enterprise applications with a low overhead.

For monitoring data acquisition, DProf utilizes the Kieker[2] framework [vHWH12]. The main reason for this was low overhead the Kieker imposes on a monitored system. Unlike profilers and debuggers used by software developers during development process, the Kieker separates monitoring from analysis of gathered data.

Deployment diagram of the DProf monitoring system is shown in figure 1. Monitoring probes gather data. Kieker's Monitoring Controller directs this data to the DProfWriter. DProfWriter sends data into the ResultBuffer. ResultBuffer holds the data and sends it in bulks to the RecordReceiver periodically. RecordReceiver then stores it into the database.

Analyzer performs the analysis of the data, reconstructs call trees from it, and, when required, creates a new set of monitoring parameters. New parameters are created based on configuration file, described in [OVK12]. Parameters are received by DProfManager. This component controls the work of the ResultBuffer and the AspectController. AspectController configures AspectJ framework through modifications in aop.xml configuration file.

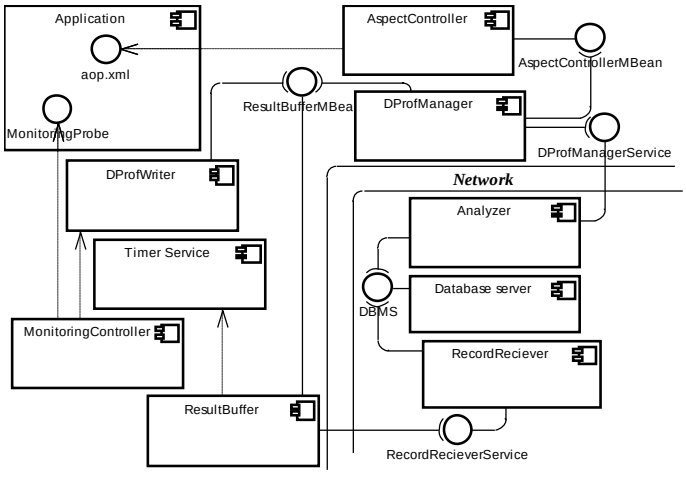


Figure 1. Deployment diagram of the DProf monitoring system

[2] At the time of implementation, Kieker 1.6 was used.

Additional components that support the change of monitoring parameters at application runtime have been developed using Java Management Extensions[3]. Use of the JMX technology allows for the reconfiguration of the DProf monitoring parameters both manually and programatically.

2.1 Limitations of this approach

In some situations the use of AspectJ imposed significant overhead that no architectural redesign of the system could reduce. While this overhead was very small in simple monitoring cases, there were cases where it rose unexpectedly. In those cases reflection and access to join-point information was used.

Another problem is the fact that AspectJ does not allow the join points to be inserted within the methods. To be more precise, they are not able to "see" into a method and check if it contains loops, since these methods are most likely to cause performance lags.

Weaving of aspects is usually performed at load-time. This means that for every monitoring configuration change, the system has to be restarted - both monitoring system and monitored application. This results in reduction of overall quality of service, because availability is reduced. Although careful planning can, as shown in [OV12], but the fact remains: run-time loading and unloading of instrumentation should be employed.

Another approach is to use proxy classes, described in [GHJV94]. All of the methods would be instrumented using aspects, but proxy classes would choose whether to accept their performance data or not. While this approach is very flexible, it adds another layer of classes and method invocations into the system, leading to even more overhead.

It is obvious that AspectJ has to replaced with another framework. This new framework will have to impose very low overhead and support run-time insertion and removal of monitoring probes.

2.2 Possible AspectJ alternatives

Since Java is a statically typed, it is very hard to manipulate code at runtime. In current Java virtual machines (JVMs), it is possible only to replace method bodies, not method signatures. This hotswapping of loaded classes is usually implemented using JVM Tool Interface[4] (JVMTI).

There are several bytecode manipulation libraries. These allow for manipulation at very low level, enabling developers to optimize instrumentation and place probes almost arbitrarily. Some of these tools are ASM[5], BCEL[6], Javassist[7] and Soot[8]. However, the resulting instrumentation code is difficult to read, maintain and debug.

[3] JMX, http://www.oracle.com/technetwork/java/javamail/javamanagement-140525.html
[4] JVMTI, http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti
[5] ASM, http://asm.ow2.org/

Modern AOP tools are based on these bytecode manipulation tools, but provide higher abstraction layer for defining instrumentation. However, AOP has not been designed for profiling and continuous monitoring. AOP and modern AOP tools suffer from supporting limited set of join points, and they do not support instrumentation of basic blocks, loops, or even, byte codes. Also, bytecode generated by AOP tools is not always optimized.

Dynamic AOP [PGA02] approach enables runtime adaptation of applications, and consequently monitoring systems, by changing aspects and reweaving code in a running system. It enables creation of tools where developers can refine the set of dynamic metrics of interest and choose the application components to be analyzed while the target application is executing. Such features are essential for analyzing complex, long-running applications, where the comprehensive collection of dynamic metrics in the overall system would cause excessive overheads and reduce developers' productivity. In fact, state-of-the-art profilers, such as the NetBeans Profiler [Dim03], rely on such dynamic adaptation, but currently these tools are implemented with low-level instrumentation techniques, which cause high development effort and costs, and hinder customization and extension.

Existing dynamic AOP frameworks are implemented using one of the following approaches.

The first approach uses pre-runtime instrumentation to insert hooks - small pieces of code - at locations that can become join-points. These locations are determined using pre-processing, and applied using load-time instrumentation or on just-in-time compilation. Another approach is to implement runtime event monitoring using low-level JVM support to capture events - method entry/exit and field access. The most challenging approach is to implement runtime weaving. It can be implemented with customized JVM or using JVM hotswapping support.

PROSE [NAR08] platform has been implemented in three versions. The first uses, now obsolete, JVM Debugging Interface[9]. The second is implemented based on the IBM Jikes Research Virtual Machine and has very large overhead. The third version is implemented for HotSpot and Jikes JVMs, but is not able to work with code where compiler already performed optimizations, such as method inlining. JAsCo [VSV+05] introduces new AOP language and concepts of aspect beans and connectors. Aspect beans are used to define join-points and advices. Connectors deploy aspect beans in a concrete component context. The development of JAsCo technology has been stalled for some time now.

HotWave [VBAM09] uses existing industry standard AspectJ language, and generates code that can be used by hotswapping mechanism. Aspects can be woven right after JVM bootstrapping, but also while the code is executing. Previously loaded classes are

[6] BCEL, http://commons.apache.org/proper/commons-bcel/
[7] Javassist, http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist
[8] Soot: a Java Optimization Framework, http://www.sable.mcgill.ca/soot
[9] http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/architecture.html

hotswapped with classes woven with new aspects. If the class was already weaved with aspects, new weaving uses original class bytes, not those from previously weaved version. HotWave lacks support for around advices. The workaround is to use a pair of before and after advices and inter-advice communication. This is acceptable for continuous monitoring because we do not want to change program behavior. Inter-advice communication allows the creation of synthetic local variables that can be passed between any advice. This is something AspectJ and other AOP frameworks do not support. HotWave has never been fully developed, and remained only a prototype.

Another approach similar to HotWave is shown in [WWS10]. DCE VM is an extension of standard JVM. It allows the classes within to be changed while JVM executes. This tool has also been only a prototype.

Domain specific language for instrumentation (DiSL [MVZ+12]) has been developed to counter some of the problems that occur when using AOP for Java software monitoring. Considering the level of abstraction, DiSL is somewhere between low-level tools like ASM, and high-level tools like AspectJ.

Using DiSL guarantees that the monitored software will never change its behavior, something that ASM does not. DiSL developer has to deal with some details of byte code manipulation, but much less than when using tools like ASM. Code generated using AspectJ will always pass bytecode verification, while with DiSL, developer has to ensure that it passes. Unlike AspectJ, it allows for instrumentation to be inserted within methods.

DiSL uses similar pointcut/advice model as AspectJ, and even similar syntax, but it removes some of the constructs. One of the omitted constructs is around advice. This advice is often used by developers of dynamic analysis tools. Instead of it, a combination of before/after advices can be used, with the addition of synthetic local variables for inter-advice communication. DiSL constructs are transformed into code snippets that are inlined before or after indicated bytecode sections. The omission of around is not a problem when constructing monitoring tools. Around advices intended use is changing of program's behavior, something that monitoring should not do.

Performance evaluation of monitoring tools developed with DiSL showed less overhead than AspectJ implementation of such tool, while providing more functionality (e.g. basic block analysis). Code generated by DiSL weaver is smaller, thus making the memory footprint of the monitoring tool smaller.

3. Using DiSL With DProf

Replacing of the AspectJ with the DiSL will be shown on the example already shown in [OvHVK13]. In this example we monitor the software configuration management application.

So far, we have used the monitoring probe implemented as aspect shown in our previous papers. This aspect intercepts execution of annotated method, and in around advice we perform measurements.

Since DiSL does not support around advices, we have to implement two new advices - `@Before(...)` and `@After(...)`. New aspect is shown in Listing 1. Synthetic local variable startTime (annotated with `@SyntheticLV`) holds the value of the method execution start time between before and after advice execution. In before advice we take time when method execution starts, and in after advice we take end time, and create and store monitoring record, in the same way as in original aspect.

```
1 public class ExecutionTimeMonitoring {
2   @SyntheticLocal public static long startTime;
3   @Before(marker = BodyMarker.class, guard =
                      DProfAnnotatedGuard.class)
4   static void onMethodEntry() { startTime = System.nanoTime();}
5
6   @After(marker = BodyMarker.class, guard =
                                  DProfAnnotatedGuard.class) {
7   static void onMethodExit(MethodStaticContext msc) {
8      double endTime = System.nanoTime();
9      DProfExecutionRecord dProfExec =
          new DProfExecutionRecord(..., endTime - startTime, ...);
10     MonitoringController.getInstance()
          .newMonitoringRecord(dProfExec);
11  }
12 }
```

Listing 1. DiSL aspect for execution time monitoring

This class is weaved with application classes and is used to monitor call tree shown in Fig. 2.

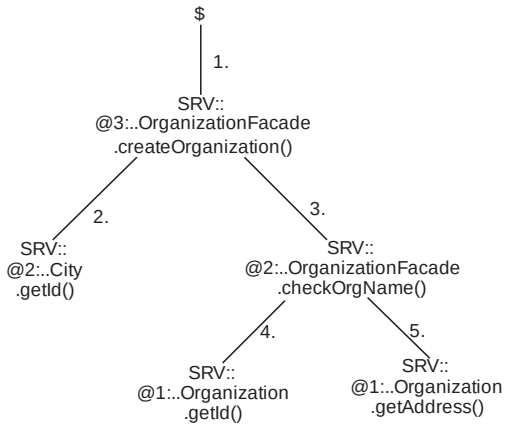


Figure 2. Monitored call tree

In the first pass only the root method - was weaved and monitored. When deviation from expected performance was detected, another level of nodes was included into monitoring. In the next pass, no deviation from expected values was detected in execution of `City.getId()` method, while there was a deviation in `OrganizationFacade.checkOrgName()` results. `City.getId()` was un-weaved, and methods invoked from `OrganizationFacade.checkOrgName()` were weaved. In the last pass, no deviation was detected in the results for methods in the lowest level of the call tree, and `OrganizationFacade.checkOrgName()` was declared to be the cause of the problem.

In the background, Analyzer analyzed the obtained data. Based on the analysis results, it issued commands to the DProfManager. Based on these commands, DiSL performed aspect weaving.

The main goal of this experiment was to measure performance overhead ogenerated by monitoring. The overhead was measured by repeatedly invoking monitored method (monitoring configuration was fixed). On our test platform, implementation that uses DiSL yielded approximately 1.2% less overhead then AspectJ implementation.

Some performance peaks were detected, in both cases, but only when weaving was initiated, on application restart. After reweaving, as when JVM restarts, new classes are loaded, linked and just-in-time compiled. This causes longer execution times at the beginning of each DProf cycle.

4 Conclusion

This paper presented the use of the DiSL framework with the DProf system in order to reduce performance overhead. The result is a tool that can be used for continuous monitoring of any kind of applications, including distributed applications. Because DiSL uses similar syntax to AspectJ, and is fully Java based, learning curve for this new tool is not an issue.

The evaluation of DProf/DiSL combination was performed by monitoring the sample software configuration management application, which was monitored using DProf on AspectJ platform in our previous work. The comparison of the results shows that the generated overhead is slightly less when using DiSL.

Further work depends on the DiSL development. FRANC [AKZ+13] framework, built on DiSL, will provide the possibility to implement runtime changing of monitoring parameters.

References

[AKZ+13] Ansaloni, D.; Kell, S.; Zheng, Y.; Bulej, L.; Binder, W.; Tůma, P.: Enabling modularity and re-use in dynamic program analysis tools for the java virtual machine. In

Proceedings of the 27th European conference on Object-Oriented Programming (ECOOP'13). Springer-Verlag, Berlin, Heidelberg, p. 352-377. 2013.
[BLC02] Bruneton, E.; Lenglet, R.; Coupaye, T.: ASM: A Code Manipulation Tool to Implement adaptable systems. http://asm.ow2.org/current/asm-eng.pdf. 2013.
[Dim03] Dimitriev, M.: Design of JFluid. Technical Report: SERIES13103. Sun Microsystems Inc., USA. 2003.
[GHJV94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1994.
[MVZ+12] Marek, L.; Villazón, A.; Zheng, Y.; Ansaloni, D.; Binder, W.; Qi Z.: DiSL: a domain-specific language for bytecode instrumentation. In Proceedings of the 11th annual international conference on Aspect-oriented Software Development (AOSD '12). p. 239-250. 2012.
[NAR08] Nicoara, A.; Alonso, G.; Roscoe, T.: Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. SIGOPS Operating Systems. v. 42, n. 4. p. 233-246. 2008.
[OV12] Okanović, D.; Vidaković, M.: Software Performance Prediction Using Linear Regression. In Proceedings of the 2nd International Conference on Information Society Technology and Management. Kopaonik, Serbia. p. 60-64. 2012.
[OVK12] Okanović, D.; Vidaković, M.; Konjović, Z.: Monitoring of JEE Applications and Performance Prediction. Journal of Information Technology and Applications, v.1, n.2. p. 136-143. 2012.
[OvHKZ13] Okanović D.; van Hoorn, A.; Vidaković, M.; Konjović, Z.: SLA-Driven Adaptive Monitoring of Distributed Applications for Performance Problem Localization, Computer Science and Information Systems (ComSIS), vol. 10, no. 1, pp. 25-50, 2013.
[PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso. 2002. Dynamic weaving for aspect-oriented programming. In Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02). ACM, New York, NY, USA, p. 141-147
[VBAM09] Villazón, A.; Binder, W.; Ansaloni, D.; Moret, P.: Advanced Runtime Adaptation for Java. SIGPLAN Not. v. 45, n. 2. p. 85-94. 2009.
[vHWH12] van Hoorn A.; Hasselbring, W.; Waller, J.: Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012). ACM, Boston, Massachusetts, USA. p. 247-248. 2012.
[VSV+05] Vanderperren, W.; Suvée, D.; Verheecke, B.; Cibrán, M. A.; Jonckers, V.: Adaptive Programming in JAsCo. In Proceedings of the 4th international conference on Aspect-oriented software development (AOSD '05). p. 75-86. 2005.
[WWS10] Würthinger, T.; Wimmer, C.; Stadler, L.: Dynamic Code Evolution for Java. 8th International Conference on the Principles and Practice of Programming in Java, Vienna, Austria. 2010.

Acknowledgement

The research presented in this paper was supported by the Ministry of Science and Technological Development of the Republic of Serbia, grant III-44010, Title: Intelligent Systems for Software Product Development and Business Support based on Models.

The DIN/ISO definition and a measurement procedure of software efficiency

Dr. W. Dirlewanger

Kassel University, Prof. i. R.
Dept. of Mathematics and Computer Science
Mozartstrasse 1034246 Vellmar
performance-we@t-online.

Summary and foreword: People like to assign the attribute speed (i. e. performance) not only to hardware but also to software. But software does not have an attribute speed. Software defines - for each user initiated task - a sequence of information processing steps (machine instructions) to be performed. A suitable term for the aspect under construction can only be found when having a second imlementation (Imp2) of the application on the same hardware. We have to perform two performance measurements of the data processing system. Measurement 1, using the first implementation (Imp1), yields the performance value P1. Measurement 2, using the second implementation (Imp1), yields the performance value P2. To find a term describing the property of the software we compare P1 to P2. This yields a measure of „how good" Imp2 transforms the task submitted into a sequence of machine instructions compared to Imp1: Imp2 is less or more efficient than Imp1.

This procedure requires a fitting definition of performance and places great demands to the measurement method. It has to simulate in detail the properties of the user entity and has to deliver detailed performance values. This is done by the ISO 14656 method. It measures the timely throughput, the mean execution times and the total throughput, each as a vector of m values, where m is the total number of task types. Based on this data the following software efficiency values are defined: The timely throughput efficiency I_{TI}, the mean execution time efficiency I_{ME} and the total throughput efficiency I_{TH}, each being a vector of m values. Additionally there is the quotient of maximal timely served users I_{MAXUSR} which is a scalar.

In the talk the ISO method will be presented and explained and it will be shown how the software efficiency values are to be computed. All is supported by examples of real world measurement projects.

Though published years ago the ISO method is little known. Published in the golden age of Mainframe-MIPS may be it was far ahead of its time. But the author is still on fascinated of the abilities of this method. And he is convinced that it may be helpful to save it from falling in oblivion. It is one of the most useful tools for computer performance measuring and Software efficiency measuring. It is applicable also to simulated software for instance in the first phases of a software project. It is mandatory in every software project to specify the enduser requirements concerning the timeliness of the service. These requirements have to be described in the system specification . The ISO workload model is an excellent template for this part of the specification.

A historic aspect is: DIN 66273 was published in 1991 [1]. The software efficiency aspect was not yet part of this National Standard. The use of the DIN performance measurement method for getting software efficiency values was firstly proposed by the author in [3]. ISO took over the idea 1999 in ISO 14756 [2]. It is comprehensively described in [4].

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

1 The international Standard ISO/IEC 14756

-- Defines the performance term for system performance.
– It describes a method for defining workloads (ISO workload data model).
– It defines a method for measuring data processing performance of a defined IP system using the ISO workload data model.
– It defines the principles of the user emulator.
– It defines a method for rating the measured performance values.
.-- It defines a method for estimating the (runtime-) efficiency of (application- or system-) software which is part of the measured data processing system.

2 Special qualities of the ISO 14756 method

(1) Arbitrary system under test (SUT):
The users in an ISO type measurement are emulated by a remote terminal emulator (RTE) whose input information is an ISO workload parameter set (WPS). The RTE is connected via the real user interface to the SUT. This ensures that any SUT can be measured, independently of manufacturer, configuration, internal construction, operating system, CPU type, system architecture and so on. It may be for instance a mono-processor computer, a cluster of computers or a cloud service. So, even a PC can be compared to a mainframe or to a super computer.

(2) Independence of RTEs manufacturer:
The measurement and its results are independent of the manufacturer or programmer of the RTE and of the computer type used for the RTE. The only condition is that the RTE fulfills the specification of an ISO RTE and naturally can be connected to the SUTs user interface. Measurement results are not influenced by the RTEs individual technical details.

(3) Control of the correct work:
The ISO RTE specification includes a set of control mechanisms which insure the correct working of the RTE during a measurement. If – for instance – the hardware used in the RTE is not fast enough to emulate the planed number of users correctly this will become evident. The same applies to the RTE if it doesn't realize the user behavior correctly and so on.

(4) Nearly every benchmark can be represented in ISO form:
The ISO workload model is very universal. Nearly every benchmark can be brought to the ISO representation. Beginning with the classic “Debit Credit OLTP Benchmark”, nearly any multi-user-multi-programing benchmark, naturally the classic SPEC batch-benchmarks, or benchmarks for today's SAP/R3-applications can be represented as an ISO type workload.

(5) Component tests can also be rewritten to ISO form:
Even computer component tests, for instant speed measurement of a hard disk, can be realized by an ISO type workload; in this case the hard disk is the SUT and the hardware beyond the disk interface (for instance SATA) is the RTE.

(6) Emulated users can be human beings or machines:
Naturally it doesn't matter if the users emulated by the RTE are human beings or machines. For instance the SUT is the central unit of a computer and the regarded interface is the connection to the so-called frontend or a terminal multiplexer. This can also be described by an ISO type RTE.

(7) Forgery-proof by random task stream:
The ISO RTE generates the the task stream transmitted to the SUT randomly. Two subsequent measurements must not have identical job or task sequences as input. Nevertheless predefined global parameters - as for instance the relative task frequencies or think time mean values of the users - are realized according to the statistic accuracy defined in the workload. The ISO RTE works in detail randomly but global deterministic. Intentionally the ISO RTE doesn't use recorded job streams or task streams as a workload. In that case the SUT could be optimized on such a task stream and the measuring results would be falsified.

(8) Reproducibility of measurement results:
ISO results are repeatable. Whenever and whoever repeats a measurement the results are the same. In fact, this is an important difference to common “performance measurements”, that often lack on repeatability

(9) High precision:
The ISO method is a high precision measure method which delivers very exact results.

3 Measurement by emulated users

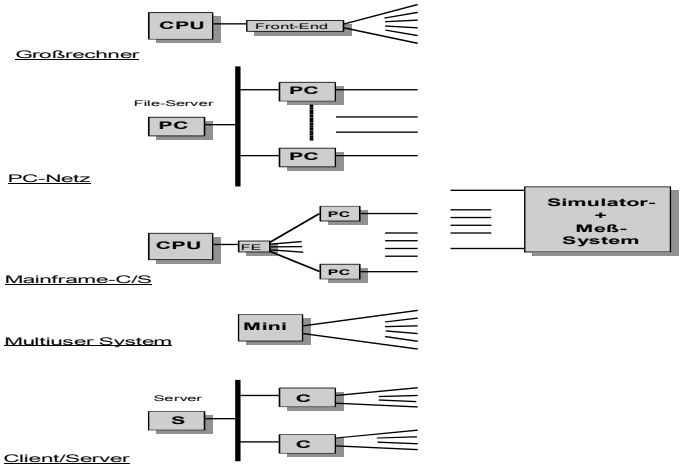


Fig. 1 SUT: Arbitrary type and architecture

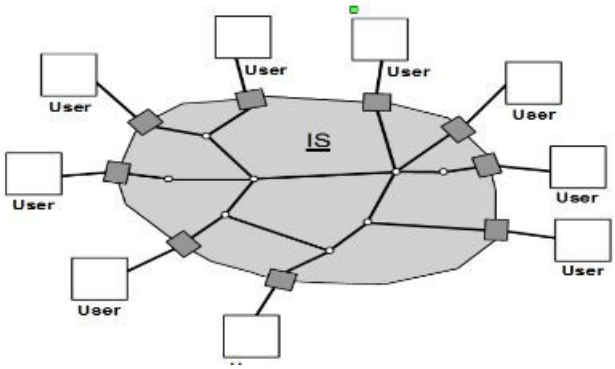


Fig.2 A SUT in real operation

For performing a measurement a RTE is used. It is table driven by the so-called workload parameter set. The ISO workload describes the behavior of all users of the information system. It consists of:

1. Workload parameter set
2. Application programs
3. Operating system command procedures
4. Computational results
5. User data stored in the SUT
6. Parameters for controlling the correct work of the RTE

Steps of the measurement:

1. Install applications in the SUT
2. Load workload parameter set into RTE
3. Run and record; store computational results
4. Testing correctness
   (RTE: correct work and statistical significance of random variables;
   SUT: correct and complete computational results)
5. Analysis of recorded data and computation of performance values and rating values

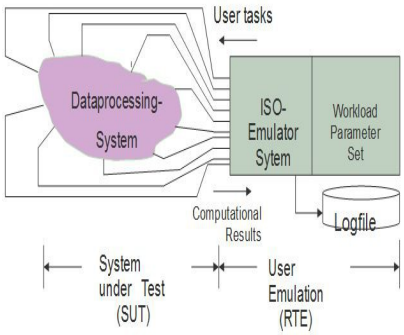


Fig. 3 Measurement Configuration

4 The workload parameter set

Basic parameters:
- Number n of user types
- Number of users of each type $N_{user}(1),..., N_{user}(n)$
- Number w of activity types
- Number p of timeliness functions
- Number m of task types
- Number u of task chain types

Definitions of the w activity types, each described by :
- Input string or mouse pointer action, rules for input variation if there is so.

Definitions of the m task types, each defined by:
- Activity type and wait mode WAIT/NOWAIT for the result of the actual task and
- timeliness functions for completing the task (Example: Maximal 80% within 2 sec, maximal 15% within 6 sec, maximal 5% within 6 sec, none longer than 15 sec.)

Definitions of the u task chain types:
- Chain length (number of tasks) and task sequence

Statistic parameters of the (random to be created) think times of the users
- Firstly: matrix of n x m think time mean values
  (Remark: Think time is task preparation time.)
- Secondly: matrix of n x m think time standard deviation values

5 The measurement run

RTE on
Stabilization phase
Rating interval
Supplementary run
RTE off

6 The ISO performance P computed from the recorded logfile

P is the following triple of vectors: $P = (B, T_{ME}, E)$

where $B = (B(1), …., B(m))$ is the (total) throughput vector.

B(j) is the mean number of tasks of the j-th task type sent from the RTE to the SUT per time unit.

$T_{ME} = (T_{ME}(1),..., T_{ME}(m))$ is the execution time vector .

$T_{ME}(j)$ is the mean execution time of tasks of the j-th task type.

$E = (E(1), …., E(m))$ is the (timely) throughput vector.

E(j) is the mean number of tasks of the j-th task type which were timely executed by the SUT per time unit.

7 Rating of the measured performance values

Throughput rating

$R_{TH} = (R_{TH}(1),..., R_{TH}(m))$ is the throughput rating vector
with
$R_{TH}(j) = B(j) / B_{Ref}(j)$
$B_{Ref}(j)$ is the throughput of the j-th task type of the so called theoretical reference machine.

Mean execution time rating	$R_{ME} = (R_{ME}(1),..., R_{ME}(m))$ is the execution time rating vector with $R_{ME}(j) = T_{Ref}(j) / T_{ME}(j)$ $T_{Ref}(j)$ is the mean execution time of tasks of the j-th task type. of the so called theoretical reference machine.
Timely throughput rating	$R_{TI} = (R_{TI}(1),..., R_{TI}(m))$ is the timely throughput rating vector with $R_{TI}(j) = E(j) / B(j)$

The "theoretical reference machine" is a fictional SUT. It is a not really existing SUT which executes all tasks so fast that all timeliness functions are just fulfilled but none faster. The $T_{Ref}(j)$ values and $B_{Ref}(j)$ values can be computed directly from the data of the workload parameter set. No measurement is needed.

Overall rating: Only if all of the 3 x m rating values are not less 1 the SUT satisfies the timeliness requirements of the user entity. Elsewhere the system has to be rejected due to insufficient response times.

8 Measurement example 1

The diagrams below (see Fig 4 and 5) show the measured performance and the rating of two mainframe computers having the same hardware. Workload: "ISO CC1-I-REP=1" (somewhat modified). The application software is identical. The operating systems are different.

Terms in the diagrams:

Leistungsgrößen –---------- Performance terms
Belastung ----------– Throughput B
Aufträge/sec ---------- Tasks per sec
Auftragsdurchlaufzeit --------- Execution time
AAx ---------- Task type x
Anzahl der EAG ---------- Number of emulated users
Bewertungsgrößen ---------- Performance rating terms

$t_{AN}(j)$ - Mean execution time $T_{ME}(j)$ of the j-th task type
L1(j) - Throughput rating value $R_{TH}(j)$ of the j-th task type
L2(j) - Mean execution time rating value $R_{ME}(j)$ of the j-th task type
L3(J) - Timely throughput rating value of the j-th task type

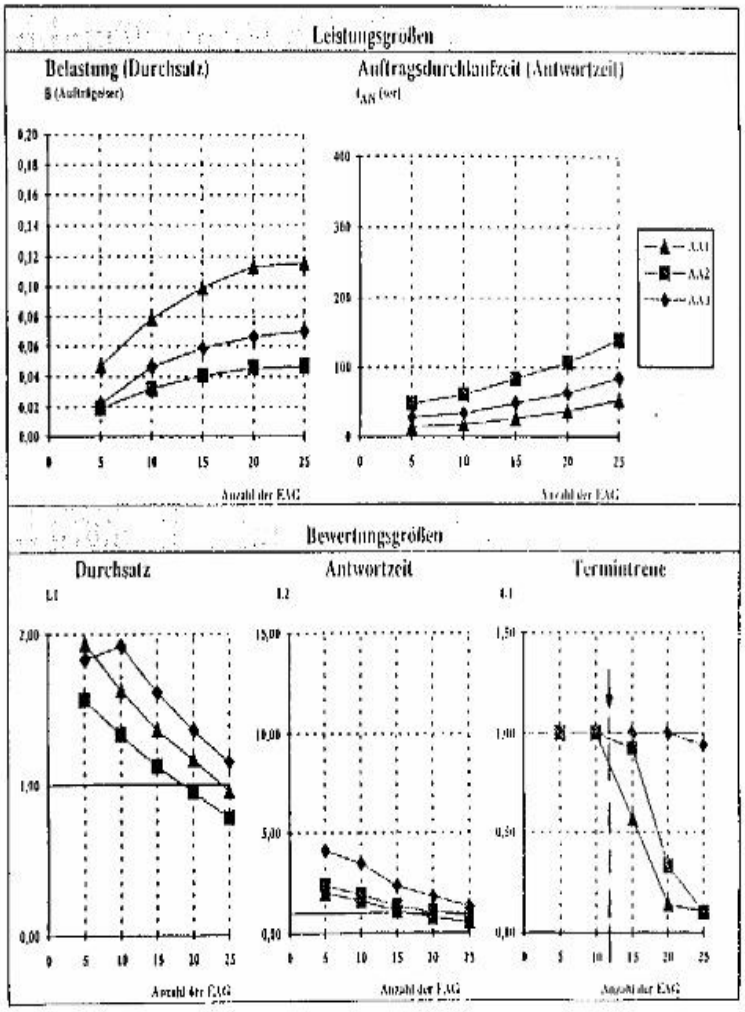


Fig. 4 Mainframe-Nr. 1, Hardware: 8 MIPS
Operating system from "Manufacturer 1"

For more than 12 users is the system no longer timely:
$N_{max} = 12$

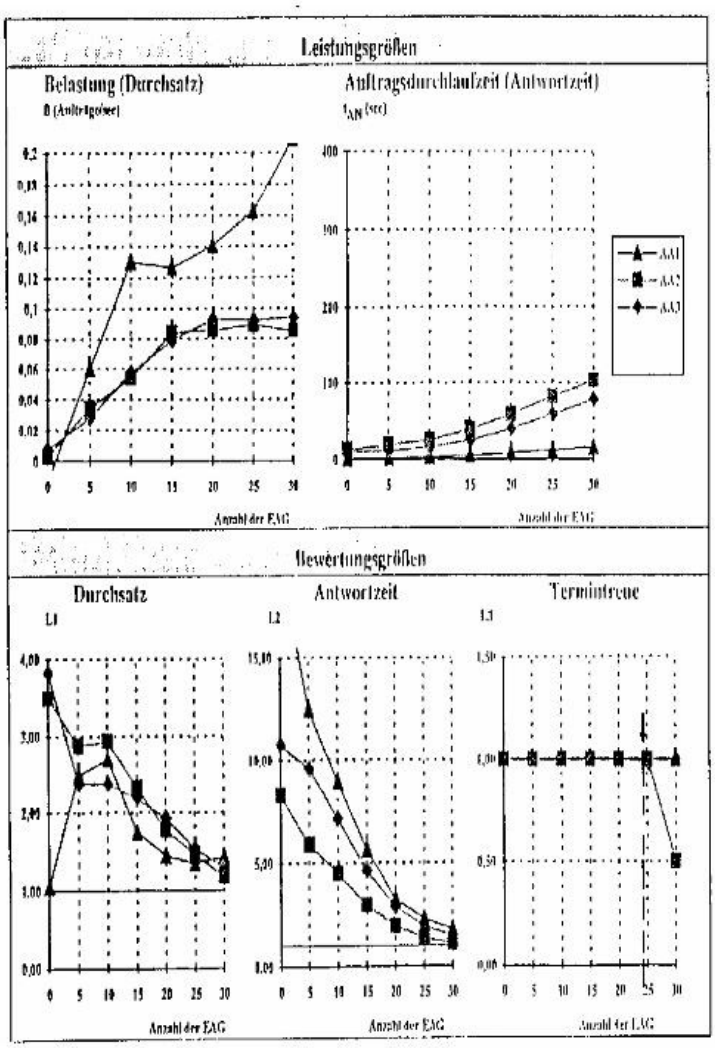


Fig. 5 Mainframe-Nr. 2
Hardware: Same as in Fig. 4 and also 8 MIPS
Operating system: Changed to those of"Manufacturer 2"

For more than 24 users is the system no longer timely:
$N_{max} = 24$

9 Software efficiency in example 1

SW efficiency has several qualities, for instance

- storage usage
- changeability
- maintainability
- runtime

The focus of ISO 14756 is runtime efficiency. Wanted here is the operation systems software efficiency.

The ISO standard 14756 describes the necessity of the reference environment (see Fig.6) in detail. It also describes how to define it for measuring software efficiency. But it does not explain detailed how to compare the measured performance values P1 and P2. To the author's knowledge the ISO working group – which developed the standard 14756 - took it (at that time, i. e. when the Standard was written) for granted how to compare two performance values P1 and P2.

The following comparison of P1 and P2 is done according to a proposal explained in [3] and [4] in detail.

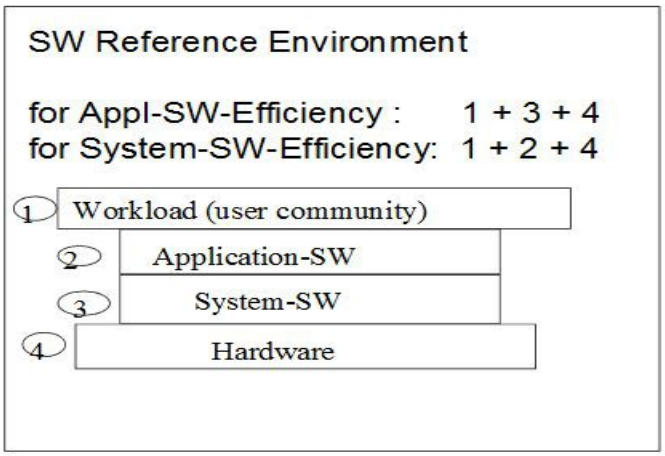


Fig. 6 The reference environment (simplified)

The example: We look for the software efficiency of the two measured systems above(see Fig. 4 and Fig 5).:

The environment is " 1 + 2 + 4" .

The measured terms are:

System 1: $P1 = (B1, T_{ME}1, E1)$
P1 represents 3xm values:

$B1 = (B1(1), \ldots\ldots\ldots\ldots, B1(3))$
$T_{ME}1 = (T_{ME}1 (1)),\ldots\ldots, T_{ME}1(3))$
$E1 = (E1(1), \ldots\ldots\ldots\ldots, E1(3))$

System 2: $P2 = (B2, T_{ME}2, E2)$
P2 represents 3xm values:

$B2 = (B2(1), \ldots\ldots\ldots\ldots, B2(3))$
$T_{ME}2 = (T_{ME}2 (1)),\ldots\ldots, T_{ME}2(3))$
$E2 = (E2(1), \ldots\ldots\ldots\ldots, E2(3))$

The software efficiency values computed from the measured values are:

Throughput efficiency values		(for n=10 users)	(for n=15 users)
Task type 1	$I_{TH}(1)= B2(1) / B1(1)$	0.130 / 0.080 = 1.63	0.125 / 0.100= 1.25
Task type 2	$I_{TH}(2)= B2(2) / B1(2)$	0.058 / 0.035 = 1,66	0.080 / 0.410 = 1.95
Task type 3	$I_{TH}(3)= B2(3) / B1(3)$	0.058 / 0.450 = 1,29	0.081 / 0.060 = 1,35

Mean execution time efficiency values		(for n= 10 users)	(for n=15 users)
Task type 1	$I_{ME}(1)= T_{ME}1(1) / T_{ME}2(1)$	14.5 / 2.10 = 6.90	20 / 4.5 = 4.44
Task type 2	$I_{ME}(2)= T_{ME}1(2) / T_{ME}2(2)$	61.0 / 21.9 = 2.78	80 / 20.5 =3.90
Task type 3	$I_{ME}(3)= T_{ME}1(3) / T_{ME}2(3)$	39.0 / 13.5 = 2.89	50 / 26 = 1.87

Timely efficiency values		(for n= 10 users)	(for n=15 users)
Task type 1	$I_{TI}(1)= E2(1) / E1(1)$	0.130 / 0.080 = 1.51	0.125 / 0.060 = 2.08
Task type 2	$I_{TI}(2)= E2(1) / E1(2)$	0.058 / 0.035 = 1.66	0.080 / 0.038 = 2.34
Task type 3	$I_{TI}(3)= E2(3) / E1(3)$	0.058 / 0.045 = 1.23	0.080 / 0.060 = 1.43

We see: For n =10 users are the throughput oriented efficiency values around 1.5 and the response time oriented efficiency values nearly at 3. For n =15 users are the throughput oriented efficiency values around 2 and the response time oriented efficiency values nearly at 4.

Remarks: 1) Be aware that n=10 users is not the same system software environment as n=15 users !
2) Due to the fact that the lowest N_{max} value of the compared two systems is 12 it is really meaningless to investigate the software efficiency for more than 12 users.

There is an additional software efficiency measure:

$$I_{maxuser} = N_{max}2 / N_{max}1$$

Its value is $24 / 12 = 2$.

Final result:
The operating system of manufacturer 2 is about two times more efficient that of manufacturer 1 when driving a computer center operation as represented by the ISO workload CC1. I. e.: When using the operating system 2 the computer system serves about 100% more users timely.

That was very surprising and was opposite to the meaning of the system programmers of both companies. They were up to this measurement convinced that there would be nearly no difference in the software efficiency of these two mainframe operating systems.

10 Example 2 and software efficiency

The hardware and operating systems are the same as in example 1. But the application software is SAP/R2 instead of the software contained in the (modified) ISO workload CC1 of example 1.

Used were 4 parts of the SAP software:

RF (Finances)	: 40% of the users
RM-MAT (Materials management)	: 30% of the users
RM-PPS (Production)	: 10% of the users
RV (Sales)	: 20% of the users

An ISO type workload was developed. The workload parameter set (shortened):

4 user types
110 activity types
110 task types
21 chain types
3 timeliness functions (mean values 3, 6 and 36 seconds)

Measured N_{max} values:

Operating system 1: $N_{max}1 = 110$
Operating system 2: $N_{max}2 = 170$

Software efficiency

$$I_{maxuser} = N_{max}2 / N_{max}1 = 170 / 110 = 1.55 \quad .$$

Using the operating system 2 instead of operating system 1 the computer system serves (about) 55% more users timely.

11 ISO 14756 and simulated software

The ISO method is not only applicable to real computer systems and its software. It is also applicable for instance to simulated software. Therefore its use is recommended from the first steps of a software project up to the running software in a real computer center operation, and further on in the maintenance phase, for instance when repairing defects or when developing new releases. This is an opportunity to realize high software efficiency (as one of the basic criteria of software quality) along the total life time of the software from its birth to its death. And – even – it can be a help in the first steps of an eventually following project.

A special point of interest is saving man power. Nowadays the practice is to write ad hoc (and highly uncoordinated) new individual test systems for software efficiency whenever the question comes up. Opposite to this it is recommended to integrate this question from the beginning of a software project. This can be done by specifying the workload as an ISO workload which has to become central point of the project specification from the beginning of the project. The workload has to be designed only once and an ISO RTE is to buy or to write only once. The pair workload-RTE can be used by all members and in all phases of the software project. Only and only than when the specification of the software project changes the ISO type workload has to be adapted.

12 Final remarks

The ISO Standard 14756 defines a very universal method for measuring computer systems performance and software (runtime-) efficiency.

A RTE which is implemented according to this standard represents a universal high precision measurement tool and it delivers reproducible measurement results (compare chapter 2).

Implementations of an ISO RTE were realized for instance by the Siemens Nixdorf Computer Company, Telekom and others. To the authors knowledge these companies didn't offer their tools in the market for sale. Free demo versions of an ISO RTE are found on the CD which is part of [4]. An ISO 14756 conform RTE offered in the market is for instance "S_aturn" (see [5]).

The ISO method is not only applicable to real computer systems and its software. It is also applicable for instance to simulated software. Therefore its use is recommended from the first steps of a software project. It is a chance to save man power and to improve the software efficiency of the delivered software product. The ISO workload principle is an outstanding template to describe the endusers demands on his computer center and the installed software.

References

[1] DIN 66273-Serie, *Messung und Bewertung der Leistung von DV-Systemen,* Deutsches Institut für Normung, 10772 Berlin, 1991

[2] ISO/IEC, *International Standard 14756*, *Information technology – Measurement and rating of performance of computer-based software systems*, ISO Copyright Office, CH 1211 Geneve 20, Casa Portale 56. 1999

[3] W. Dirlewanger, *Messung und Bewertung der DV-Leistung auf Basis der Norm DIN 66273*, Hüthig-Verlag, ISBN: 3-7785-2147-0, 1994

[4] W. Dirlewanger", *Measurement and Rating of Computer Systems Performance and of Software Efficiency. An Introduction to the ISO/IEC 14756 Method and a Guide to its Application*, ISBN-10: 3-89958-233-0, ISBN-13: 078-3-89958-239-8, Kassel University Press, 2007 (free extract: http://www.upress.uni-kassel.de)

[5] http://www.zott.net

Technical Open Challenges on Benchmarking Workflow Management Systems

Marigianna Skouradaki, Dieter H. Roller, Frank Leymann

Institute of Architecture and Application Systems
University of Stuttgart
Germany
{skouradaki, dieter.h.roller, leymann}@iaas.uni-stuttgart.de

Vincenzo Ferme, Cesare Pautasso
Faculty of Informatics
University of Lugano
Switzerland
{firstname.lastname@usi.ch}

Abstract: The goal of the BenchFlow project is to design the first benchmark for assessing and comparing the performance of BPMN 2.0 Workflow Management Systems (WfMSs). WfMSs have become the platform to build composite service-oriented applications, whose performance depends on two factors: the performance of the workflow system itself and the performance of the composed services (which could lie outside of the control of the workflow). Our main goal is to present to the community the state of our work, and the open challenges of a complex industry-relevant benchmark.

1 Introduction

This paper is an introduction to the Benchflow project[1]. The goal of this project is to design the first benchmark for assessing and comparing the performance of BPMN 2.0[2] Workflow Management Systems (WfMSs). WfMSs have become the platform to build composite service-oriented applications (SOA), whose performance depends on two factors: the performance of the workflow system itself and the performance of the composed services (which could lie outside of the control of the workflow). Therefore the development of such a benchmark, reveals a number of challenges that were not present in benchmark of other types of systems, such as database management systems, or programming language compilers.

In this work we present a list of challenges that are recognized during the design of such a benchmark. We present the current state of our work, as well as the design decisions taken so far, and also discuss the open challenges.

[1] http://www.iaas.uni-stuttgart.de/forschung/projects/benchflow.php
[2] http://www.bpmn.org/

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

2 Description of the Workflow Management System environment

The purpose of this section is to provide to the reader a high-level understanding of the WfMSs. For that we first need to clarify the concept of "workflow", which is basically, the focus point of a WfMS. According to [Spe99] a "workflow is the computerized facilitation or automation of a business process, in whole or part". In other words, a workflow is the automation of a series of business activities that are needed for achieving a goal. In this respect a Workflow Management System (WfMS) is a system that supports the reengineering and automation of business and information processes. Its main characteristics are the definition of workflows, and the provision of fast re(design) and re(implementation) as the business needs change [GHS95]. All the components of the WfMS, are compliant with the semantics of a *metamodel* that defines concepts such as the structure of a process and the operations that can be performed on a process model instance [LR00].

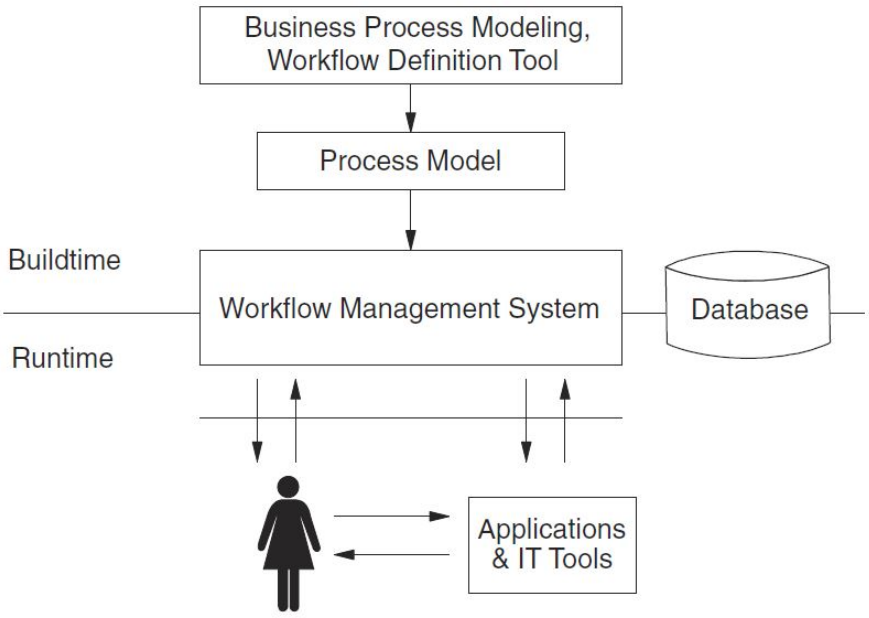


Figure 1: Major Components of a Workflow Management System [LR00]

In Figure 2 we can see the main design and run-time components of a WfMS:

Buildtime provides means to the user to define the constructs such as process models, organizational structures, and information technology aspects (e.g. scripts, programs).
Workflow Definition Tool is the tool with which the users define the workflow according to a workflow language that is compliant with the WfMS.
Process Model is an intermediate representation [Ley10] of the previously defined workflow, in a format that the WfMS can process. Process Models can be expressed through

different definition languages. For many years there was no standard definition language, and each company chose to implement a customized form of the stardard definition languages on their engines. WS-BPEL [Org07] became an industry-accepted standard for executing processes over WSDL-based Web services. Later, it was followed by the creation and industrial acceptance of the BPMN 2.0 standard [JE11]. This work focuses on benchmarking WfMSs that support BPMN 2.0.
Runtime is the part that performs the actual execution of the processes, featuring operations, such as creating, navigating and controlling processes.
Database of the WfMS keeps and handles all the information that are essential for the build and runtime. Thus, in the database we find stored the process models, and also the state of their instances.
User and IT Tools the user (either the owner of a workflow or a participant in it) interacts with the WfMS to report about the progress of his or her tasks, which may be performed directly or through applications and other IT tools.

Currently, WfMSs can be seen as one of the key middleware components of Service Oriented Architectures (SOA). As human users are less and less part of the workflow execution, whereby many different, possibly geographically distributed, service components interact to provide the corresponding functionality, under the control of the WfMS [ZDGH05, GKW+12].

3 Open Technical Challenges

Given the peculiar characteristics of WfMSs, in the following we identify and briefly discuss some of the technical challenges in benchmarking WfMSs.

Automate the generation of realistic workload for different use case scenarios In order to keep our measurements accurate we need to create realistic workload scenarios. As workflow-based applications are currently present in various types of application domains[LR97], it is challenging to select a sufficiently large subset of domains and synthesize a domain-independent workload. In order to address this challenge we have started designing a workload generator for creating process models according to pre-defined criteria. The workload generator needs to take into consideration a set of variables that are related to the correct operation of the workflows, e.g. data-flow, key performance indicators (KPI), actors, and interactions with external entities (e.g. Web Services, databases, external scripts).

The setup of the benchmark environment During the setup of the benchmark environment the challenge is to eliminate the interference of non-WfMS resources as for example database systems, and Web Services used by a process. In order to do that, we design the benchmark environment to be distributed on different physical machines that are connected through the same local network. To do this we need to ensure a flexible deployment mechanism.

System Internal Load Optimization The request load to WfMS is different for the various day times. In order to handle this situation the WfMS shifts work to the daytime where

the load is lower. The challenge in this case is to take into account these optimizations in the measurements of the throughput.

Benchmark Long Running Processes The lifetime of a process can span from some milliseconds to hours, weeks, or even some years. The processes that run for more than some hours are called "long-running" processes. An example of a long running process might be a process that sends a message to an external partner and needs to wait for weeks or even years for the reply. The long-running processes have two main characteristics:

1. the storage of the instances in database increases in size and
2. caching cannot be used from either the WfMS or the DBMS

The challenge in this case is to find a method to benchmark this type of processes without having to wait for years for the completion of the benchmark.

Performance Impact of Workflow Language Features BPMN 2.0 provides a large set of constructs, that express iteration, parallelism, exception handling, interactions with external entities etc. According to a research among the Websites and release notes of the currently available engines, we have observed that the documentation on the coverage of BPMN 2.0 constructs is usually inadequate. There is also the hypothesis that WfMSs avoid implementing all of the constructs for performance reasons. It is also important to consider which of these constructs are actually used in real-world processes [MR08]. The challenge in this case is to: a) run compliance tests to define which constructs are actually supported by the available WfMSs, and b) decide which constructs need to go into the workload mix.

4 Project Status

The initial phase of the project focused on analyzing a large collection of process models, for collecting data to proceed to workload characterization and generation. The set up of the benchmark environment is also planned at this phase.

Currently we have collected a large collection of 8363 real-life process models that are expressed in various modelling languages. Collecting process models can be a big challenge as companies are not willing to share them to protect their corporate assets. The ways by which we countenanced the process model sharing was to sign confidentiality agreements, and to develop tools for obfuscating the models [SRPL14], and thus protecting the company's intellectual property. Our collection contains: 1% WS-BPEL, 4% EPC, 7% YAWL, 24% Petri Net, and 64% BPMN & BPMN 2.0 Models, where 2/3 of them are BPMN 2.0. These data justify our decision to focus on the engines that support the BPMN 2.0 standard, as it seems to have become widely accepted in industry and academia.

However, the focus on the BPMN 2.0 standard was not the only decision that was taken with respect to the data derived from the analysis of the process models collection. We have currently ran statistical analysis on more than hundred different metrics, observing process models characteristics (e.g. control flow gateways fan-in, fan-out, cyclomatic complexity), and the frequency of appearance of the BPMN 2.0 elements in the collected

process models. This led to a better understanding of which BPMN 2.0 features are present in the real world, and thus would need to be supported by the engines participating in the benchmark. We also arrived at the definition of representative process model mockups, generated based on the results of a cluster analysis over structural complexity, and language construct usage metrics.

The statistical analysis that we did on the process models gave us information about the model characteristics at a "micro" level. We also wanted to define complete structures that are repeatedly found in the process models. By this way we can combine different structures with each other and have an even more realistic workload. The problem of discovering the reoccurring structures is reduced to "Frequent Pattern Discovery", and it is NP-complete [Ben02]. To address this challenges we have implemented an extension of the VF-2 algorithm which is said to be one of the most efficient algorithms in graph isomorphism [PCFSV04]. The exported structures must now be characterized according to benchmark-related criteria, and combined with respect to the metrics found in the aforementioned statistical analysis. The idea is to give all the acquired information as an input to a workload generator, and create realistic workload, that is compliant for different benchmarking scenarios.

The choice BPMN 2.0 engines to be included in the benchmark is also a decision that needs to be taken at this phase of the project. The current state is that there are more than 20 engines available implementing the BPMN 2.0 Standard. We are currently collecting the available engines, and relevant information in a Wikipedia page[3]. Unfortunately, the lack of documentation in terms of the engine's compliance to the BPMN 2.0 standard, is currently delaying our decision. In order to find out this information we are contacting the vendors asking for information, and conducting compliance tests towards the engines. The aforementioned statistical analysis also plays its roles at this point, as we need to choose among these engines which implement at least the most frequently used elements of the BPMN 2.0 Standard.

In terms of the benchmark set-up we developed and deployed the first prototype of a WFMS on different physical machines connected through the same local network. The challenges that we needed to address are currently solved using Docker[4] that offers a flexible deployment mechanism with a minimal impact on the performance measurement. Docker guarantees a good level of isolation and a quick start up, and enables repeatability of the tests because the initial conditions are kept inside the Docker containers and are exactly the same for each execution of the benchmark.We are currently using Faban[5] to develop the first prototypes of benchmark drivers, we are running the first experiments using the three models we derived from the statistical analysis of a large model collection, and we are analyzing the execution log to compute the first KPIs such as throughput.

[3] http://en.wikipedia.org/wiki/List_of_BPMN_2.0_engines
[4] https://www.docker.com
[5] http://www.faban.org

5 Related Work

In the topic of benchmarking the performance of WfMSs, only some work is available (e.g., [GMW00, BBD10, DPZ11]). In the most recent approaches, the SOABench project [BBD10] can be seen as an initial step to provide a performance assessment and comparison framework for SOA middleware systems.

SPEC has also introduced a group that focuses on benchmarking SOA infrastructures [Sta10]. However information on their progress of research is still not available on the website. Vendors of proprietary systems are also executing internal benchmarks (e.g. [SAP11], [Inc11], [IC07]) in order to evaluate their work and inform the prospective customers. However, there is not any standard method followed in these becnhmark approaches. As easily concluded, and also emphasized by the literature, the need to introduce a benchmark standard for that addresses the industry state-of-art needs is now imperative [KKL06, WLR+09, RvdAH07, LMJ10]. Our work intends to the creation of such a standard benchmark, that differs from the related work, in terms of: a) the number of WfMSs to be compared, b) the complexity and diversity of the workload mix, c) the number of the executed performance tests, and d) the number of performance metrics that will be taken into consideration, and their aggregation into a meaningful number.

6 Discussion & Conclusion

In this paper we have presented the open challenges for the creation of a benchmark for WfMSs that we have been addressing in the first phase of the BenchFlow project as well as some solutions for addressing them. We have seen how the synthesis of the workload mix needs to be created through a workload generator, how we plan to setup the benchmark environment and isolate it from internal interferences, and how the compliance tests and statistics on BPMN 2.0 support are important before deciding the workload mix. We have also raised issues such us long-running processes benchmarking, and the consideration of WfMS internal load optimization into our benchmark.

By presenting this set of open research challenges in this position paper, we aim to present the SOSP community with a complex industry-relevant benchmarking challenge and discuss how to deal with the aforementioned open challenges.

References

[BBD10] D. Bianculli, W. Binder, and M.L. Drago. SOABench: Performance Evaluation of Service-Oriented Middleware Made Easy. In *Proc. of the 19th International World Wide Web Conference (WWW 2010)*, Raleigh, NC, 2010.

[Ben02] E. Bengoetxea. *Inexact Graph Matching Using Estimation of Distribution Algorithms*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec 2002.

[DPZ11] Florian Daniel, Giuseppe Pozzi, and Ye Zhang. Workflow Engine Performance Evaluation by a Black-Box Approach. In *Proc. of the International Conference on Informatics Engineering & Information Science*, ICIEIS 2011, pages 189–203. Springer, November 2011.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3(2):119–153, 1995.

[GKW+12] Spyridon Gogouvitis, Kleopatra Konstanteli, Stefan Waldschmidt, George Kousiouris, Gregory Katsaros, Andreas Menychtas, Dimosthenis Kyriazis, and Theodora Varvarigou. Workflow management for soft real-time interactive applications in virtualized environments. *Future Generation Computer Systems*, 28(1):193 – 209, 2012.

[GMW00] Michael Gillmann, Ralf Mindermann, and Gerhard Weikum. Benchmarking and Configuration of Workflow Management Systems. In *Proc. of the 7th International Conference on Cooperative Information Systems*, CoopIS 2000, pages 186–197, 2000.

[IC07] Intel and Cape Clear. BPEL Scalability and Performance Testing. White paper, 2007.

[Inc11] Active Endpoints Inc. Assessing ActiveVOS Performance, 2011.

[JE11] Diane Jordan and John Evdemon. Business Process Model And Notation (BPMN) Version 2.0. Object Management Group, Inc, January 2011.

[KKL06] Rania Khalaf, Alexander Keller, and Frank Leymann. Business processes for Web Services: Principles and applications. *IBM Systems Journal*, 45(2):425–446, 2006.

[Ley10] Frank Leymann. BPEL vs. BPMN 2.0: Should You Care? In Jan Mendling, Matthias Weidlich, and Mathias Weske, editors, *Business Process Modeling Notation*, volume 67 of *Lecture Notes in Business Information Processing*, pages 8–13. Springer Berlin Heidelberg, 2010.

[LMJ10] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web*, 4(1):2:1–2:33, January 2010.

[LR97] Frank Leymann and Dieter Roller. Workflow-Based Applications. *IBM Systems Journal*, 36(1):102–123, 1997.

[LR00] Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[MR08] M. Muehlen and J. Recker. How much language is enough? Theoretical and practical use of the business process modeling notation. In *Advanced Information Systems Engineering*, pages 465–479. Springer, 2008.

[Org07] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, April 2007.

[PCFSV04] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, October 2004.

[RvdAH07] N. Russell, W.M.P. van der Aalst, and A.H.M. Hofstede. All That Glitters Is Not Gold: Selecting the Right Tool for Your BPM Needs. *CUTTER IT JOURNAL*, 20(11):31–38, November 2007.

[SAP11] IBM SAP. SAP NetWeaver Business Process Management Performance, Scalability, and Stability Proof of Concept. Technical report, IBM SAP International Competence Center (ISICC)Walldorf, Germany SAP NetWeaver BPM Product Management, Walldorf, Germany SAP Labs, Berlin, Germany SAP Active Global Support, Walldorf, Germany, 2011.

[Spe99] Workflow Management Coalition Specification. *Workflow Management Coalition, Terminology & Glossary (Document No. WFMC-TC-1011)*. Workflow Management Coalition Specification, February 1999.

[SRPL14] Marigianna Skouradaki, Dieter Roller, Cesare Pautasso, and Frank Leymann. BPELanon: Anonymizing BPEL Processes. In *Proc. of ZEUS'14*, pages 09–15, 2014.

[Sta10] Standard Performance Evaluation Corporation. SPEC SOA Subcommittee, February 2010. `http://www.spec.org/soa/`.

[WLR+09] B. Wetzstein, P. Leitner, F. Rosenberg, I. Brandic, S. Dustdar, and F. Leymann. Monitoring and Analyzing Influential Factors of Business Process Performance. In *Proc. of the IEEE International Conference on Enterprise Distributed Object Computing Conference*, EDOC 2009, pages 141 –150, September 2009.

[ZDGH05] Olaf Zimmermann, Vadim Doubrovski, Jonas Grundler, and Kerard Hogg. Service-oriented Architecture and Business Process Choreography in an Order Management Scenario: Rationale, Concepts, Lessons Learned. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 301–312, New York, NY, USA, 2005. ACM.

Evaluating the Prediction Accuracy of Generated Performance Models in Up- and Downscaling Scenarios

Andreas Brunnert[1], Stefan Neubig[1], Helmut Krcmar[2]
[1]fortiss GmbH
Guerickestr. 25, 80805 München, Germany
{brunnert, neubig}@fortiss.org
[2]Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
krcmar@in.tum.de

Abstract: This paper evaluates an improved performance model generation approach for Java Enterprise Edition (EE) applications. Performance models are generated for a Java EE application deployment and are used as input for a simulation engine to predict performance (i.e., response time, throughput, resource utilization) in up- and downscaling scenarios. Performance is predicted for increased and reduced numbers of CPU cores as well as for different workload scenarios. Simulation results are compared with measurements for corresponding scenarios using average values and measures of dispersion to evaluate the prediction accuracy of the models. The results show that these models predict mean response time, CPU utilization and throughput in all scenarios with a relative error of mostly below 20 %.

1 Introduction

Numerous performance modeling approaches have been proposed to evaluate the performance (i.e., response time, throughput, resource utilization) of enterprise applications [BDMIS04, Koz10, BWK14]. These models can be used as input for analytical solvers and simulation engines to predict performance. Performance models are especially useful when scenarios need to be evaluated that cannot be tested on a real system. Scaling a system up or down in terms of the available hardware resources (e.g., number of CPU cores) are examples for such scenarios.

Evaluating the impact of up- or downscaling on performance is a typical activity during the capacity planning and management processes. Capacity planning concerns questions such as "How many hardware resources are required for the expected workload of new enterprise application deployments?" and involves evaluating the behavior of an application when a system is scaled up. Capacity management on the other hand is usually concerned with evaluating whether the existing hardware resources are sufficient for the current or expected load. This involves not only upscaling but also downscaling scenarios in which the amount of hardware resources needs to be reduced to save costs (e.g., license fees that depend on the number of CPU cores used).

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

Nowadays, creating a performance model requires considerable manual effort [BVD+14]. This effort leads to low adoption rates of performance models in practice [Koz10]. To address this challenge for Java Enterprise Edition (EE) applications, we have proposed an automatic performance model generation approach in [BVK13]. This work improves the existing approach by further reducing the effort and time for the model generation.

In order to evaluate whether the automatically generated performance models are fit for use during capacity planning and management, we evaluate the improved model generation approach in up- and downscaling scenarios. In a first step, an automatically generated performance model is used to predict the performance of a system in an upscaling scenario, in which additional CPU cores are added to the system. Afterwards, a downscaling scenario is evaluated in which the number of CPUs is reduced. During the evaluation of the up- and downscaling scenarios not only the number of CPU cores is modified, but also the amount of users interacting with the system simultaneously.

2 Generating Performance Models

This section is based on our previous work on generating performance models for Java EE applications [BVK13]. In this work, we are using the same concepts for the model generation but reduce the time required for generating a performance model to mostly less than a minute. To make this work self-contained, a brief overview of the model generation process is given, changes are described in more detail. The model generation process is divided into a data collection and a model generation step, the explanation follows these two steps.

2.1 Data Collection

The data that needs to be collected to create a representative performance model is dependent on which components should be represented in the model [BVK13]. Following Wu and Woodside [WW04], Java EE applications are represented using the component types they are composed of. The main Java EE application component types are Applets, Application Clients, Enterprise JavaBeans (EJB) and web components (i.e., JavaServer Pages (JSP) and Servlets) [Sha06]. As Applets and Application Clients are external processes that are not running within a Java EE server runtime, the remainder of this paper focuses on EJB and web components. To model a Java EE application based on these component types, the following data needs to be collected [BVK13]:

1. EJB and web component as well as operation names
2. EJB and web component relationships on the level of component operations
3. Resource demands for all EJB and web component operations

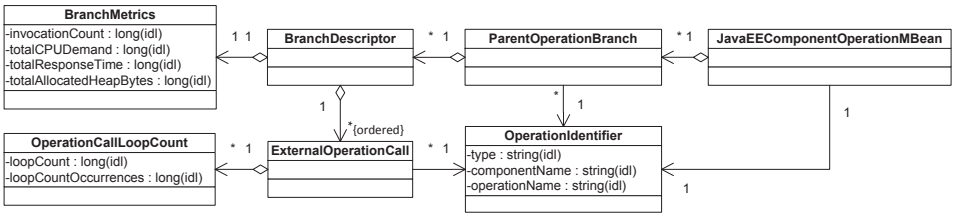


Figure 1: JavaEEComponentOperationMBean data model

In [BVK13] we have collected this information using dynamic analysis, saved it in comma-separated value (CSV) files and used an additional process step to aggregate this information into a database. To speed up the model generation process we are no longer using files as persistence layer and have removed the additional step of aggregating the data stored in the files in a relational database. Instead, the data required for modeling an application is collected and aggregated in Managed Beans (MBeans) [Mic06] of the Java Virtual Machine (JVM). MBeans are managed Java objects in a JVM controlled by an MBean server.

The reason for choosing MBeans as persistence layer is that the Java Management Extension (JMX) specification defines them as the standard mechanism to monitor and manage JVMs [Mic06]. The JMX and related specifications also define ways to manage, access and control such MBeans locally as well as from remote clients. For example, the JMX remote application programming interface (API) allows access to all MBeans of a system remotely using different network protocols. Building upon the JMX standard therefore ensures that the approach is applicable for all products that are compliant with the Java EE [Sha06] and JMX [Mic06] specifications.

One of the key challenges for the transition from CSV files to MBeans is to find a data model with low impact on an instrumented system. The instrumentation for the dynamic analysis collects structural and behavioral information as well as resource demands for each component operation invocation. As storing the data for each invocation separately in an MBean is not possible due to the high memory consumption, the data needs to be aggregated. Additionally, recreation of existing control flows from the data needs to be possible. To accompany these requirements and to implement the MBean data collection with low impact on the monitored system, the data model shown in figure 1 is used.

A *JavaEEComponentOperationMBean* is registered for each externally accessible component operation. Internal component operations are not represented in the data model. Each *JavaEEComponentOperationMBean* instance is identified by an *OperationIdentifier* attribute. The *OperationIdentifier* is a unique identifier of a component operation in a Java EE runtime. It therefore contains the respective *componentName* (i.e., EJB or web component name), the component *type* (i.e., Servlet/JSP/EJB) and its *operationName* (i.e., Servlet/JSP request parameter or EJB method name).

The component relationships are also stored in the data model. These relationships differ depending on component states as well as input and output parameters of their operations (i.e., whether an external operation is called or not). To simplify the data model, com-

ponent states and parameters of component operations are not represented. Instead, the invocation counts for different control flows of a component operation are stored in the model.

A control flow of a component operation is represented by the *BranchDescriptor* class and its ordered list of *ExternalOperationCalls*. *ExternalOperationCalls* are identified using an *OperationIdentifier* attribute. Using this attribute, *ExternalOperationCalls* can be linked to the corresponding *JavaEEComponentOperationMBeans*. This link allows recreating the complete control flows of requests processed by applications in a Java EE runtime.

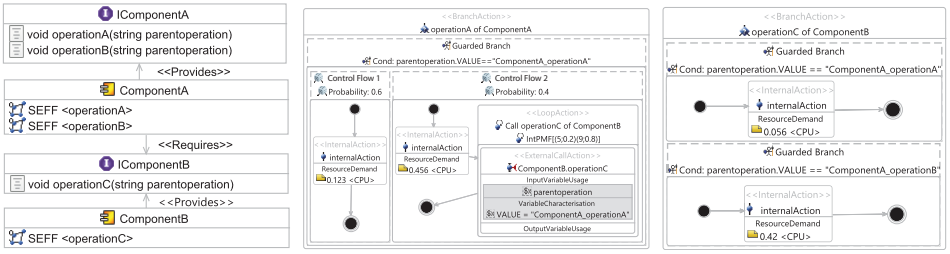
ExternalOperationCalls have an additional *OperationCallLoopCount* attribute, which is used to track the number of times an external operation is called in a row. This attribute helps to limit the amount of data that needs to be stored in the MBeans, as repeating invocations do not need to be stored separately. Instead, each *loopCount* that may occur in an otherwise equal control flow can be tracked using the same data structure. For each *loopCount*, the *OperationCallLoopCount* class stores the number of times a *loopCount* occurred in the *loopCountOccurrences* attribute.

A *BranchDescriptor* also contains a *BranchMetrics* attribute, which tracks the number of times a control flow occurred (*invocationCount*) and how much CPU, heap and response time is consumed by the current component operation in this control flow in total. This information allows calculating the control flow probability and its resource demand during the model generation.

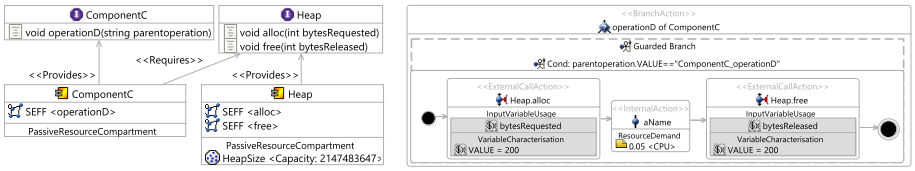
To differentiate requests processed by applications in a Java EE runtime, control flows of an operation are grouped according to the component operation that is invoked first during a request (i.e., by users or external systems). This grouping is specified in the *ParentOperationBranch* class. It maps a list of *BranchDescriptors* to a *JavaEEComponentOperationMBean* and contains a reference to the *OperationIdentifier* of the first operation. This reference improves the data collection and model generation performance.

2.2 Performance Model Generation

The data stored in the MBean data model (see figure 1) is used to generate component-based performance models. The meta model for the generated models is the Palladio Component Model (PCM) [BKR09]. PCM consists of several model layers that are all required to use PCM for performance predictions [BKR09]. This section describes how the repository model layer can be generated automatically as the other model layers can only be created using the information contained in this model. The PCM repository model contains the components of a system, their operation behavior and resource demands as well as their relationships. Repository model components are assembled in a system model to represent an application. User interactions with the system are described in a usage model. The other two model layers in PCM are the resource environment and allocation model. The purpose of a resource environment model is to specify available resource containers (i.e., servers) with their associated hardware resources (CPU or HDD). An allocation model specifies the mapping of components to resource containers. To simplify the use



(a) Repository model example [BVK13] (b) RDSEFF of ComponentA.operationA [BVK13] (c) RDSEFF of ComponentB.operationC



(d) Heap component (e) Calling Heap.alloc and Heap.free from RDSEFFs

Figure 2: PCM repository model elements

of generated repository models, default models for the other PCM model layers are generated automatically once the repository model generation is complete [BVK13]. The PCM usage model is not generated automatically and has to be modeled manually.

Following the data model in figure 1, *JavaEEComponentOperationMBean* instances in a Java EE runtime are used to generate a repository model. The model generation is implemented as a Java-based client that accesses the MBean data using the JMX Remote API. The list of available *JavaEEComponentOperationMBeans* is first of all used to generate component elements in the repository model (e.g., *ComponentA* and *ComponentB* in figure 2(a)). The component list can be derived from the data model by filtering the available *JavaEEComponentOperationMBeans* by the *componentName* attribute of the associated *OperationIdentifier*. Operations provided by Java EE components are also available in the same data structure and are generated in the same step. In a PCM repository model, operations provided by a component are specified in an interface (e.g., *IComponentA* and *IComponentB* in figure 2(a)).

Afterwards, the data in the list of *ExternalOperationCalls* for each *BranchDescriptor* of a *JavaEEComponentOperationMBean* is used to represent component relationships. These relationships are specified in a repository model by a *requires* relationship between the repository component that calls a specific component operation and the interface that provides this operation (e.g., *ComponentA requires IComponentB* in figure 2(a)). The model generation can therefore use information about external operations called in specific operation control flows (using the *OperationIdentifier* of the *ExternalOperationCalls*) to create the relationships of repository model components.

So far, only components, interfaces and their relationships are available in the repository

model. In the next step, the behavior of component operations needs to be specified. The component operation behavior is specified in Resource Demanding Service Effect Specifications (RDSEFF). RDSEFFs are behavior descriptions similar to activity diagrams in the Unified Modeling Language (UML).

As explained in the data collection section 2.1, a component operation can be included in different requests processed by a Java EE runtime (see *ParentOperationBranch* in figure 1). To represent the resulting behavior differences in a performance model, a *parent-operation* parameter is passed between component operations. An example can be found in figure 2(b): *operationA* of *ComponentA* is the first operation called during a request, it thus specifies the *parentoperation* as *ComponentA_operationA* in the external call to *ComponentB.operationC*. This parameter is used in the RDSEFF of *operationC* of *ComponentB* to differentiate the operation behavior depending on the parameter value (see figure 2(c)). The initial *parentoperation* parameter value that is set when a request starts is passed on to all sub-invocations. For example, *ComponentA_operationA* would be passed on if *ComponentB.operationC* would call another external operation within this request. The behavior description of *ComponentA.operationA* in figure 2(b) is also contained in a guarded branch with the condition that the current operation needs to be *ComponentA_operationA*. This is necessary to ensure that all component operations can be used equally in the PCM repository and usage model layers. Thus, operations that only start requests and those that are also (or only) used within requests are indistinguishable.

A component operation can behave differently even though the same *parentoperation* initiated the request processing. These control flow differences are represented in RDSEFFs using probability branches [BKR09]. The probability of each branch (=*BranchDescriptor*) can be calculated based on data in *BranchMetrics* objects. If only one *BranchDescriptor* object exists for a *ParentOperationBranch* object, the probability is one. Otherwise, the *invocationCount* sum of all *BranchMetrics* objects for a *ParentOperationBranch* is used to calculate the probability for a single probability branch in a RDSEFF. An example for such probability branches can be found in figure 2(b). The RDSEFF of *ComponentA.operationA* contains two probability branches (*Control Flow 1* and *Control Flow 2*). One is executed with 60 % probability whereas the second is executed with 40 % probability. The *OperationCallLoopCounts* for different *ExternalOperationCalls* in a specific branch are represented as loop probabilities. For example, in figure 2(b), the external call to *operationC* of *ComponentB* is executed five times in 20 % of the cases and nine times in the other 80 %.

Resource demand data in *BranchMetric* objects is also represented in a probability branch of a RDSEFF. The mean CPU demand in milliseconds (ms) calculated based on the *BranchMetrics* data can be directly assigned to an internal action of a probability branch. In the example in figure 2(b), *ComponentA.operationA* consumes 0.123 ms CPU time in *Control Flow 1*, whereas *Control Flow 2* consumes 0.456 ms.

Representing heap memory demand of a component operation is not directly supported by the PCM meta model. Therefore, the passive resources element of the meta model is reused for this purpose [BKR09]. Even though passive resources are intended to be used as semaphores or to represented limited pool sizes (e.g., for database connections), one can also use them to create a simplistic representation of the memory demand of an application. For this purpose, a heap component is generated in each repository model as shown

in figure 2(d). This heap component contains a specified amount of passive resources that represents the maximum heap size available in a JVM. The maximum configurable value for the available passive resources of the heap component is 2^{31}-1. Thus, if one interprets one passive resource as one byte (B), the maximum configurable heap is two gigabytes (GB). As this is a very low value for Java EE applications nowadays, the model generation can be configured to interpret one passive resource as 10 bytes, so that the maximum representable heap is 20 GB. To do this, all heap memory demands read from the *BranchMetrics* objects are divided by ten and are rounded because passive resources can only be acquired as integer values. As this reduces the accuracy of the model, one passive resource is interpreted as one byte by default.

To allow other component operations in the repository model to consume heap memory, the heap component offers two operations: *alloc(int bytesRequested)* and *free(int bytesReleased)* (see figure 2(d)). This model follows the API for applications written in the programming language C. Using the information about the heap demand gathered in the data collection step (see section 2.1), calls to the *Heap.alloc* operation are generated at the beginning of each execution flow and calls to *Heap.free* at the end. An example is shown in figure 2(e): *operationD* of *ComponentC* calls *alloc* with a value of 200 bytes at the beginning, performs some internal processing and releases the 200 bytes allocated previously. Even though this memory model representation is not realistic for Java applications as the garbage collector (GC) behavior is not represented, the overall utilization of the passive resources helps to get an idea of the heap memory demand of an application.

3 Evaluating the Performance Prediction Accuracy

The feasibility of the model generation approach is evaluated in a case study using a SPECjEnterprise2010[1] industry standard benchmark deployment. SPECjEnterprise2010 is used for this evaluation to make it reproducible as it defines an application, a workload as well as a dataset for a benchmark execution.

3.1 SPECjEnterprise2010 Deployment

The SPECjEnterprise2010 benchmark represents the business case of an automobile manufacturer. It is divided into three application domains. The evaluation in this paper focuses on the Orders domain. This domain is used by automobile dealers to order and sell cars. To avoid the need to model all domains, the communication between the Orders and the other domains is disabled. The setup of the Orders domain consists of a benchmark driver to generate load and a system under test (SUT) on which the Orders domain application com-

[1] SPECjEnterprise is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjEnterprise2010 is located at http://www.spec.org/jEnterprise2010.

ponents are executed. The Orders domain is a Java EE web application that is composed of Servlet, JSP and EJB components. The automobile dealers (hereafter called users) access this application using a web interface over the hypertext transfer protocol (HTTP) and can perform three different business transactions: browse, manage and purchase. These three business transactions are composed of several HTTP requests to the system. The user interactions with the system are implemented as load test scripts in the Faban harness[2]. Faban is a workload creation and execution framework which is used to generate load on the SUT.

The benchmark driver and the SUT are each deployed on a virtual machine (VM) to simplify changing the number of available CPU cores. These two virtual machines are connected by a one gigabyte-per-second network connection and are mapped to an IBM System X3755M3 hardware server which is exclusively used for the SPECjEnterprise2010 benchmarks performed for this evaluation. Both virtual machines run CentOS 6.4 as operating system and are configured to have 20 GB of random-access memory (RAM). The benchmark driver is equipped with eight CPU cores, the number of CPU cores of the SUT is varied during the evaluation. The SPECjEnterprise2010 Orders domain application is deployed on a JBoss Application Server (AS) 7.1.1 in the Java EE 6.0 full profile. The database on the SUT VM is an Apache Derby DB in version 10.9.1.0. The JBoss AS and the Apache Derby DB are both executed in the same JVM, which is a 64 bit Java OpenJDK Server VM in version 1.7.0. An external Java-based client for the model generation is connected to the SUT using the JBoss JMX remote API.

3.2 Evaluating the Data Collection Overhead

The model generation approach introduced in this work relies on data collected using a runtime instrumentation. The instrumentation overhead for collecting the required data is analyzed in this section. As mentioned in the data collection section (see section 2.1), the instrumentation is capable of collecting the CPU and Java heap memory demand for each externally accessible component operation. The instrumentation code is therefore always executed before and after a component operation and can have a great influence on the performance data stored in the performance model.

To evaluate the impact of the data collection, the resource demand of several control flows of the SPECjEnterprise2010 Orders domain application is analyzed using different data collection configurations. For each of the following data collection configurations a SPECjEnterprise2010 benchmark run is executed and afterwards a performance model is generated. The SPECjEnterprise2010 deployment outlined in section 3.1 is used in a configuration with four CPU cores for the SUT. To avoid influences of warm up effects and varying user counts, only steady state data (i.e., data collected during 10 minutes between a five minute ramp up and a 150 second ramp down phase) is collected. The data collection runs are executed with a workload of 600 concurrent users which corresponds to a CPU utilization of the SUT of about 50 %. The resulting performance models contain

[2] http://java.net/projects/faban/

Table 1: Measured instrumentation overhead for the data collection - control flow one

Component Operation		Model 1.1		Model 1.2		Model 2.1	Model 2.2
Order	**Name**	**CPU**	**Heap**	**CPU**	**Heap**	**CPU**	**CPU**
1	app.sellinventory	1.023 ms	33,650 B	3.001 ms	225,390 B	0.756 ms	3.003 ms
2	CustomerSession.sellInventory	0.785 ms	60,450 B			0.731 ms	
3	CustomerSession.getInventories	0.594 ms	49,540 B			0.548 ms	
4	OrderSession.getOpenOrders	0.954 ms	70,600 B			0.878 ms	
5	dealerinventory.jsp.sellinventory	0.108 ms	16,660 B			0.103 ms	
Total Resource Demand		3.464 ms	230,900 B	3.001 ms	225,390 B	3.015 ms	3.003 ms
Mean Data Collection Overhead		0.116 ms	1378 B			0.003 ms	

the aggregated resource demands collected in the MBeans for these component operations and therefore simplify the analysis. The resource demand for the database is already included in the following measurements, as the embedded derby DB is executed in the same thread as the Servlet, JSP and EJB components.

The mean CPU and heap demands for single component operations involved in three different control flows are shown in tables 1, 2 and 3. Both resource demand types (CPU and heap) are represented as mean values for the data collected during the steady state. The heap demand values in this table are rounded to 10 byte intervals as the model generation is configured to do so to have 20 GB of heap available in the model (see section 2.1).

In a first step, a benchmark run is executed while the CPU and heap demand for all component operations involved in the request processing is collected. The performance model generated based on this configuration is called *Model 1.1* in tables 1, 2 and 3. Afterwards, a benchmark run is executed while only the CPU demand for each component operation is collected. The resulting performance model based on this data collection configuration is called *Model 2.1* in tables 1, 2 and 3. Both benchmark runs are repeated but this time, only resource demand data (CPU or CPU & heap) for the first component operations (those where Order==1 in tables 1, 2 and 3) of each HTTP request is collected. These measurements already include the CPU and heap demands of the sub-invocations (Order $>$1 in tables 1, 2 and 3). The resulting performance models are called *Model 1.2* for the first configuration (CPU and heap collection turned on) and *Model 2.2* for the second configuration (CPU collection turned on).

The total mean CPU and heap demand values in the first model versions (*1.1* and *2.1*) are compared with the corresponding values for the second model versions (*1.2* and *2.2*) to calculate the instrumentation overhead. It can be shown that collecting heap and CPU demands for each component operation is a lot more expensive than only collecting CPU demand. For the HTTP request analyzed in table 1, the mean overhead for the data collection including heap demand is 0.116 ms CPU time and 1378 byte heap memory for each component operation. If only the CPU demand is collected, the mean data collection overhead drops dramatically to 0.003 ms for each component operation.

Other execution flows during the same benchmark runs confirm these values (two additional examples are given in tables 2 and 3). The mean instrumentation overhead for the CPU-only collection is mostly below 0.020 ms whereas the mean instrumentation overhead for the CPU and heap collection ranges mostly between 0.060 and 0.120 ms. As some component operations in the SPECjEnterprise2010 benchmark have an overall CPU

Table 2: Measured instrumentation overhead for the data collection - control flow two

Component Operation		Model 1.1		Model 1.2		Model 2.1	Model 2.2
Order	**Name**	**CPU**	**Heap**	**CPU**	**Heap**	**CPU**	**CPU**
1	app.view_items	0.406 ms	20,560 B	3.529 ms	615,440 B	0.165 ms	3.566 ms
2	ItemBrowserSession.browseForward	3.315 ms	565,130 B			3.282 ms	
3	ItemBrowserSession.getCurrentMin	0.003 ms	60 B			0.003 ms	
4	ItemBrowserSession.getCurrentMax	0.003 ms	60 B			0.002 ms	
5	ItemBrowserSession.getTotalItems	0.003 ms	60 B			0.002 ms	
6	purchase.jsp.view_items	0.147 ms	40,380 B			0.142 ms	
Total Resource Demand		3.877 ms	626,250 B	3.529 ms	615,440 B	3.598 ms	3.566 ms
Mean Data Collection Overhead		0.070 ms	2162 B			0.006 ms	

Table 3: Measured instrumentation overhead for the data collection - control flow three

Component Operation		Model 1.1		Model 1.2		Model 2.1	Model 2.2
Order	**Name**	**CPU**	**Heap**	**CPU**	**Heap**	**CPU**	**CPU**
1	app.add_to_cart	0.213 ms	11,300 B	0.393 ms	27,520 B	0.108 ms	0.388 ms
2	OrderSession.getItem	0.276 ms	13,460 B			0.255 ms	
3	shoppingcart.jsp.add_to_cart	0.059 ms	5960 B			0.058 ms	
Total Resource Demand		0.548 ms	30,720 B	0.393 ms	27,520 B	0.421 ms	0.388 ms
Mean Data Collection Overhead		0.077 ms	1600 B			0.017 ms	

demand of below 0.150 ms, collecting the heap demand for this deployment causes too much overhead. The following evaluation therefore focuses on models generated based on the CPU demand collection.

3.3 Comparing Measured and Simulated Results

In the next two sections, the prediction accuracy of generated performance models is evaluated in an upscaling and a downscaling scenario. The steps for both evaluations are similar and are described in the following paragraphs.

Load is generated on the SUT to gather the required data for the model generation in each scenario using the data collection approach outlined in section 2.1. As the database is included within the server JVM, the collected data already contains its CPU demands. Similar to the benchmark runs in the overhead evaluation, only steady state data (i.e., data collected during 10 minutes between a five minute ramp up and a 150 second ramp down phase) is collected. Afterwards, a software prototype that implements the performance model generation approach is used to generate a PCM model based on the collected data.

PCM models can be taken as the input for a simulation engine to predict the application performance for different workloads and resource environments. The standard simulation engine for PCM models is SimuCom which uses model-2-text transformations to translate PCM models into Java code [BKR09]. The code is then compiled and executed to start a simulation. To evaluate the accuracy of the simulation results, they are compared with measurements on the SUT. The following comparisons only use steady state data collected during simulation and benchmark runs of similar length.

The benchmark driver reports the mean response time and throughput for each business

transaction for a benchmark run. However, the predicted response time values cannot be compared with response time values reported by the driver, because they do not contain the network overhead between the driver and the SUT. Therefore, response time of the business transactions browse (B), manage (M) and purchase (P) is measured on the SUT using an additional instrumentation. To identify business transactions using this instrumentation, the benchmark driver is patched to add a unique transaction identifier to each request. This identifier allows combining several HTTP requests into one business transaction. Incoming requests are aggregated on the fly to the business transaction they belong to by summing up their response times. The resulting business transaction response time measurements are stored with a timestamp to calculate the mean throughput on a per-minute basis.

The CPU time consumed by the JVM process of the JBoss AS on the SUT (and thus its CPU utilization) is collected every second to reconstruct its approximate progression and to compare the measured and simulated ranges. The calculation of the mean CPU utilization is based on the first and the last CPU time consumption value in the steady state of a benchmark run in order to avoid biasing effects caused by unequal measurement intervals.

Each benchmark run is performed three times, the results are combined giving each run the same weight. Since all runs have the same duration, the overall mean value of CPU utilization can be calculated by averaging the corresponding values of each run. The throughput values represent the amount of times a business transaction is invoked per minute, thus the collected per-minute values are combined to a mean value. To evaluate response times, samples of equal sizes are drawn from each result. Response time measurement and simulation results are described using mean and median values as well as values of dispersion, namely the quartiles and the interquartile range (IQR). Variance and standard deviation are excluded from our investigation due to the skewness of the underlying distributions [Jai91] of the response times of browse, manage and purchase. In the following sections, means are illustrated tabularly, medians and quartiles are illustrated using boxplot diagrams.

3.4 Evaluating Prediction Accuracy in an Upscaling Scenario

To evaluate the performance prediction accuracy of automatically generated performance models in an upscaling scenario, the number of CPU cores for simulation and benchmark runs is increased step by step. To increase the CPU core count of the SUT for the benchmark runs, the VM is reconfigured accordingly. The number of simulated CPU cores is varied by editing the generated resource environment model.

If workload stays stable, the CPU utilization significantly declines with each increase of cores as does its impact on the overall application performance. As a result, after reaching a sufficient number of CPU cores, the measured response times stay almost constant regardless of any further increases while the simulated response times decrease further reaching their lower bound only at a very high number of CPU cores. Therefore, an increasing inaccuracy in the simulated values is expected since the generated model solely

Table 4: Measured and simulated results in an upscaling scenario

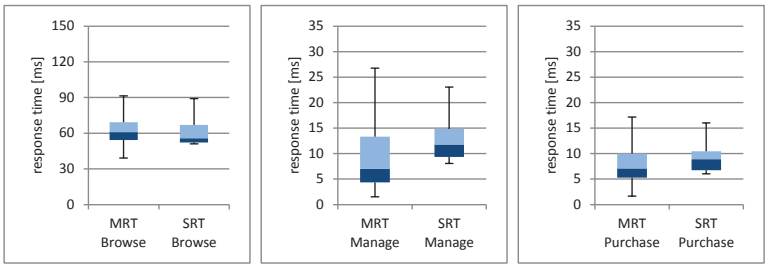
C	U	T	MMRT	SMRT	RTPE	MMT	SMT	TPE	MCPU	SCPU	CPUPE
4	600	B	63.23 ms	65.06 ms	2.91 %	1820.6	1813.1	0.41 %	48.76 %	46.87 %	3.88 %
		M	11.58 ms	13.28 ms	14.71 %	906.8	917.3	1.16 %			
		P	8.27 ms	9.73 ms	17.67 %	904.9	900.3	0.50 %			
6	900	B	69.25 ms	57.56 ms	16.89 %	2708.3	2721.5	0.49 %	51.72 %	46.85 %	9.42 %
		M	12.54 ms	11.95 ms	4.69 %	1354.3	1354.4	0.01 %			
		P	8.95 ms	8.72 ms	2.60 %	1352.4	1368.1	1.16 %			
8	1200	B	88.82 ms	56.25 ms	36.66 %	3617.8	3641.9	0.67 %	57.34 %	46.97 %	18.09 %
		M	14.13 ms	11.64 ms	17.67 %	1806.4	1795.0	0.63 %			
		P	9.31 ms	8.46 ms	9.15 %	1811.6	1819.2	0.42 %			

depends on CPU demands and disregards other factors such as I/O operations on hard disk drives. Thus, to keep the CPU utilized, the workload on the system is varied proportional to the number of CPU cores by increasing the number of concurrent users accessing the SUT. In the following, a performance model generated on the SUT configured with 4 CPU cores is used. The average CPU utilization while gathering the data required for the model generation was 52.46 % which corresponds to a closed workload consisting of 600 users with an average think time of 9.9 s.

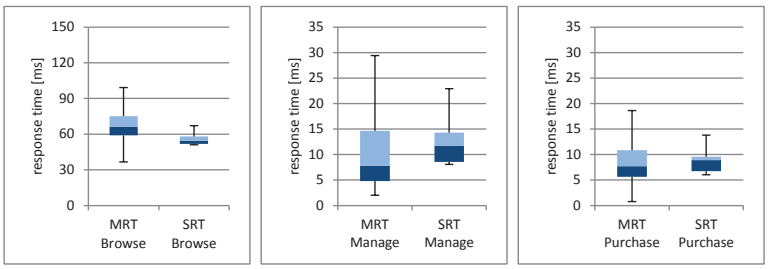
In a first step, the generated model is evaluated by simulating the application performance for an environment which is equal to the one the model has been generated with. Afterwards, the model is evaluated for environments with an increased number of CPU cores. The measured and simulated results are shown in table 4. For each configuration specified by the number of cores (C) and the number of users (U), the table contains the following data per business transaction (T): Measured Mean Response Time (MMRT), Simulated Mean Response Time (SMRT), relative Response Time Prediction Error (RTPE), Measured Mean Throughput (MMT), Simulated Mean Throughput (SMT), relative Throughput Prediction Error (TPE), Measured (MCPU) and Simulated (SCPU) Mean CPU Utilization and the relative CPU Utilization Prediction Error (CPUPE).

The simulation predicts the mean response time of the business transactions with a relative error of less than 20 %, except for the browse transaction in the case of 8 CPU cores and 1200 concurrent users, which shows a relative prediction error of 36.66 %. CPU utilization is predicted with relative errors ranging from 3.88 % to 18.09 %. Due to space limitations, the span consisting of the minimum and maximum of the measured and simulated CPU utilization values is not shown. However, while both ranges mostly overlap, the measured span lies slightly above the simulated one. The same applies to the mean CPU utilization values shown in table 4, as the simulated mean is slightly lower than the measured one. The prediction of the mean throughput is very close to the real values, as the think time of 9.9 s is much higher than the highest response time. Response time prediction errors thus have a low impact on the throughput. Except for the last simulation of browse, the quality of the predictions ranges from very good to still acceptable for the purpose of capacity planning [MAL+04].

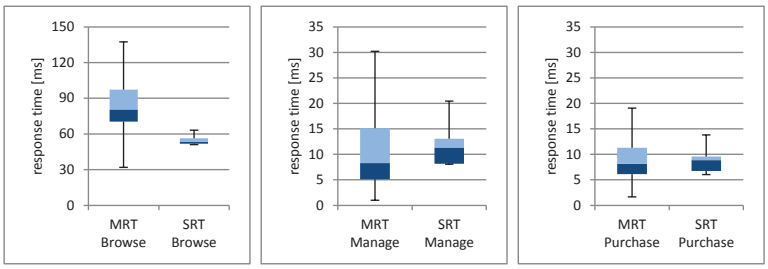
Further statistical measures are illustrated as boxplot diagrams in figure 3. Boxplot diagrams consist of a box whose bounds denote the first quartile Q_1 (lower bound) as well as the third quartile Q_3 (upper bound) of the underlying data sample. The quartiles are con-



(a) 4 CPU cores and 600 users



(b) 6 CPU cores and 900 users



(c) 8 CPU cores and 1200 users

Figure 3: Boxplot diagrams of an upscaling scenario

nected by vertical lines to form the box that indicates the interquartile range (IQR) which is defined as $Q_3 - Q_1$. Furthermore, the median Q_2 is illustrated by a horizontal line within the box, thus separating it into two parts. Vertical lines outside the box (whiskers) indicate the range of possible outliers while their length is limited to 1.5 times the IQR.

The relative prediction error of the median response time ranges from 8.38 % to 33.80 % for the browse and purchase transactions. The median response time of the manage transaction, however, is predicted with a relative error of 36.52 % to 68.29 %. The skewness of a business transaction's underlying distribution can be determined considering the median's position between the quartiles Q_1 and Q_3. The boxplot diagrams in figure 3 show that the skewness is not simulated correctly. To investigate the dispersion of business transactions, we determine the IQR. Its relative prediction error ranges from 21.96 % to 50.94 % for the

Table 5: Measured and simulated results in a downscaling scenario

C	U	T	MMRT	SMRT	RTPE	MMT	SMT	TPE	MCPU	SCPU	CPUPE
8	800	B	71.54 ms	64.03 ms	10.50 %	2413.9	2415.8	0.08 %	37.41 %	35.17 %	5.99 %
		M	12.96 ms	12.64 ms	2.49 %	1203.5	1209.2	0.48 %			
		P	9.36 ms	9.33 ms	0.25 %	1215.9	1228.7	1.05 %			
6	800	B	67.62 ms	66.03 ms	2.35 %	2413.9	2425.4	0.48 %	46.38 %	46.94 %	1.21 %
		M	12.52 ms	13.08 ms	4.45 %	1202.0	1196.6	0.45 %			
		P	9.05 ms	9.64 ms	6.57 %	1208.2	1215.0	0.56 %			
4	800	B	71.15 ms	87.46 ms	22.92 %	2437.0	2420.8	0.66 %	65.60 %	70.27 %	7.12 %
		M	12.98 ms	17.04 ms	31.29 %	1199.7	1193.5	0.51 %			
		P	8.93 ms	12.88 ms	44.33 %	1211.6	1212.1	0.04 %			

manage and purchase transactions and is up to 83.13 % for the browse transaction.

In the measurement results, the effect of increasing the workload dominates, thus the measured CPU utilization slightly increases from 48.76 % to 57.34 %. The response times increase accordingly. In the simulation results, the effect of core increase slightly dominates over the effect of increasing the workload. Therefore, the response times slightly decrease over the course of the experiment, while the simulated CPU utilization remains almost constant.

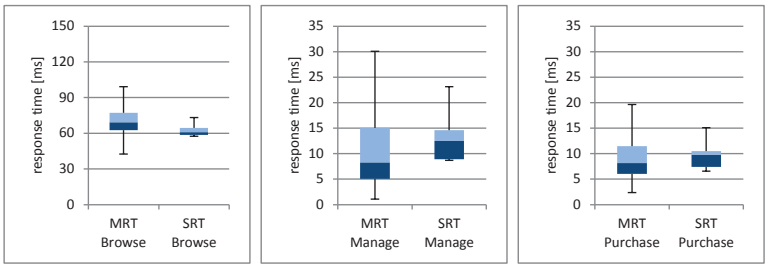
3.5 Evaluating Prediction Accuracy in a Downscaling Scenario

The prediction accuracy of generated performance models in a downscaling scenario is evaluated by reducing the number of CPU cores step by step. Starting with 8 CPU cores, the number of cores is decreased by 2 in each evaluation step. This scenario does not require the number of users to be varied, as the CPU utilization increases. The business case of scaling the number of CPU cores down is to optimize production systems (e.g., to evaluate if several applications can be hosted on one machine or to reduce license fees). In this case, the number of users does not change. Therefore, the workload is kept constant at 800 users with a think time of 9.9 s accessing the SUT in parallel.

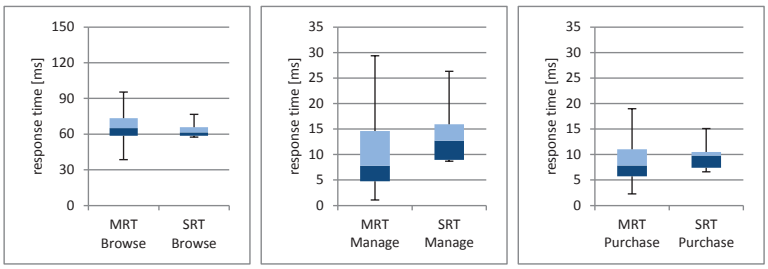
Since the number of cores is reduced during the experiment, a sufficiently low starting value of CPU utilization is required. Therefore, data to generate a performance model for this evaluation is collected with an average CPU utilization of 38.9 %. To compare the simulation results with the measured ones, the previously described evaluation process is applied. The comparison of the mean response time, CPU utilization and throughput values is shown in table 5.

The relative prediction error for the mean response time of all business transactions is at most 44.33 %. CPU utilization is predicted with a maximum relative error of 7.12 %. In contrast to the upscaling scenario, the simulated CPU utilization grows slightly above the measured results as the CPU cores are decreased. The relative prediction error of the mean throughput is about 1 %.

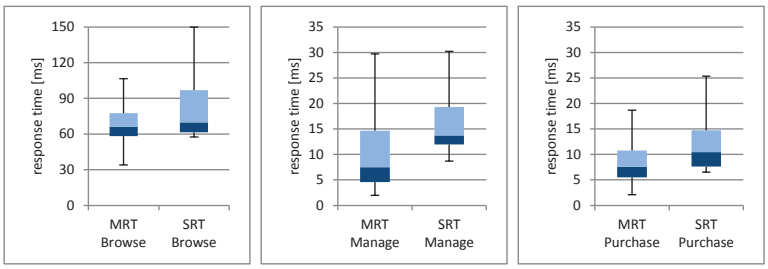
The relative prediction error of the median response time as shown in the boxplots in figure 4 ranges from 5.27 % to 38.50 % for the browse and purchase transactions and from



(a) 8 CPU cores and 800 users



(b) 6 CPU cores and 800 users



(c) 4 CPU cores and 800 users

Figure 4: Boxplot diagrams of a downscaling scenario

50.51 % to 82.96 % for the manage transaction. This is in line with the observations previously made in the upscaling scenario. The relative IQR prediction error ranges from 27.34 % to 43.48 % for the manage and purchase transaction; for the browse transaction it is up to 83.37 %.

Comparing the measured mean and median response times shows that the lowest values are achieved in the 6 CPU core configuration. Even with 4 CPU cores, the browse and purchase response times are lower than in the 8 CPU core configuration. As the CPU utilization of the investigated configurations is relatively low, the lower performance of the SUT with 8 CPU cores can be explained by an increased scheduling overhead. Due to the low CPU utilization, its impact on the overall performance of the SUT is lower than the impact of other factors such as I/O operations. The response time prediction behaves

incorrectly in these cases, as the generated performance model only relies on the CPU demand measured during the data collection step and does not take these effects into account. However, the simulation of CPU utilization is still very close to the measurements. This is useful for determining a lower bound of feasible configurations regarding the amount of cores. Simulating an environment consisting of 3 CPU cores results in a simulated CPU utilization of 91.96 % and indicates that this would lead to instability of the SUT for the given workload. This configuration is thus not investigated in this downscaling scenario.

4 Related Work

Running Java EE applications have already been evaluated using performance models by several authors. Chen et al. [CLGL05] derive mathematical models from measurements to create product-specific performance profiles for the EJB runtime of a Java EE server. These models are intended to be used for performance predictions of EJB components running on different Java EE products. Their approach is thus limited to Java EE applications that solely consist of this component type.

Liu et al. [LKL01] also focus on EJB components and show how layered queuing networks can be used for the capacity planning of EJB-based applications. In their work, they model an EJB-based application manually and describe how an analytical performance model needs to be calibrated before it can be used for the capacity planning. To improve this manual process, Mania and Murphy [MM02] proposed a framework to create analytical performance models for EJB applications automatically. However, the framework was never evaluated to the best of our knowledge.

The difficulties in building and calibrating performance models for EJB applications manually are also described by McGuiness et al. [MML04]. Instead of using analytical solutions to predict the performance of an EJB-based application, they are using simulation models. The authors argue that simulation models are better suited for the performance evaluation of EJB applications due to their flexibility and increased accuracy compared to analytical models.

The applicability of analytical performance models for Java EE applications with realistic complexity is analyzed by Kounev and Buchmann [KB03] using the SPECjAppServer2002 industrial benchmark. Kounev extends this work in [Kou06] by using queuing Petri nets to evaluate the performance of a SPECjAppServer2004 benchmark deployment. The latest version of the SPECjAppServer benchmark (SPECjEnterprise2010) is used by Brosig and Kounev in [BHK11] to show that they are able to semi-automatically extract PCM models for Java EE applications. Their model generation approach is based on data generated by the monitoring framework of Oracle's WebLogic product and thus not transferable to other Java EE server products. It also requires manual effort to distribute the resource demand based on the service demand law once a model is generated.

The previous work is extended by the approach introduced in this work as it is applicable for all Java EE server products and can generate performance models for EJB as well as for web components automatically.

5 Conclusion and Future Work

The approach presented in this work aims to make performance modeling better applicable in practice. The ability to generate performance models at any time simplifies their use in Java EE development projects, as the effort to create such models is very low. The evaluation showed that the generated performance models predict the performance of a system in up- and downscaling scenarios with acceptable accuracy. The approach can thus support related activities during the capacity planning and management processes.

Future work for this approach includes extending the data collection and model generation capabilities. First of all, we need to investigate whether the user session information available in the Java EE runtime can be used to generate usage models automatically. Further extensions are required to support additional technologies specified under the umbrella of the Java EE specification, such as JavaServer Faces (JSF) or web services. For this purpose, the model generation approach also needs to be extended to support distributed systems. A key challenge for such an extension is the integration and correlation of MBean data collected from multiple Java EE servers. Additional improvements are required to reduce the instrumentation overhead as soon as heap demand needs to be collected.

6 Acknowledgements

The authors would like to thank Jörg Henß, Klaus Krogmann and Philipp Merkle from the Karlsruhe Institute of Technology (KIT) and FZI Research Center for Information Technology at KIT for their valuable input and support while implementing the heap representation approach in PCM.

References

[BDMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[BHK11] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE)*, pages 183–192, Oread, Lawrence, Kansas, USA, 2011.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[BVD+14] Andreas Brunnert, Christian Vögele, Alexandru Danciu, Matthias Pfaff, Manuel Mayer, and Helmut Krcmar. Performance Management Work. *Business & Information Systems Engineering*, 6(3):177–179, 2014.

[BVK13] Andreas Brunnert, Christian Vögele, and Helmut Krcmar. Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications. In Maria Simonetta Balsamo, William J. Knottenbelt, and Andrea Marin, editors, *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin Heidelberg, 2013.

[BWK14] Andreas Brunnert, Kilian Wischer, and Helmut Krcmar. Using Architecture-Level Performance Models As Resource Profiles for Enterprise Applications. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 53–62, New York, NY, USA, 2014. ACM.

[CLGL05] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance Prediction of Component-based Applications. *Journal of Systems and Software*, 74(1):35–43, 2005.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley Computer Publishing, John Wiley & Sons, Inc., 1991.

[KB03] Samuel Kounev and Alejandro Buchmann. Performance Modeling and Evaluation of Large-Scale J2EE Applications. In *Proceedings of the 29th International Conference of the Computer Measurement Group on Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG), Dallas, Texas, USA*, pages 273–283, 2003.

[Kou06] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, 2006.

[Koz10] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634–658, 2010.

[LKL01] Te-Kai Liu, Santhosh Kumaran, and Zongwei Luo. Layered Queueing Models for Enterprise JavaBean Applications. In *Proceedings of the IEEE International Conference on Enterprise Distributed Object Computing*, pages 174–178, Washington, DC, USA, 2001. IEEE.

[MAL+04] Daniel A. Menascé, Virgilio A. F. Almeida, F. Lawrence, W. Dowdy, and Larry Dowdy. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall, Upper Saddle River, New Jersey, 2004.

[Mic06] Sun Microsystems. Java Management Extensions (JMX) Specification, vers. 1.4, 2006.

[MM02] D. Mania and J. Murphy. Framework for Predicting the Performance of Component-Based Systems. In *Proceedings of the 10th IEEE International Conference on Software, Telecommunications and Computer Networks*, Croatia, Italy, 2002.

[MML04] D. McGuinness, L. Murphy, and A. Lee. Issues in Developing a Simulation Model of an EJB System. In *Proceedings of the 30th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems*, Las Vegas, Nevada, USA, 2004.

[Sha06] Bill Shannon. Java Platform, Enterprise Edition (Java EE) Specification, v5, 2006.

[WW04] Xiuping Wu and Murray Woodside. Performance modeling from software components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.

Using and Extending LIMBO for Descriptive Modeling of Arrival Behaviors

Jóakim v. Kistowski, Nikolas Herbst, Samuel Kounev

Chair for Computer Science II, Software Engineering
University of Würzburg
joakim.kistowski@uni-wuerzburg.de
nikolas.herbst@uni-wuerzburg.de
samuel.kounev@uni-wuerzburg.de

Abstract: LIMBO is a tool for the creation of load profiles with variable intensity over time both from scratch and from existing data. Primarily, LIMBO's intended use is the description of load arrival behaviours in open workloads. Specifically, LIMBO can be employed for the creation of custom request or user arrival time-stamps or for the re-parameterization of existing traces.

LIMBO bases on the Descartes Load Intensity Model (DLIM) for the formalized description of its load intensity profiles. The DLIM formalism can be understood as a structure for piece-wise defined and combined mathematical functions. We outline DLIM and its elements and demonstrate its integration within LIMBO.

LIMBO is capable of generating request or user arrival time stamps from DLIM instances. In a next step, these generated time-stamps can be used for both open workload based benchmarking and simulations. The TimestampTimer plug-in for JMeter already allows the former. LIMBO also offers a range of tools for easy load intensity modeling and analysis, including, but not limited to, a visual decomposition of load intensity time-series into seasonal and trend parts, a simplified load intensity model as part of a model creation wizard, and an automated model instance extractor.

As part of our LIMBO tutorial, we explain these features in detail. We demonstrate common use cases for LIMBO and show how they are supported. We also focus on LIMBOs extensible architecture and discuss how to use LIMBO as part of another tool-chain.

1 Introduction

Today's cloud and web-based IT services need to handle huge amounts of concurrent users. Customers access services independently of one another and expect reliable quality-of-service under highly variable and dynamic load intensities. In this context, any knowledge about a service's load intensity profile is becoming a crucial information for managing the underlying IT resource landscape. Load profiles with large amounts of concurrent users are typically strongly influenced by human habits, trends, and events. This includes strong deterministic factors such as time of the day, day of the week, common working hours and planned events.

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

Common benchmarking frameworks such as Faban[1], Rain [BLY+10], and JMeter [Hal08] allow job injection rates to be configured either to constant values, stepwise increasing rates (e.g., for stress tests), or rates based on recorded workload traces. The tool we present in this document aims at closing the gap between highly dynamic load intensity profiles observed in real life and the current lack of support for flexible handling of variable load intensities in benchmarking frameworks.

In [vKHK14b], we introduce two modeling formalisms at different abstraction levels: At the lower abstraction level, the *Descartes Load Intensity Model* (DLIM) offers a structured and accessible way of describing the load intensity over time by editing and combining mathematical functions. The *high-level DLIM* (hl-DLIM) allows the description of load variations using few defined parameters that characterize the seasonal patterns, trends, as well as bursts and noise elements. An example load profile consisting out of a seasonal part, a trend and bursts is presented in Fig. 1.

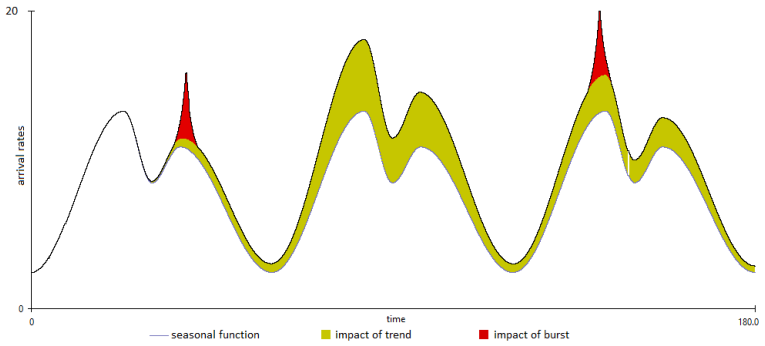


Figure 1: Example profile with a seasonal pattern, trend, and bursts.

In this document, we present LIMBO[2] [vKHK14a] - an Eclipse-based tool for handling and instantiating load intensity models based on DLIM. LIMBO offers an accessible way of editing DLIM instances and extracting them from existing traces. It also supports using hl-DLIM parameters for easy creation of new DLIM instances through a model creation wizard. In addition, we provide the *TimestampTimer* plugin for JMeter [Hal08], which enables the use of LIMBO-generated time stamps for the definition of JMeter work unit start times. We also provide extensive documentation for LIMBO on our website[2], including a thorough tutorial. In this document, we provide a description of the LIMBO tool and its architecture. We demonstrate LIMBO's extensibility and its use as part of other code projects.

[1]Faban `http://faban.org`
[2]LIMBO `http://go.uni-wuerzburg.de/limbo`

2 Definition of Load Intensity

In the context of LIMBO, *load intensity* is a discrete function describing *arrival rates* of workload units (e.g. users, sessions or requests) over time. We assume that the work units are of a homogeneous type and define the *arrival rate* $r(t)$ at time t as follows:

$$r(t) = R'(t)$$
$$\text{with } R(t) = |\{u_{t_0} | t_0 \leq t\}|$$

where $R(t)$ is the amount of all *work units* u_{t_0}, with their respective *arrival time* t_0, that have arrived up until time t.

3 Use-Cases for LIMBO

LIMBO can be used in many contexts, some of which might not be directly apparent. The following sections describe a few use-case scenarios in which the ability to create and modify a load intensity model is extremely helpful. This is intended to also demonstrate the usefulness of load intensity modeling in general and to give the reader a few additional ideas on what to do with LIMBO.

3.1 Core Use-Cases

LIMBO was created with the specific goal of being used as part of the following use cases:

- Creation of artificial load intensity profiles for specific benchmarking purposes
- Extraction of existing load intensity profiles from pre-existing traces.

LIMBO features an extensible architecture, which allows LIMBO's application beyond these core use cases, e.g.:

1. **Creating artificial Load Intensity Profiles for Benchmarking**

   A model describing the load intensity variations over time can be used to create request or user arrival time-stamps that can then be used to define the beginning time of a unit of work within a benchmarking framework. This enables a user to use a multitude of different varying workloads, which in turn helps with the benchmarking of system properties that deal with such variations (such as elasticity) [WHGK14].

   This use-case describes the possibility of a custom created load intensity profile, that has been specifically designed to help with the benchmarking of such a property. Of course, this load intensity profile may be subject to additional requirements, such as representability.

2. **Creating a Load Intensity Model Instance from an existing Trace**

   A model instance can be used to describe a past real-world load profile (within a certain error). This opens up a number of sub-use-cases:

   - *Parametrization of Request / User Arrival Traces*

     Among others, Zakay et al. [ZF13] sample request traces for benchmark workload generation. When doing so several problems can arise. The trace might be taken from a system that is magnitudes larger than the test system on which the benchmark is to be executed. The trace might also have been taken over a long time period and has to be temporally compressed for the benchmark. When using a load intensity model instead of a simple trace, these problems become easily manageable. They can be managed either by modifying the model instance directly, or through parametrization of the request time-stamp generation.

   - *Anonymization of Request / User Arrival Traces*

     Request traces of real cloud or web based systems often include additional information that may contain information about the system's users. Even the exact time-stamps themselves may still provide a reader of the trace with the ability to extract information about the behavior of single users.

     An abstract load variation representation helps to minimize this problem. System providers, who are concerned about customer anonymity, might be more likely to provide usage information for research purposes in an abstract form as made possible by a load intensity model.

3.2 Additional Use-Cases

These additional use-cases are either derived from the two previous cases or constitute new approaches to LIMBO's Load Intensity Model. They describe more complex scenarios, in which the features provided by LIMBO can play a central part.

1. **Load Intensity Forecasting**

   A model instance that has been derived from the incoming request trace of a currently running system can be used to predict future request intensity variations. Doing so can also help with the detection of unplanned events that deviate from an extrapolated periodic model. Such a forecasting mechanism could be deployed on cloud systems. In that context it would help to improve dynamic resource management, by increasing the efficiency of elastic resource re-allocation.

2. **Anomaly Detection**

   A calibrated load intensity model instance could serve as a baseline allowing the computation of anomaly metrics. This approach incorporating DLIM model instances as baseline will most presumably end up with a higher anomaly detection

accuracy and less false positives [Bie12]. Such a baseline can also be used in other fields of computer science, since it can always be used for comparison against anomalies. In a security context, it might be used for finding access patterns that deviate from usual access patterns in the form of their load intensity as part of an intrusion detection benchmark as proposed in [MK12].

4 LIMBO

LIMBO allows editing of load intensity models based on DLIM and supports guided model creation using the parameters defined in hl-DLIM.

4.1 Models

- **Descartes Load Intensity Model**: DLIM describes request arrival rates over time and offers a way to define a piece-wise mathematical function for the approximation of variable arrival rates with support for (partial) periodicity, flexibility and composability.
- **High-level DLIM**: hl-DLIM offers abstracted knowledge about load intensity variations modeled through a limited number of workload parameters. Inspired by the time series decomposition approach in BFAST [VHNC10], a hl-DLIM instance describes a *Seasonal* and *Trend* part. Additionally, it features a *Burst* and *Noise* part.

4.2 Features

LIMBO offers a significant number of different features, all targeted at enabling easy and comprehensive creation and modification of load intensity profiles. LIMBO has been implemented using an extensible architecture. It is thus open for extension with additional features. At this time, LIMBO's major features are:

1. **Creation of new load intensity profiles using hl-DLIM**: LIMBO enables the use of hl-DLIM parameters for easy creation of new DLIM instances through a model creation wizard.
2. **Modification of DLIM load profiles**: LIMBO allows for modification of DLIM load profiles, by adding, removing, and modifying the piece-wise mathematical functions of which these profiles are composed.
3. **Visualization of load profiles**: LIMBO includes a graphical view for the display of DLIM instances. This view also contains a more detailed visualization feature, which decomposes DLIM instances into their seasonal parts, trends, and bursts. It then displays each part's contribution towards the total load intensity.

4. **Timestamp generation**: Load intensity profiles can be used to generate request or user arrival time stamps. These time stamps can be used as input for common benchmarking frameworks, such as JMeter [Hal08]. LIMBO's extensible architecture also allows for easy addition of additional time-stamp exporters for other benchmarking frameworks, such as FinCos [MBM13].

5. **Timestamp use for load generation**: We provide the *TimestampTimer* plugin for JMeter. This plugin allows the use of LIMBO-generated time-stamps for the definition of work unit start times.

6. **Model instance extraction**: DLIM instances can be extracted from existing arrival rate traces using one of the implemented model instance extractors:

   - *Simple DLIM Extraction Process (s-DLIM)*: An accurate extraction process, which extracts DLIM instances from existing arrival rate traces. Our evaluation of s-DLIM accuracy in [vK14] shows a median extraction error of 12.4%. Comparison with BFAST[VHNC10] also shows, that s-DLIM provides excellent performance, with all extractions completing in less than 0.2 seconds and providing an average speedup of 8354 compared to BFAST decomposition.
   - *Periodic DLIM Extraction Process (p-DLIM)*: A less accurate extraction process to extract DLIM instances from existing arrival rate traces. Other than s-DLIM, p-DLIM instances are intended to be repeated for load intensity forecasting.
   - *high-level DLIM Extraction Process*: Extracts hl-DLIM instances from existing arrival rate traces.

   LIMBO's extensible architecture allows furhter model extraction and calibration methods to be integraded, as well as for reading other file formats, such as trace files as exported by Kieker [vHWH12].

4.3 Implementation

LIMBO, the tooling for DLIM and hl-DLIM models, is realized as a plug-in for the Eclipse IDE. It provides an editor for the creation and modification of model instances, as well as additional utilities for using the created models. Using DLIM's EMF-generated code base as a basis, the following features have been implemented:

- **Model Evaluation**: Support for the DLIM function output calculation and manual refinement of model instances.
- **Modeling Process Assistance**: Includes an automated process for the creation and extraction of DLIM instances. Additionally, LIMBO provides a model instantiation guidance by means of a wizard.
- **Utilities**: Additional functionality is provided for existing DLIM instances. Including functionality for the generation of arrival rate series from a time-stamp series,

and a tool that calculates the difference between an arrival rate trace and a model instance.

LIMBO consists of five individual plug-ins as visualized in Fig. 2. Note that all packages and plug-ins begin with the prefix "*tools.descartes.dlim*". For better readability "*tools.descartes*" is omitted for the remainder of this paper.

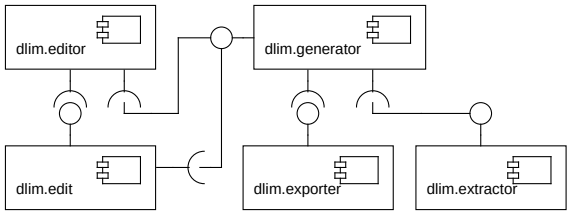


Figure 2: LIMBO architecture.

1. **DLIM Generator** The *dlim.generator* plug-in contains the DLIM element interfaces and implementations, as well as their default utilities (e.g., for validation). It also contains model evaluation tools, as well as arrival rate and time-stamp series generators. It features two extension points:

   - **Exporter** extension points supports custom implementations by implementing the *dlim.exporter.IDlimExporter* interface. Default exporters are contained in the *dlim.exporter* plug-in.
   - **Extractor** extension point allows the addition of extractors for deriving a model instance from an existing trace. Extractors must implement *dlim.reader.IDlimArrivalRateReader* for their trace parser and *dlim.extractor.IDlimExtractor* for the model instance creator. Default extractors are contained in the *dlim.extractor* plug-in.

   The *dlim.generator* plug-in also provides LIMBO's core functionalities for use as part of other projects. The most important provided packages are:

   - **dlim**: This package contains the model element interfaces. It is generated by the EMF genmodel, but has been modified to return a *CustomDlimFactoryImpl* Instance for the *DlimFactory.eINSTANCE*, instead of the generated *DlimFactoryImpl*.
   - **dlim.generator**: This package contains the model evaluation logic, primarily used for arrival rate and time-stamp series generation. The **ModelEvaluator** class, specifically, is the primary access point to all model evaluation logic. It is instantiated using the model's root element (which is always a *Sequence*) and a seed for the random number generator (for *Noise* evaluation). It provides the **getArrivalRateAtTime(double rootTime)** method, which returns the model's resulting arrival rate for a given time.

- **dlim.exporter.utils**

  This package contains utilities that help when implementing a new exporter. The use of these utilities is highly recommended.

  - **ArrivalRateGenerator**: Provides functionality to sample a list of arrival rates from a DLIM instance, represented by its *dlim.generator.ModelEvaluator*.
  - **TimeStampWriter**: Provides functionality to generate a list of request time stamps using a list of arrival rates (as is provided by *ArrivalRateGenerator*.

- **dlim.reader**

  This package contains classes and interfaces responsible for the parsing of time series.

  - **ArrivalRateReader**: Provides functionality to read arrival rates from an arrival rate file. Can read either a single arrival rate at a given time, or returns a list of all *ArrivalRateTuples* contained in the file.
  - **IDlimArrivalRateReader**: Interface for an arrival rate reader. Must be implemented by a reader for the *Extractor* extension point.
  - **DefaultArrivalRateReader**: A default implementation of *IDlimArrivalRateReader*. Is able to read arrival rate files of the same format as produced by the arrival rate file exporter.
  - **RequestTimeSeriesReader**: Provides functionality to parse a request time-stamp trace into an file containing the arrival rates per second for each second.

2. **DLIM Generator Edit**

   This plug-in contains the providers used by the editor, which provide display specific information, such as the display images and labels of model elements.

3. **DLIM Generator Editor**

   The *dlim.editor* plug-in contains all GUI elements and their utilities. It also contains implicit modeling process knowledge in its GUI.

4. **DLIM Exporter**

   The *dlim.exporter* plugin offers three default implementations of the *dlim.generator* plugin's *dlim.exporter. IDlimExporter* interface and the *exporter* extension point:

   - **DlimArrivalRateExporter**: Exports an arrival rate time series.
   - **DlimEqualDistanceRequestStampExporter**: Exports request time stamps with an equal distance from one another within each sampled arrival rate interval.
   - **DlimUniformRequestStampExporter**: Exports request time stamps with a uniform random sampling within each sampled arrival rate interval.

5. **DLIM Extractor**

   The *dlim.extractor* plug-in offers two default implementations of the *dlim.extractor.IDlimExtractor* interface and the *extractor* extension point:

   - **PeriodicProcessExtractor**: Extracts a DLIM instance based on the Periodic DLIM Extraction Processes (p-DLIM).
   - **SimpleProcessExtractor**: Extracts a DLIM instance based on the Simple DLIM extraction process (s-DLIM).

   Both *extractor* extension point implementations in this plug-in use the provided default *dlim.reader.ArrivalRateReader* provided by the *dlim.generator* plug-in.

5 Conclusions

This paper provides a detailed description of LIMBO: A toolkit for creating and editing of DLIM instances. By enabling the flexible handling of load intensity profiles, LIMBO addresses a strong need in the areas of benchmarking and elastic capacity management. We describe the features of LIMBO and summarize the use cases and fields of possible application. We also provide a detailed description of LIMBO's architecture with a focus on extension points and functionality provided for use in other projects.

References

[Bie12] Tillmann Carlos Bielefeld. Online performance anomaly detection for large-scale software systems, 2012.

[BLY+10] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, and David A. Patterson. Rain: A Workload Generation Toolkit for Cloud Computing Applications. Technical Report UCB/EECS-2010-14, EECS Department, University of California, Berkeley, Feb 2010.

[Hal08] Emily H Halili. *Apache JMeter: A Practical Beginner's Guide to Automated Testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[MBM13] Marcelo R.N. Mendes, Pedro Bizarro, and Paulo Marques. FINCoS: Benchmark Tools for Event Processing Systems. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 431–432, New York, NY, USA, 2013. ACM.

[MK12] Aleksandar Milenkoski and Samuel Kounev. Towards Benchmarking Intrusion Detection Systems for Virtualized Cloud Environments. In *Proceedings of the 7th International Conference for Internet Technology and Secured Transactions (ICITST 2012)*, pages 562–563, New York, USA, December 2012. IEEE.

[VHNC10] Jan Verbesselt, Rob Hyndman, Glenn Newnham, and Darius Culvenor. Detecting trend and seasonal changes in satellite image time series. *Remote Sensing of Environment*, 114(1):106 – 115, 2010.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 247–248, New York, NY, USA, 2012. ACM.

[vK14] Jóakim v. Kistowski. Master's Thesis: Modeling Variatons in Load Intensity Profiles, March 2014.

[vKHK14a] Jóakim Gunnarson von Kistowski, Nikolas Roman Herbst, and Samuel Kounev. LIMBO: A Tool For Modeling Variable Load Intensities. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, ICPE '14, pages 225–226, New York, NY, USA, March 2014. ACM.

[vKHK14b] Jóakim Gunnarson von Kistowski, Nikolas Roman Herbst, and Samuel Kounev. Modeling Variations in Load Intensity over Time. In *Proceedings of the 3rd International Workshop on Large-Scale Testing (LT 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, pages 1–4, New York, NY, USA, March 2014. ACM.

[WHGK14] Andreas Weber, Nikolas Roman Herbst, Henning Groenda, and Samuel Kounev. Towards a Resource Elasticity Benchmark for Cloud Environments. In *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (HotTopiCS 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. ACM, March 2014.

[ZF13] Netanel Zakay and Dror G. Feitelson. Workload resampling for performance evaluation of parallel job schedulers. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 149–160, New York, NY, USA, 2013. ACM.

Approaching the Cloud: Using Palladio for Scalability, Elasticity, and Efficiency Analyses*†

Sebastian Lehrig
Software Engineering Chair
Chemnitz University of Technology
Straße der Nationen 62
09107 Chemnitz, Germany
sebastian.lehrig@informatik.tu-chemnitz.de

Matthias Becker
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
matthias.becker@upb.de

Abstract: In cloud computing, software architects develop systems for virtually unlimited resources that cloud providers account on a pay-per-use basis. Elasticity management systems provision these resource autonomously to deal with changing workloads. Such changing workloads call for new objective metrics allowing architects to quantify quality properties like scalability, elasticity, and efficiency, e.g., for software design analysis. However, analysis approaches such as Palladio so far did not support these novel metrics, thus rendering such analyzes inefficient.

To tackle this problem, we (1) extended Palladio's simulation approach SimuLizar by additional metrics for scalability, elasticity, and efficiency and (2) integrated the Architectural Template language into Palladio allowing architects to model cloud computing environments efficiently. A novel analysis process guides software architects through these new capabilities. In this paper, we focus on illustrating this new process by analyzing a simple, self-adaptive system.

1 Introduction

In cloud computing, software architects develop applications on top of compute environments being offered by cloud providers. For these applications, the amount of offered resources is virtually unlimited while elasticity management systems provision resources autonomously to deal with changing workloads. Furthermore, providers bill provisioned resources on a per-use basis [AFG+10]. As a consequence of these characteristics, architects want their applications to use as few resources as possible in order to save money while still maintaining the quality requirements of the system. Quality properties that focus directly on these aspects are scalability, elasticity, and efficiency [BLB15, HKR13].

These quality properties need to be quantified for requirements engineering and software design analysis by means of suitable metrics. For instance, cloud consumers and cloud

*The research leading to these results has received funding from the EU Seventh Framework Programme (FP7/2007-2013) under grant no 317704 (CloudScale).

†This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

providers need to negotiate service level objectives (SLOs), i.e., metrics and associated thresholds [EMP13]. Such SLOs have to consider characteristics like changing workloads ("how fast can an application adapt to a higher workload?") and pay-per-use pricing ("how expensive is serving an additional consumer?"). However, architecture-level analysis approaches so far did not support these novel metrics. Therefore, software architects currently cannot efficiently analyze their designs at the architectural level and potentially have to implement and try all reasonable design variants for making informed design decisions. Such an approach leads to high effort and high development costs in the end.

In related work, design-time engineering methods for analyzing performance properties exist. These approaches currently have a limited support for scalability, elasticity, and efficiency analyses. Scalability analyses (e.g., Palladio [BKR09]) are semi-automated, i.e., based on a series of manually conducted and interpreted performance analyses. Elasticity analyses (e.g., Palladio's simulation approach SimuLizar [BBM13]) allow to model and analyze self-adaptations, typically used by cloud computing environments, but have a high modeling effort. Efficiency analyses (e.g., CDOSim [FFH12]) require an implemented SaaS application to determine the most cost-efficient cloud computing environment but are limited to IaaS environments and lack support for early design-time analyses.

To tackle these problems, we (1) extended SimuLizar [BBM13] by additional metrics for scalability, elasticity, and efficiency and (2) integrated the Architectural Template language [Leh13] into Palladio allowing architects to model cloud computing environments efficiently. A novel analysis process guides software architects through these new capabilities.

The contribution of this paper is a tool-based illustration of this new process. For our illustration, we analyze a simple, self-adaptive system.

This paper is structured as follows. We introduce the simple, self-adaptive system we use as a running example in Sec. 2. In Sec. 3, we overview our novel process for analyzing cloud computing systems regarding scalability, elasticity, and efficiency. Afterwards, we illustrate this process and accompanying tools by applying our running example to this process (Sec. 4). We conclude the paper with a summary and an outlook on future work in Sec. 5.

2 Running Example: A Simplified Online Book Shop

As an example scenario, we consider a simplified online book shop. An enterprise assigns a software architect to design this shop, given the following requirements:

R_{fct}**: Functionality** In the shop, customers shall be able to browse and order books.

R_{scale}**: Scalability** The enterprise expects an initial customer arrival rate of 100 customers per minute. It further expects that this rate will grow by 12% in the first year, i.e., increase to 112 customers per minute. In the long run, the shop shall therefore be able to handle this increased load without violating other requirements.

R_{elast}**: Elasticity** The enterprise expects that the context for the book shop repeatedly

changes over time. For example, it expects that books sell better around Christmas while they sell worse around the holiday season in summer. Therefore, the system shall proactively adapt to anticipated changes of the context, i.e., maintain a response time of 3 seconds or less as well as possible. For non-anticipated changes of the context, e.g., peak workloads, the system shall re-establish a response time of 3 seconds or less within 10 minutes once a requirement violation is detected.

R_{eff}**: Efficiency** The costs for operating the book shop shall only increase (decrease) by \$0.01 per hour when the number of customers concurrently using the shop increases (decreases) by 1. In other words, the marginal cost of the enterprise for serving an additional customer shall be \$0.01.

Requirements R_{scale}, R_{elast}, and R_{eff} are typical reasons to operate a system in an elastic cloud computing environment [HKR13], i.e., an environment that autonomously provisions the required amount of resources to cope with contextual changes. Thus, the software architect designs the shop as a 3-layer Software as a Service (SaaS) application operating in a rented Infrastructure as a Service (IaaS) cloud computing environment that provides replicable virtual servers (see Fig. 1). The three layers involve the typical layers of web applications: presentation, application, and data layer.

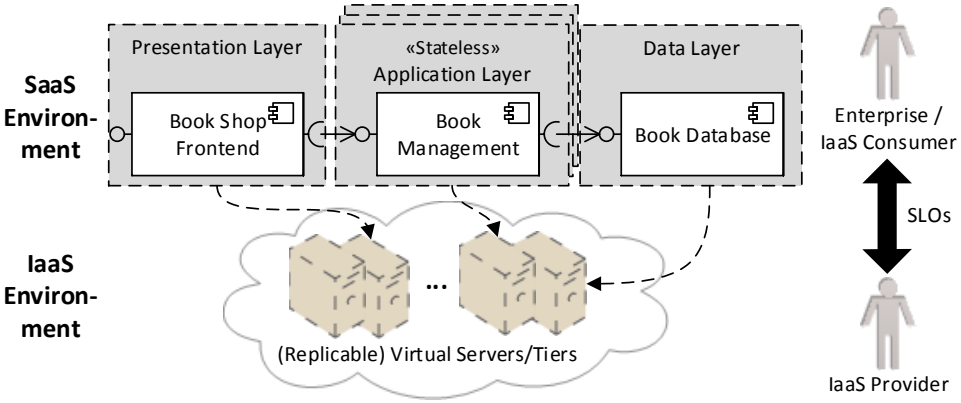


Figure 1: Overview of the simplified online book shop.

The architect investigates possible variants of the 3-layer architectural style. First, the architect considers a 3-layer/3-tier architecture [TMD09]. In a 3-layer/3-tier architecture, the three layers of a 3-layer architecture (presentation, application, and data) are allocated to three different tiers. In an elastic IaaS environment, these tiers are represented by different replicable virtual servers. Second, the architect considers the SPOSAD architectural style [Koz11], a 3-layer variant with an application layer that can safely be replicated to foster scalability. The SaaS middle layer has to be stateless to achieve this safe replication.

Now, the software architect would like to know whether the planned online shop should be realized according to (a) a 3-layer/3-tier architecture, (b) the SPOSAD architectural style, or (c) neither of the two. The architect wants to decide based on whether the scalability (R_{scale}), elasticity (R_{elast}), and efficiency (R_{eff}) requirements will be met by the

finally implemented application. In other words, the architect would like to conduct an architecture-level what-if analysis for the available options to make an informed decision.

However, as current analysis tools do not support metrics for scalability, elasticity, and efficiency in the context of cloud computing. Therefore, the architect cannot efficiently analyze the book shop at the architectural level and, therefore, potentially has to implement and try all considered variants (leading to high effort and high costs).

3 Process

In this section, we propose a novel high-level process for modeling and analyzing cloud computing based systems. Our process supports architects in conducting architecture-level analyses for scalability, elasticity, and efficiency. Moreover, we integrated a set of analysis tools into our process for making such analyzes highly efficient: Palladio [BKR09], Architectural Templates [Leh13], and SimuLizar [BBM13].

Architectural Templates and SimuLizar are extensions to the Palladio tool suite. Architectural Templates (ATs) are a mean to express architectural blueprint that architects can efficiently use and refined for a broad range of software systems. ATs can be specified and reused for various architectural styles, like "3-layer/3-tier" and "SPOSAD", and can be analyzed with SimuLizar. SimuLizar provides modeling capabilities for self-adaptive systems, i.e., different views for a system and its reconfiguration, as well as a simulation-based scalability-, elasticity-, and efficiency-analysis.

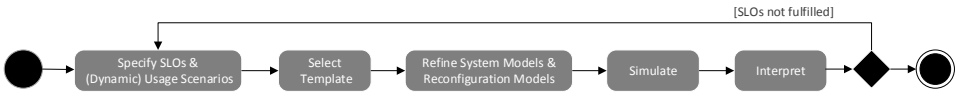


Figure 2: High-level process for applying Palladio in cloud computing scenarios.

Figure 2 illustrates our high-level process. The process starts with specifying service-level objectives (SLOs) and (dynamic) usage scenarios that both formalize the requirements of the system to be developed (e.g., the ones of our example scenario in Sec. 2). Next, the software architect has to select an appropriate Architectural Template. The Architectural Template has then to be further refined by application-specific parts, i.e, interfaces, components, and service effect specifications have to model the target systems structure and behavior. Reconfiguration rules, defined by the Architectural Template, can optionally be refined as well. To validate whether the SLOs for the target system are met, the system can be simulated with SimuLizar. The result of the simulation is a set of measurements for different metrics as defined in [BLB15]. The software architect has now to interpret these measurements and check whether the modeled target system fulfills the SLOs. If not, the process continues with the first step to incrementally refine the SLOs and the system until SLOs can be met.

4 Application

In this section, we apply the book shop scenario of Sec. 2 to our novel process as described in Sec. 3. This application serves as a first proof-of-concept evaluation. We describe each process step in a separate subsection.

4.1 Specify SLOs & (Dynamic) Usage Scenarios

Based on the requirements of the book shop scenario, we derive and specify SLOs and (dynamic) usage scenarios.

For the specification of SLOs, we use a dedicated SLO language of our tool suite. Our language allows to organize SLO specifications in repositories as shown in Fig. 3. For example, we derived a "3 Seconds Response Time SLO" based on R_{elast} where the requirement is checked against a threshold of three seconds. Our SLO of Fig. 3 makes this threshold explicit as can be seen within the properties view. The simulation can later-on make use of such information.

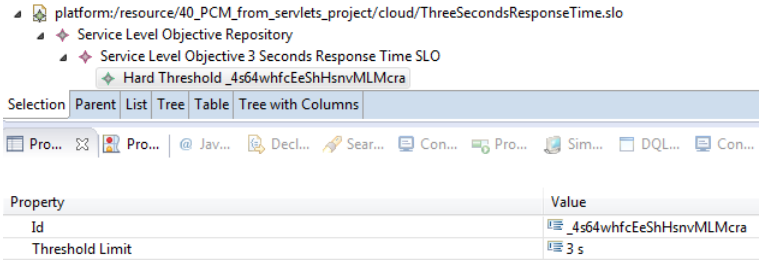


Figure 3: Example SLO specification as derived from R_{elast}.

For the specification of dynamic usage scenarios, we integrated LIMBO [vKHK14], a tool for load intensity modeling, into our tool suite. Before our integration, usage scenarios had to be specified in a static manner, i.e., could not vary over time. Therefore, scenarios like we described for the book shop scenario could not be realized, e.g., to vary load around Christmas. Because LIMBO allows to model such time-dependent changes in workloads, we extended our simulation such that LIMBO's dynamic changes in workloads can be applied on Palladio's usage scenarios.

For example, we modeled a change of arrival rates over one year for the book shop scenario as shown in Fig. 4. The arrival rates generally increase from 100 users per second to 112 users per second as that is the expected trend for one year. We additionally modeled a peak around Christmas as well as lower arrival rates around the holiday season in summer. Finally, we added some noise to reflect a realistic use of web applications. Our extended simulation later-on follows this specification for the arrival rate.

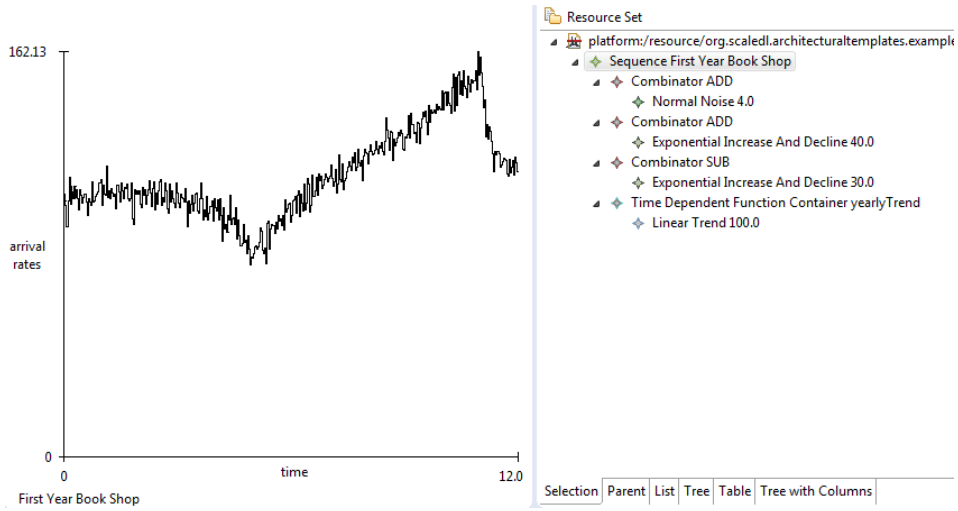


Figure 4: Example LIMBO specification as derived from R_{scale} and R_{elast}.

4.2 Select Template

The template-based design process starts with selecting an Architectural Template (AT) from a repository of available ATs. We created such a repository for the CloudScale project[1].

The architect of the book shop scenario directly models the system in two variants, as a 3-layer/3-tier architecture and as a SPOSAD architecture. For each variant, the architect selects an appropriate AT from the repository. As an example, the AT specification for SPOSAD is illustrated in Fig. 5. This AT specifies the different roles of SPOSAD (SPOSAD itself; presentation, middle, data layers) and a completion. The latter is a transformation able to weave application-independent SPOSAD information into the model of the book shop. For example, the replication logic in the form of adaptation rules is application-independent and can, thus, be included in the completion of the AT. ATs particularly provide the means to parametrize such completions, e.g., by the concrete condition when replication shall be triggered.

Having decided to choose a particular AT for system design, architects can apply that AT to their system. In Fig. 6, we illustrate such an application for the SPOSAD AT. The figure shows the PCM Profiles view; a special view for listing applications of Palladio's profile mechanism. Each AT role can be applied via this mechanisms, thus, allowing the architect to assign the different AT roles to the corresponding entities. Accordingly, he has top assign the SPOSAD role to the system itself and each layer role to the corresponding component assembly within the system. Creating and refining such component assemblies is the task of the next process step.

[1]The repository is available at `github.com/CloudScale-Project`; the process to engineer ATs is described by Lehrig [Leh13].

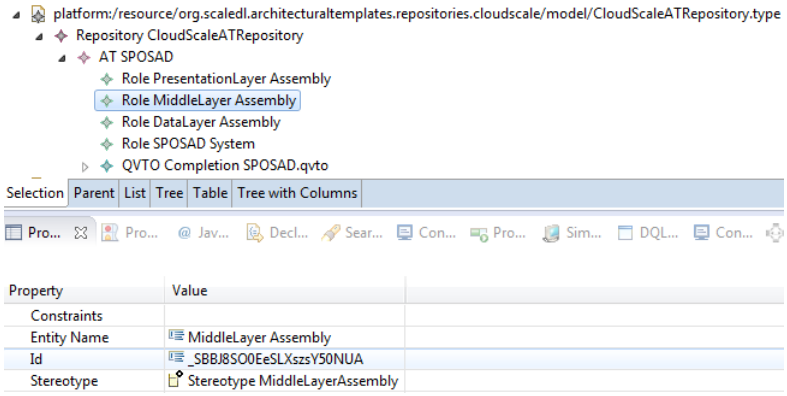


Figure 5: Example AT for SPOSAD. The AT includes a role for the system and three roles for assigning the layers of a 3-layer architecture to component assemblies. The SPOSAD completion weaves application-independent SPOSAD information into the model.

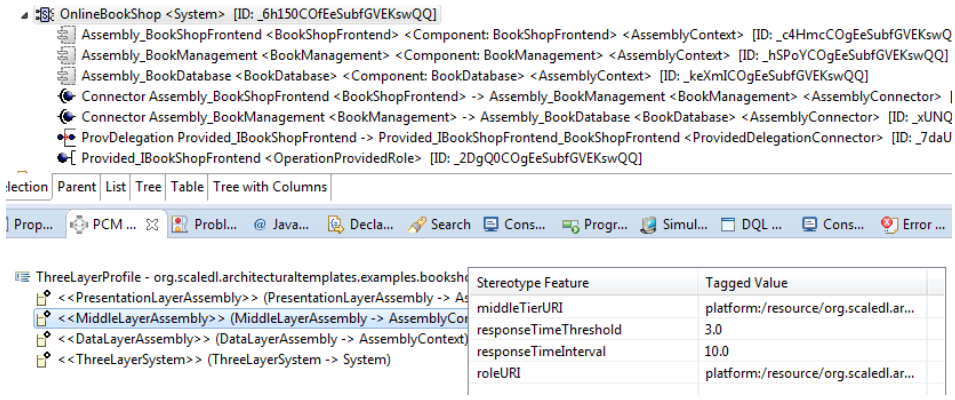


Figure 6: Example application of the SPOSAD AT. Each AT role is assigned to corresponding PCM entities using its profile mechanisms. The middle layer assembly profile allows for specifying response time thresholds and intervals for triggering self-adaptations. In the example, response time is averaged in intervals of 10 seconds and checked against a 3 seconds threshold.

4.3 Refine System Models & Reconfiguration Models

Now that the software architect has configured two system models to make use of ATs (3-layer/3-tier AT and SPOSAD AT), the architect has to assign all AT roles to application-dependent component assemblies. For the presentation and data layer, the architect can assemble the same repository components for both design variants. The two variants, 3-layer/3-tier AT and SPOSAD AT, do not constrain such components differently. For the application layer, the architect has to assemble a stateless variant of the book management component to the system if following the SPOSAD AT. Here, the architect has to follow the constraints of the SPOSAD architectural style. For the case of the 3-layer/3-tier AT, no such constraints limit the design of the book management component. Both variants, stateless and statefull, are allowed here.

Fig. 7 illustrates the system model as designed by the architect. The system model is similar for both variants; only the encapsulating component of the book management assembly may be different as explained above. Having these assemblies available, the architect can assign the remaining AT roles to these. This assignment is shown in Fig. 6 (note the three profile applications for each layer).

For the application layer of the SPOSAD AT, the architect has an additional option to parametrize the adaptation rules used for replication. As illustrated in Fig. 6 (bottom, right), the application layer profile provides tagged values for specifying replication threshold and observation intervals.

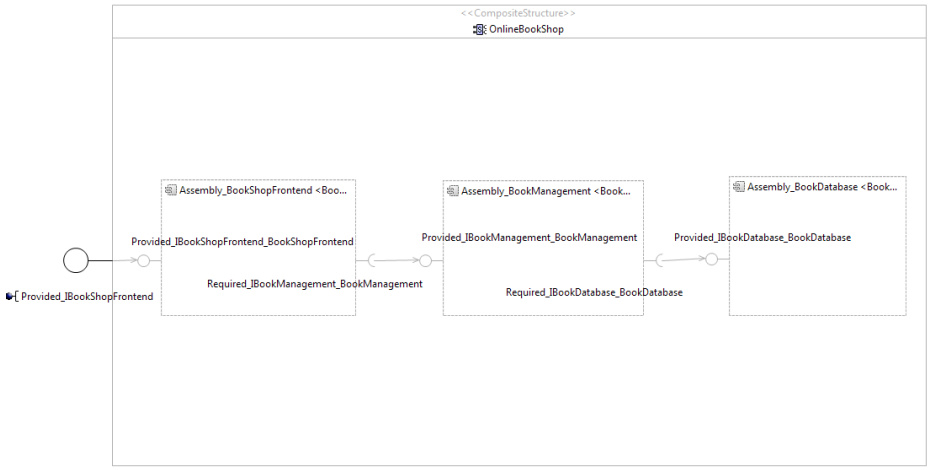


Figure 7: The system model of the book shop scenario

4.4 Simulate

For our simulation, we reimplemented and improved the Experiment Automation framework, initially developed by Merkle [Mer11]. Experiment Automation is now able to start several SimuLizar [BBM13] simulations, each time varying model parameters in order to measure our novel metrics. For instance, we measure user capacity (prerequisite for our scalability range metric, cf. [BLB15]) by varying the number of users within the system over several simulation runs. For details about Experiment Automation, we refer to our developer guide[2].

We configured an appropriate Experiment Automation model for the book shop scenario. Our model references SimuLizar as simulation tool, due to its self-adaptation capabilities.

4.5 Interpret

In this section, we exemplify some first new metric measurements that are now supported. Implementing the full range of our proposed metrics (and even more) is part of our future work.

In Fig. 8, we illustrate a result for our user capacity metric (scalability metric). For the book shop implemented as 3-layer/3-tier variant, we observe a user capacity of 30. Therefore, if more than 30 users reside within the system, SLOs (and corresponding requirements) are violated.

The architect of the book shop investigates the root causes for this insufficient user capacity; a higher number was expected. By investigating the CPU utilization of the application

[2]Quality Analysis Lab (QuAL): Software Design Description and Developer Guide - Version 0.2: `https://svnserver.informatik.kit.edu/i43/svn/code/QualityAnalysisLab/Documentation/trunk/org.palladiosimulator.qual.docs/QualityAnalysisLab.pdf` (User: anonymous; Password: anonymous; Visited on 31/10/2014).

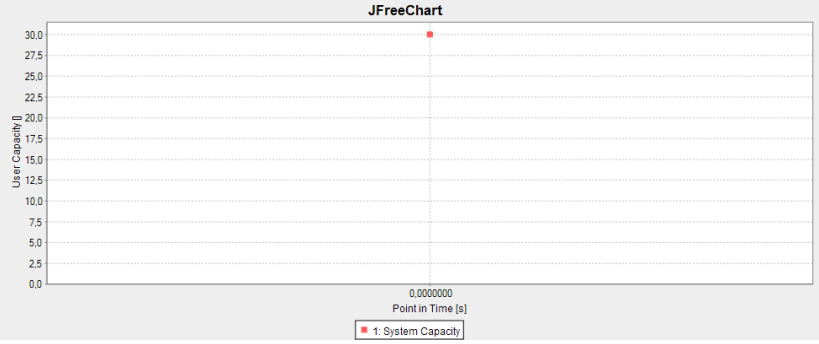


Figure 8: User capacity is 30 for the book shop scenario implemented as 3-layer/3-tier variant

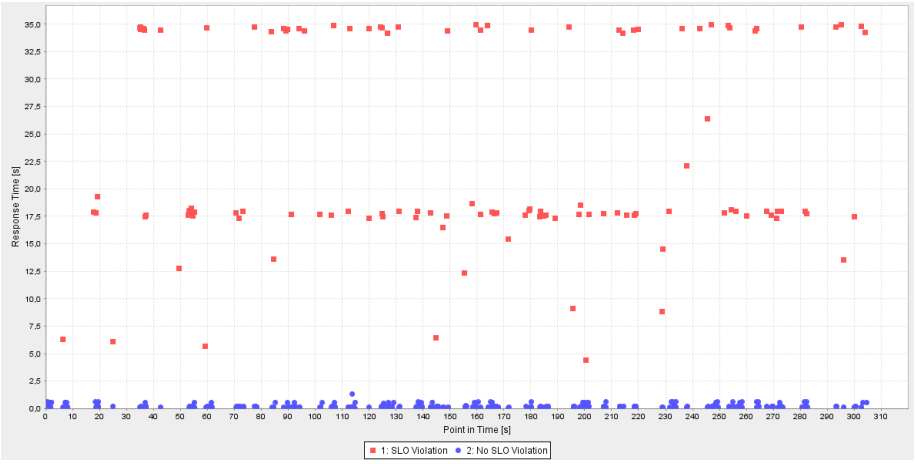


Figure 9: SLO violations over time for the book shop scenario implemented as SPOSAD variant

tier, the architect figures out that the application layer component overloads the CPU. Because the demands to the CPU are realistically modeled, the architect decides to apply the SPOSAD variant instead (SPOSAD ensures a scalable application layer).

The architect figures out that the capacity of the new system is around 500, which seems to be fine. However, by investigating the measurements for our "SLO violations over time" elasticity metric (Fig. 9), the architect observes that there are too many SLO violations throughout the year.

The software architect now suspects that the database is the new bottle neck resource. The architect therefore remodels the system with a faster HDD on the data tier and reexecutes the simulation. Now, no SLOs are violated anymore. Eventually, the architect therefore suggests to implement the book shop following the SPOSAD architectural style and with a fast HDD for the data tier. The architect additionally provides the 1-year operation costs for this variant based on dedicated cost metrics that our simulation now supports.

5 Conclusions

In this paper, we show how we integrated novel metrics for quality properties of cloud computing systems into Palladio, accompanied by a novel process and a more efficient modeling language. Following our process, we where able to efficiently analyze a first, simple example regarding scalability, elasticity, and efficiency.

Our new process and tool suite helps software architects in efficiently analyzing applications that shall operate within cloud computing environments. For scientists, our tools are especially interesting because their capabilities for scalability, elasticity, and efficiency analysis open a plethora of new possibilities for engineering systems.

In future work, we plan to further improve the usability of our tools because several editors

are in early-stage development. Afterwards, we want to evaluate these tools and our process with more extensive examples. We also plan to optimize our analysis by building up on already gained measurements and only analyzing changed/additional parts of system models (scalability, elasticity, and efficiency analysis composition).

References

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[BBM13] Matthias Becker, Steffen Becker, and Joachim Meyer. SimuLizar: Design-Time Modelling and Performance Analysis of Self-Adaptive Systems. In *Proceedings of Software Engineering 2013 (SE2013), Aachen*, 2013.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1), January 2009.

[BLB15] Matthias Becker, Sebastian Lehrig, and Steffen Becker. Systematically Deriving Quality Metrics for Cloud Computing Systems. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, New York, NY, USA, 2015. ACM. Accepted for publication.

[EMP13] Thomas Erl, Zaigham Mahmood, and Ricardo Puttini. *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall, 2013.

[FFH12] Florian Fittkau, Sören Frey, and Wilhelm Hasselbring. CDOSim: Simulating Cloud Deployment Options for Software Migration Support. In *MESOCA '12*, page 3746, 2012.

[HKR13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity: What it is, and What it is Not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013), San Jose, CA, June 24–28*, 2013.

[Koz11] Heiko Koziolek. The SPOSAD Architectural Style for Multi-tenant Software Applications. In *Proc. 9th Working IEEE/IFIP Conf. on Software Architecture*, pages 320–327. IEEE, July 2011.

[Leh13] Sebastian Lehrig. Architectural Templates: Engineering Scalable SaaS Applications Based on Architectural Styles. In *Proceedings of the MODELS 2013 Doctoral Symposium co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, volume 1071, pages 48–55, Miami, USA, 2013. CEUR-WS.org.

[Mer11] Philipp Merkle. Comparing Process- and Event-oriented Software Performance Simulation. Master's thesis, Karlsruhe Institute of Technology (KIT), Germany, 2011.

[TMD09] R.N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[vKHK14] Jóakim Gunnarson von Kistowski, Nikolas Roman Herbst, and Samuel Kounev. LIMBO: A Tool For Modeling Variable Load Intensities. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, ICPE '14, pages 225–226, New York, NY, USA, March 2014. ACM.

Towards Performance Awareness in Java EE Development Environments

Alexandru Danciu[1], Andreas Brunnert[1], Helmut Krcmar[2]
[1]fortiss GmbH
Guerickestr. 25, 80805 München, Germany
{danciu, brunnert}@fortiss.org
[2]Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
krcmar@in.tum.de

Abstract: This paper presents an approach to introduce performance awareness in integrated development environments (IDE) for Java Enterprise Edition (EE) applications. The approach predicts the response time of EE component operations during implementation time and presents these predictions within an IDE. Source code is parsed and represented as an abstract syntax tree (AST). This structure is converted into a Palladio Component Model (PCM). Calls to other Java EE component operations are represented as external service calls in the model. These calls are parameterized with monitoring data acquired by Kieker from Java EE servers. Immediate predictions derived using analytical techniques are provided each time changes to the code are saved. The prediction results are always visible within the source code editor, to guide developers during the component development process. In addition to this immediate feedback mechanism, developers can explicitly trigger a more extensive response time prediction for the whole component using simulation. The paper covers the conceptional approach, the current state of the implementation and sketches for the user interface.

1 Introduction

Detecting performance problems of applications systems is essential before their release to the field. However, evaluating the performance of application systems in terms of response time, resource utilization and throughput is a difficult task. Performance tests require realistic environments comprising hardware, middleware, utilized components, test data and a test workload. Load testing is often one of the last steps performed during a development process, even though performance problems can be solved easier, the earlier they are detected.

Performance awareness represents the ability to detect performance problems and to react to them [T̊14]. One of the core targets of this concept is supporting developers with insights on the performance of code they are currently developing. This paper presents an approach to introduce performance awareness in integrated development environments (IDE) for Java Enterprise Edition (EE) applications. Java EE supports the implementa-

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

tion of component-based software systems. The Java EE specification defines application component types such as applets, servlets and Enterprise JavaBeans [LD13]. The emphasis of component-based development is on the specification of loosely coupled independent components to enable separation of concerns and reuse across the system. Component developers reuse other components to implement the required functionality. The performance of a new component depends among others on the performance of reused components [Koz10]. Therefore component developers are facing questions such as:

- Are the service-level agreements (SLA) imposed to my component violated by reusing a specific component?
- Can the SLAs imposed to my component be achieved with the current component implementation?
- Does a particular change in the control flow of my component lead to an SLA violation?
- How is the performance of my component changing for varying workloads?

Answering these questions is an increasingly difficult task, due to three main factors: application system architecture, system life cycle and IT governance [BVD+14]. Complex architectures often lead to geographical, organizational, cultural and technical variety. Developers lack the knowledge about the structure and the deployment of reused components. Components are subject to a constant iteration between development and operation. It is difficult for developers to maintain an overview of the performance behavior of components. Additionally, components are assigned to different organizational units. Accessing monitoring data is thus more difficult for the developer. From a technical point of view, the developer needs experience in the performance engineering domain and in using corresponding tools. This article proposes an automated and integrated approach to answer these questions. The approach predicts the response time of component operations during implementation time and presents the results within the IDE of the developer.

The remainder of this paper is organized as follows. Section 2 provides an overview of the approach and describes the three main phases in detail. Section 3 describes existing research related to this approach. Section 4 concludes this article and describes future research directions.

2 Developer Performance Awareness Approach

The goal of the approach presented in this paper is to support developers with response time estimations for Java EE component operations. The main phases and the architecture of the proposed approach are shown in Figure 1. First, performance data of running Java EE components is collected from existing application deployments (1). Our approach is based on the assumption that new components will reuse existing ones to some extent. The collected data is then aggregated over different component instances, versions and user workloads (2). Response time estimations for new components are derived by using Palladio Component Models (PCM). These PCM models are generated based on the source code of the component operations representing the control flow with an emphasis

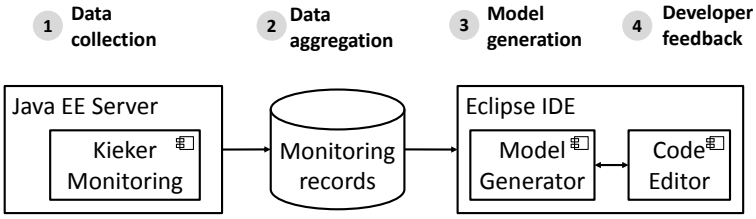


Figure 1: Overview of the approach (structure adapted from [BVK13])

on external calls (3). Two types of estimations can be triggered (4): an immediate estimation after the developer saves a change to the code and an explicitly triggered simulation. The immediate estimation is performed using an analytical solver and stochastic expressions representing the performance behavior of reused components. The prediction results are visible within the source code editor. Simulations are intended to provide more precise predictions and are performed using performance curves [WHW12]. Simulation results will be displayed in a separate view. The data collection and aggregation phases are executed periodically. The model generation and developer feedback phases are triggered by the developer. Each phase is described in detail below.

2.1 Data Collection

The response time behavior of running components within the application landscape is collected using the Kieker framework [Pro14]. Applications are instrumented using aspect-oriented programming. Using the adaptive monitoring feature [Pro14], data collection is activated for specific component operations. Monitoring records are passed by a new monitoring writer via a web-service call to a database application that serves as a backend for the approach presented in this work. This database application serves as a central repository for monitoring records. The web-service deserializes the record and stores it to a relational database. Monitoring records include response times of component operations and the current workload of the application in terms of queue length and resource utilization. The monitoring writer stores for each record an additional description of the deployment from which the measurement was obtained. This description includes information about the host and the application server instance. Information about the deployment is retrieved by the monitoring writer from local configuration files. The monitoring writer also stores meta-data of the application binaries provided by a central build tool so that performance measurements can be related to component versions.

2.2 Data Aggregation

The response time behavior of component operations can be collected from multiple application servers and for different workloads. Also, different versions of a component can be deployed within the application landscape. The aim of this step is to aggregate individual records to performance curves and stochastic expressions describing the response time behavior of operations in dependence on a set of input parameters such as workload characteristics. Aggregation is performed using statistical methods and measurements are aggregated for each component version separately. Java functionality is delivered as deployable units by assembling source code to Java Archives, Web Archives or Enterprise Archives. The current version of the code, identified by a revision number, is exported from a repository and assembled. The resulting archive is then deployed to an application server. However, neither the revision, the assembly nor the deployment suggest which component version is addressed by a performance measurement. The source code of a component might remain constant over multiple revisions, assemblies and deployments. The proposed approach stores for each component the revision numbers which contained changes to its source code. This revision number is then compared to the revision contained in the deployable unit. A central build tool stores the revision number of the assembled code in a property file within the deployable unit. Measurements can thus be assigned to specific component versions. The results of this step are provided over a web-service interface to clients (i.e., IDEs).

2.3 Model Generation

The source code of a new component and the performance data collected for any reused components are used to generate a component-based performance model. PCM is used as the meta-model for the generated models.

The main factors influencing the performance of reusable software components are the component implementation, required services, deployment platform, usage profile and resource contention [Koz10]. The focus of this approach are the component implementation and the required services. The component implementation is represented in terms of the control flow of individual component operations. Required services are modeled explicitly in the model as external calls. The deployment platform is assumed to be constant for all components. The usage profile is represented in terms of the response times of reused components in dependence on the workload. The impact of input parameters and resource contention are not addressed by the approach.

The proposed approach first generates a PCM repository model which specifies the currently investigated component. The model generation is based on the approaches of Kappler et al. [KKKR08] and Becker et al. [BHT+10]. Source code of component operations is parsed and represented as an abstract syntax tree (AST). Each AST is then converted to a Resource Demanding Service Effect Specifications (RDSEFF). An RDSEFF specifies the behavior of a single component operation. Calls to reused components are modeled as

external call actions. The response time behavior of external calls is modeled either using stochastic expressions or performance curves depending on whether the model is used as input for an analytical solver (stochastic expression) or a simulation engine (performance curve). This information is retrieved from the monitoring record database. Calls within the boundaries of the investigated component are modeled as internal action calls. Branches, loops and forks are also represented in the RDSEFF. Since the modeled component is under development, the probabilities of different execution flows are unknown. Therefore, branches are modeled having equal probabilities. Since binary code is not available for the investigated component, a dynamic analysis [KKKR08] of the component behavior can not be performed. Thus, resource demands of internal call actions are also not represented in the RDSEFF.

After creating the PCM repository model, the remaining PCM models that are required for a simulation, are also generated automatically. The PCM system model describes how components are combined and which interfaces they provide. The investigated component is represented as a single instance in the PCM system model. Each public component operation is modeled as an externally accessible interface. The PCM usage model describes a closed workload where all operations of the investigated component are called equally often. The hardware resources such as central processing units (CPU) available to the investigated component are specified within the PCM resource environment model. A resource environment model containing one server and a single CPU is generated. The investigated component is then mapped to this server within the PCM allocation model.

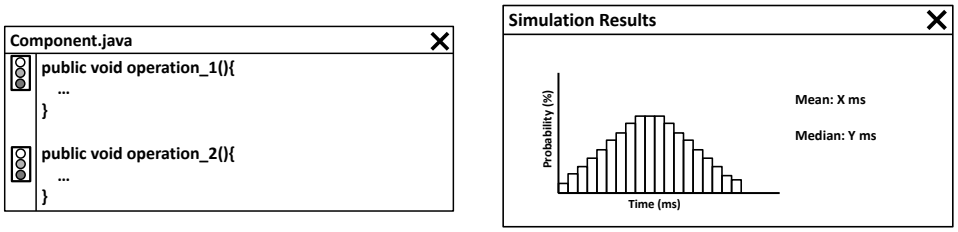
2.4 Developer Feedback

The main goal of the proposed approach is to integrate estimations of the expected response time of component operations in the development environment. Two types of feedback are envisioned:

1. An implicit estimation of the response time of component operations is automatically performed each time changes to the code are saved.
2. A detailed response time estimation of all component operations triggered by the developer.

Since changes to code are frequently saved, the immediate estimation must be executed very fast. The estimation is therefore performed using an analytical solver. Figure 2(a) shows how this type of feedback could look in the IDE. If the estimated response time exceeds a specific threshold, a notification will be displayed within the code editor. Thresholds are configured with default values and can be adjusted by the developer. Notifications are displayed as yellow and red traffic lights and are associated to the corresponding operation signatures.

The explicitly triggered response time estimation aims at providing more precise results. Therefore a simulator is used to process the PCM model. The response time behavior of reused components is modeled using performance curves. Figure 2(b) shows how this type of feedback could look in the IDE. Results are displayed as probability density function in



(a) Immediate feedback mockup

(b) Simulation results mockup

Figure 2: User interface mockups

a separate view. Important metrics such as mean and median values are also displayed.

3 Related Work

The presented approach is related to existing research on developer performance awareness and performance model generation. In the following, we discuss existing work that addresses these research areas.

Developer Feedback

Several approaches aim at providing feedback to developers. Weiss et al. [WWHM13] propose an approach for evaluating the performance of software artifacts based on tailored benchmarks applications during the implementation phase. The authors describe a scenario, how immediate feedback could be provided to developers in the IDE. Developers can either track the performance impact of changes to the implementation or compare different design alternatives. Performance estimations are displayed as numerical values and bar charts and in a separate view. The presented approach is applicable only to Java Persistence API services. Instructions on how to design benchmark applications and how to apply the approach to other components are provided.

Heger et al. [HHF13] present an approach to integrate performance regression root cause analysis into development environments. The change in performance between two revisions is displayed graphically as a function. Methods causing the regression are presented to the developer as a graph. The approach employs unit tests to gather performance measurements and thus provides no feedback on the performance expected in realistic environments.

Bureš et al. [BHK+14] propose the integration of performance awareness in the development life cycle of autonomic component ensembles. The authors describe how the design and operations phases can be augmented with performance-related activities such as formulating performance goals or collecting performance data. One of these activities aims at providing feedback to developers. The authors envision the presentation of performance measurements to the developer directly in the IDE. Feedback is provided graphically within a pop-up window. The approach is suitable for the presentation of historical

data and doesn't support a real-time interaction with the developer.

Automatic Performance Model Generation

Brosig et al. [BHK11] present a semi-automatic approach for extracting PCM models from Java EE applications based on monitoring data collected at run-time using WebLogic-specific monitoring tools. Call path tracing is employed to determine the control flow. Only executed paths are identified by the approach. Brunnert et al. [BVK13] present a similar approach that is applicable for all Java EE server products based on data collected from custom Servlet filters and EJB interceptors. Neither of the approaches support evaluating the performance of single components during the implementation phase.

Becker et al. [BHT+10] present an approach for reverse engineering Java applications based on static source code analysis. Similar to the approach proposed by Kappler et al. [KKKR08], source code is parsed to an AST which is then converted to an RDSEFFs. Krogmann et al. [KKR10] use static and dynamic analysis to reverse engineer Java applications. This approach requires the evaluated components to be executed in testbeds. Parametric dependencies between input parameters and the control flow are derived using genetic search.

4 Conclusion and Future Work

This paper proposed an approach to provide feedback on the estimated response time of Java EE components to developers within the IDE. Using the approach does not require any knowledge about reused components or experience in the performance engineering domain. Developers are not required to make efforts for obtaining access to performance data and do not have to employ additional tools for processing these data. Feedback is provided automatically to the developer and requires no additional effort. The approach focuses on the performance impact of component reuse and ignores the resource demand of the investigated component. Therefore, the application of this approach is only useful if component reuse exists. The prediction of response times in dependence on the workload is only possible if the behavior of reused components was measured under various workloads.

Future research will evaluate this approach within a case study. Additionally, the approach needs to enable the developer to refine generated PCM models, for example by specifying probabilities for branches within an RDSEFF. Refinements could be performed by annotating code.

References

[BHK11] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 183–192, Nov 2011.

[BHK+14] Tomáš Bureš, Vojtěch Horký, Michał Kit, Lukáš Marek, and Petr Tůma. Towards Performance-Aware Engineering of Autonomic Component Ensembles. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.

[BHT+10] Steffen Becker, Michael Hauck, Mircea Trifu, Klaus Krogmann, and Jan Kofron. Reverse Engineering Component Models for Quality Predictions. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track*, pages 199–202. IEEE, 2010.

[BVD+14] Andreas Brunnert, Christian Vögele, Alexandru Danciu, Matthias Pfaff, Manuel Mayer, and Helmut Krcmar. Performance Management Work. *Business & Information Systems Engineering*, 6(3):177–179, 2014.

[BVK13] Andreas Brunnert, Christian Vögele, and Helmut Krcmar. Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications. In MariaSimonetta Balsamo, WilliamJ. Knottenbelt, and Andrea Marin, editors, *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin Heidelberg, 2013.

[HHF13] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated Root Cause Isolation of Performance Regressions During Software Development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 27–38, New York, NY, USA, 2013. ACM.

[KKKR08] Thomas Kappler, Heiko Koziolek, Klaus Krogmann, and Ralf H. Reussner. Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. In *Software Engineering 2008*, volume 121 of *Lecture Notes in Informatics*, pages 140–154, Munich, Germany, February 18–22 2008. Bonner Köllen Verlag.

[KKR10] K. Krogmann, M. Kuperberg, and R. Reussner. Using Genetic Search for Reverse Engineering of Parametric Behavior Models for Performance Prediction. *IEEE Transactions on Software Engineering*, 36(6):865–877, Nov 2010.

[Koz10] Heiko Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634 – 658, 2010. Special Issue on Software and Performance.

[LD13] Bill Shannon Linda DeMichiel. Java Platform, Enterprise Edition (Java EE) Specification, v7. 2013.

[Pro14] Kieker Project. Kieker User Guide. Research report, April 2014.

[Tů14] Petr Tůma. Performance Awareness: Keynote Abstract. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 135–136, New York, NY, USA, 2014. ACM.

[WHW12] Alexander Wert, Jens Happe, and Dennis Westermann. Integrating Software Performance Curves with the Palladio Component Model. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 283–286, New York, NY, USA, 2012. ACM.

[WWHM13] Christian Weiss, Dennis Westermann, Christoph Heger, and Martin Moser. Systematic Performance Evaluation Based on Tailored Benchmark Applications. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 411–420, New York, NY, USA, 2013. ACM.

Identifying Semantically Cohesive Modules within the Palladio Meta-Model

Misha Strittmatter, Michael Langhammer
Institute for Data Structures and Program Organisation (IPD)
Karlsruhe Institute of Technology (KIT)
{strittmatter | langhammer}@kit.edu

Abstract: The Palladio Component Model (PCM) is a modeling language for component-based business information systems. Its primary and original focus lies on performance prediction. Its core meta-model is organized within a package structure within one meta-model.

Over time, the meta-model was extended, which allowed to model additional concerns. The extensions were made directly into the package structure, mostly without creating sub packages. Some cross-cutting concerns were placed inconsistently. This deteriorated the organizational structure of the meta-model, which negatively influences maintainability, understandability and extensibility.

To solve this, the meta-model should be restructured into meta-model modules. Within this paper, we identified concerns which are contained in the meta-model, to form a basis for the future modularization. This paper does not propose a definite modularization, but possible building blocks. What may be put as a single module or just a package within a module will be subject to future discussion.

1 Introduction

The Palladio meta-model is currently contained within one file and is subdivided into packages. This structure formed through the years, while the PCM was developed and extended. The top-level packages are partly aligned with the containment structure of the PCM submodels (e.g. repository, system). There are other top-level packages, which contain cross-cutting or general concepts (e.g. parameter, composition, entity). Further ones are concerned with quality dimensions (e.g. reliability, QoS Annotations). Some extensions distributed their new classes across packages, which were semantically fitting. This injecting as well as the scattering of new content is problematic, as some concepts cannot be easily associated with their concern.

Within the scope of the Palladio refactoring [SMRL13], the current all-enclosing structure is decomposed into a more modular one. The goal is to divide the Palladio meta-model into smaller, semantically cohesive, concern based meta-models (meta-model modules). From this, a small structural core can be formed. Further needed modeling constructs are regarded as extensions and can be loaded, if desired.

This modularization of the meta-model has several advantages. The meta-model becomes better maintainable and easier to extend and understand. When a Palladio model is created,

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

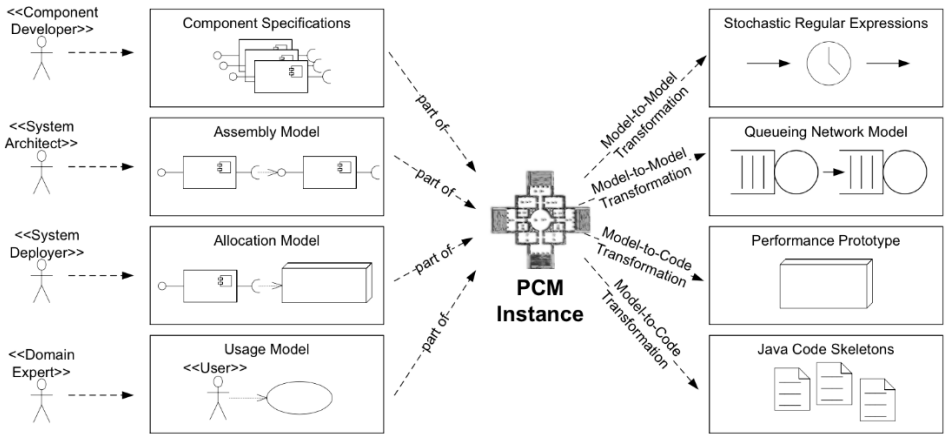


Figure 1: Simplified overview of the different PCM models that have to be modeled (from [RBB+11])

the model developer can choose the meta-model modules which are relevant to him and is not confused by the full complexity of content from all extensions. This is especially necessary, when the Palladio approach is extended to further quality attributes and even other domains than software engineering.

Within this paper, we identify concerns which are contained in the meta-model, to form a basis for the future modularization. This paper does not propose a definite modularization, but possible building blocks. What may be put as a single module or just a package within a module will be subject to future discussion.

This paper is structured as follows: In Section 2, we provide an introduction to the PCM and further literature references. In Section 3, the current structure of the Palladio meta-model is explained and illustrated. Section 4 describes how we approached the identification of the concerns, which are contained in the meta-model. Next, in Section 5, the semantically cohesive modules which were identified are presented and their interdependencies explained. Section 6 concludes the paper and states lessons learned. Finally, Section 7 outlines the next steps which are necessary for the modularization process.

2 Foundations

The PCM is a component model, that can be used to create a model of a software system and, amongst other, predict its performance without actually implementing the system. It has been introduced in various publications [BKR09, Koz08, RBB+11]. In order to predict the performance of a system users of the PCM have to model the system in six different models (see Figure 1).

First, a user has to create the Repository model. Within the Repository the components

and interfaces as well as the provided and required roles between them have to be defined.

In the second model, the System model, the composition of the components has to be defined. Moreover, provided and required roles of the whole system have to be defined.

The third model is the Service Effect Specification (SEFF) model. A SEFF, which is a behavior model, has to be specified for each method provided by a component. It contains control flow elements like resource demands and internal call actions as well as external call actions. These provided information is used by PCM to predict the performance of the system in a later step.

The fourth model is the so called Resource Environment Model. It specifies the resources which are available for the system, i.e. different servers, their computing and memory power as well as their network connection have to be defined.

The fifth model is the Allocation model. It specifies how components (of the system) are deployed on the resources.

The last model is the Usage model. The Usage model describes the number or arrival rate of users who are using the software system.

These six models act as input values for the PCM allowing to predict the performance of the system. The results of the prediction are the response time of the system and methods and resource utilization. The results can be visualized in, for example, time charts or pie diagrams.

The Palladio bench, which is the implementation of the PCM approach, is build using the Eclipse Modeling Framework (EMF) [SBPM08]. To allow performance prediction and code generation, model-to-model transformations and model-to-code generation are used.

3 Current Package and Dependency Structure

In this Section we give a brief overview about composition of the current Palladio Component meta-model. Currently, the meta-model is divided in several hierarchical packages. Table 1 gives an overview about the packages. Generally, the packages are divided according to the six models presented in Section 2. The Allocation package, for example, contains the classes that are used in the allocation model.

The Palladio meta-model also has dependencies to the Identifier, the Units and the Stochastic Expression (StoEx) meta-model. The Identifier meta-model only includes the class Identifier, which has the unmodifiable attribute ID. During the creation of an instance a (UUID) is created and assigned to the ID attribute. Hence, Identifier, and all of its subclasses, do have a UUID to identify the elements globally. The Units meta-model contains classes that specify different Units that are used in a SEFF. The StoEx meta-model includes classes to create stochastical expressions in a SEFF. Therefore, the StoEx model uses classes that allows the creation of new random variables.

Figure 2 shows the current package structure and the dependencies between the packages within the Palladio meta-model.

Package	Explanation
pcm	root-package that contains all other packages
core	contains the entity and composition packages
entity	contains all core classes that are shared between other packages
composition	contains all classes that are used in composite models
repository	contains all classes that are used in the repository model
resourceenvironment	contains the classes to specify the resource environment
system	contains the System class that represents the system model
allocation	contains classes that are necessary for an allocation model
usagemodel	contains classes that are necessary to create a usage model
subsystem	contains a class to compose sub-systems of a system
resourcetype	contains classes to specify resource types
protocol	contains an abstract class as a stub, such that interfaces could be extended by protocols
parameter	contains the specification for variable usage
reliability	contains classes for reliability prediction
seff	contains all classes that are necessary to model a SEFF
seff_performance	sub-package of SEFF that contains performance specific classes
seff_reliability	sub-package of SEFF that contains reliability specific classes
qosannotations	enables the annotation of quality information
qos_reliability	sub-package of qosannotations that contains classes to that are necessary for reliability annotations
qos_performance	sub-package of qosannotations that contains classes to that are necessary for performance annotations

Table 1: Current packages of the Palladio Component meta-model

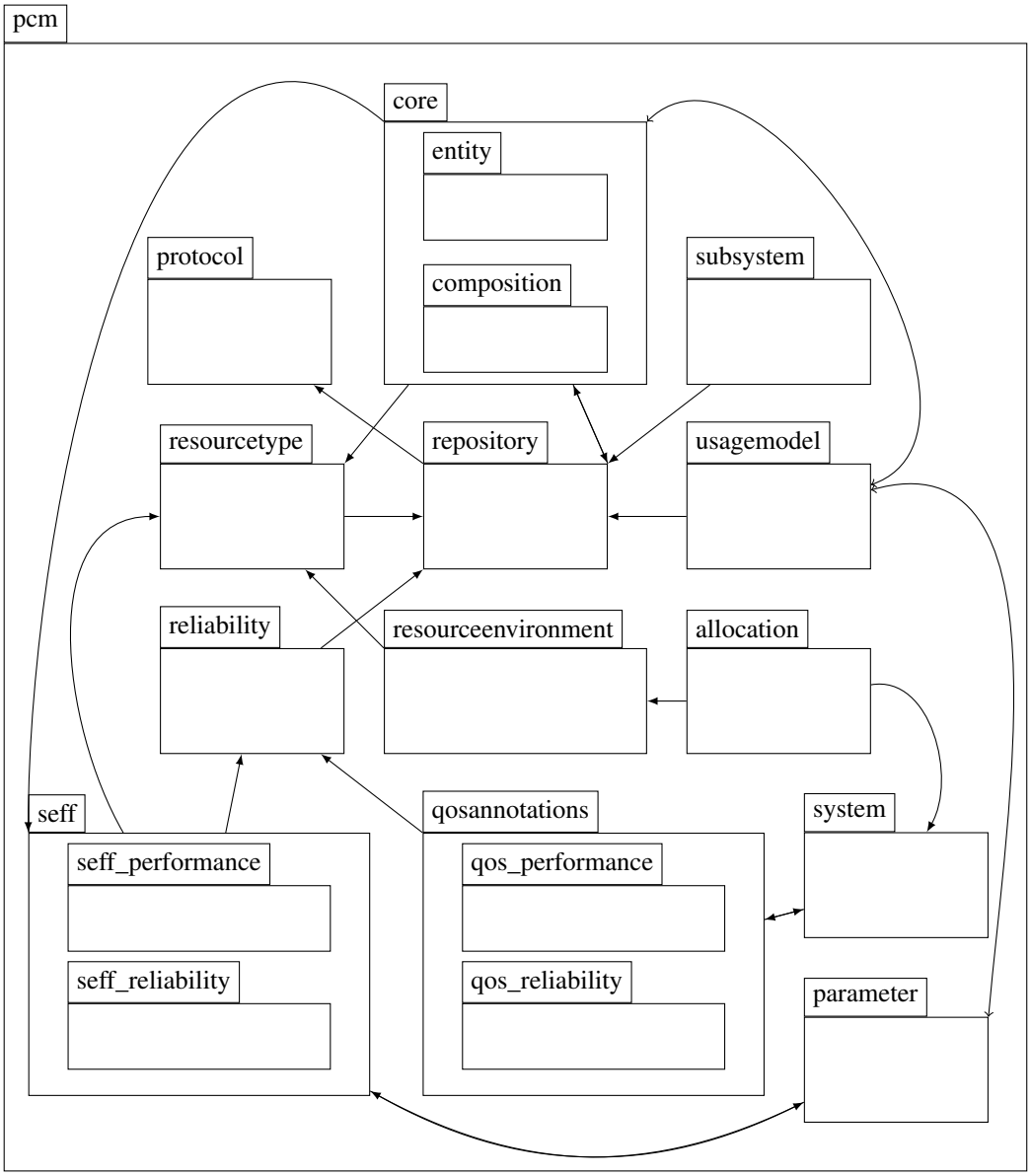


Figure 2: Current package structure of the Palladio Component meta-model. An arrow between two packages implies a dependency from one package to the other package. In order to simplify the Figure we assume that the arrows are transitive. For example, qosannotations depends on reliability and repository. However, since reliability already depends on repository we omit the arrow from qosannotations to repository. We also count dependencies from and to sub-packages as dependencies from and to the parent package. Finally, we omited dependencies to the entity package.

4 Identification Procedure

The content of this paper originates from a thorough, longer running inspection of the meta-model. Beforehand, we outlined the concerns and their dependencies from an outside perspective (similar to the view of an experienced Palladio user). It was the goal of the inspection to confirm, refine and correct this top-down classification with technical bottom-up information. We examined, how the package structure, relates to the concerns and arranged all classes to their belonging concerns. We also sought for new concerns.Because besides the top-down visible concerns, the invisible infrastructure of the meta-model (abstract class hierarchy) and hidden features can only be seen when looking into the meta-model. We considered how the concerns could be internally structured (e.g. by subpackages or subconcerns) and which concerns should rather be fused with others. Further, we inspected the dependencies of every class and if they confirmed to our idea of dependencies between concerns.

5 Concerns of the Palladio Meta-Model

Within this section, we will describe the concerns, which we identified in the Palladio meta-model. We will explain, what is the purpose of the concern and what are the main concepts. How do they originate within the package structure and why they were assembled like this. Outgoing dependencies are shortly explained. Incoming dependencies which currently exist in the meta-model and are problematic are discussed. Such dependencies could be of the type which go from a core concern to an enhancing concern. Please note that this is in no way an explanation of the meta-model in detail. Not every class and individual dependency is explicitly explained, but only the ones which are important for the concerns from an outside perspective. For additional details, please consult the tech report [RBB+11].

An overview of the core concerns is shown in Figure 3. Dependencies to the entity concern are not shown for simplicity, as almost every concern is dependent on it.

5.1 Entity

The entity concern contains a simple class hierarchy to provide IDs and names to subclasses. The content of this concern stems mostly from the entity package. The interface dependent part has been factored out. The exception is the identifier class, which carries an ID and makes all subclasses uniquely identifiable. It is contained within its own minimal meta-model. It could be beneficial to locate this whole concern into that meta-model, as names are also used in further meta-models.

This concern has no outgoing dependencies. However, most classes are dependent on this concern, as IDs and names are widely used. So, we will not explicitly state these dependencies everywhere.

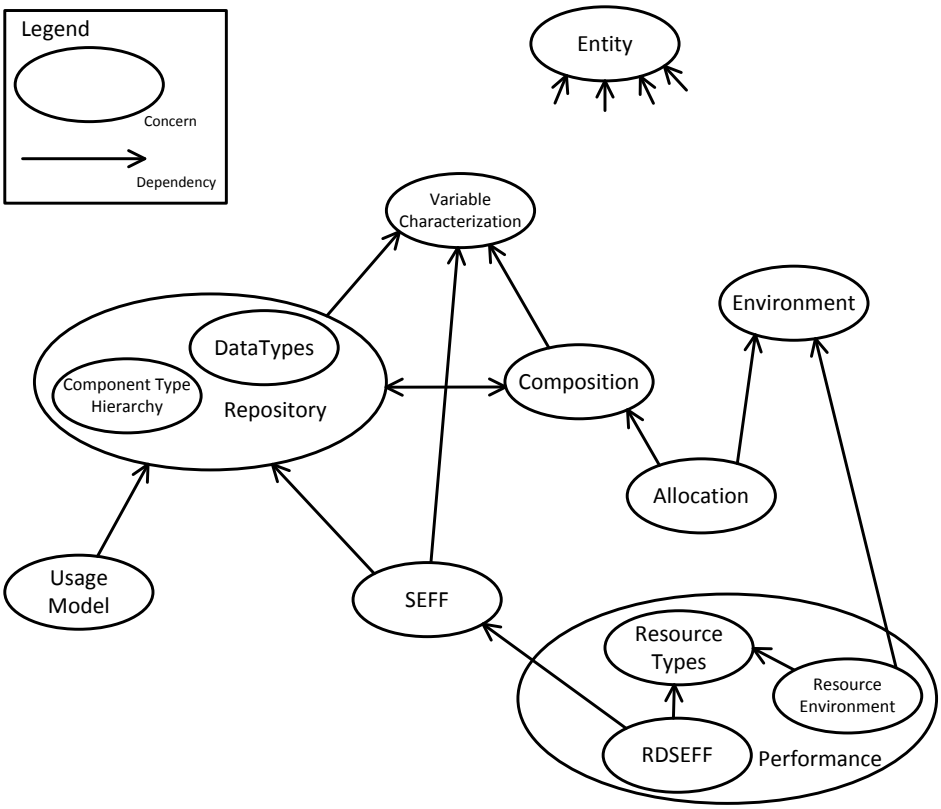


Figure 3: Main Concerns

5.2 Variable Characterization

The variable characterization concern provides capabilities to specify properties of variables (e.g. bytesize, structure, value), These variables are mostly input and output parameters of calls as well as in component parameter specification. This concern is currently known as the parameter package. However, as the parameter is a concept and meta-class from the repository, we chose variable characterization as a name. As the concepts, which are contained in the concern, are used in different places, we did not merge it, but left it as an independent concern.

The main concepts of this concern are: The VariableUsage holds the name of a variable and an arbitrary number of characterizations. The VariableCharacterisation holds type and value of the characterization. The VariableCharacterisationType declares which characterization should be specified: bytesize, structure, value and so on.

There are no outgoing dependencies, but to the StoEx meta-model, as random variables are used.

5.3 Repository

The repository concern covers the main concepts, which users know from the conventional repository model within the PCM. Within a repository model, interfaces, components and their relations can be specified. The repository may also contain custom data types, which are meta-modeled in a subconcern. Another subconcern provides modeling constructs for the component type hierarchy (Provides- and CompletesTypes). Further, the content of the repository concern could be subdivided into an abstract part which contains the abstract class hierarchy and a concrete part, which assigns semantics to the abstract classes. Such a further division could foster reuse in other domains and constrain the direction of dependencies. The classes of the repository constraints originate from the repository, subsystem and entity packages. However, some specific content of the repository package was factored out into other concerns.

From the outside view, it is clear what the main concepts of the repository concern are: components, interfaces and their relations. However, from the inside view, there is an elaborate class hierarchy for classifying different types of components. Further main concepts are: roles and a specialization of the abstract type hierarchy to use operation list signature.

The repository concern is dependent on the composition concern, as it describes how composed elements of the repository are constructed. The variable characterization is also referenced to enable component parameters. There are some problematic outgoing references, which should be reversed. Namely to the reliability and QoS annotations concerns. To form a clean architecture description language, also the dependency to the behavior of components (SEFF) should be reversed, so that the repository is not dependent on behavior but behavior extends or annotates content of the repository concern.

DataTypes This subconcern of the repository concern provides modeling capabilities for custom data types. These data types can then be used in definitions of signature lists for OperationInterfaces. It originates completely from the repository package. Its main concepts are: the PrimitiveDataType (such as integer, char, string), the CompositeDataType (similar to struct concept of object oriented programming) and the CollectionDataType (a set containing multiple instances of the same data type). There are no outgoing dependencies.

Component Type Hierarchy This subconcern of the repository concern contains the classes to specify the component type hierarchy. The whole concern was contained in the repository package. ProvidesComponentTypes are used to declare mandatory provided roles. CompletesComponentTypes are used to constrict required roles. These role constraints have to be fulfilled for a conforms relation to hold. These component types can then be substituted by their conforming components within an composed structure. There are no outgoing dependencies (except to the parent concern), nor problematic incoming ones.

5.4 Composition

The composition concerns defines how composed structures can be constructed: components are put into their context within a composed structure, which results in an assembly context. The roles of the assembly contexts are connected by assembly connectors. The roles of the outer structure are connected to the roles of assembly connectors by delegation connectors. This concern stems solely from the composition package and will therefor be known to Palladio users as the composed structure or system diagram. This concern is dependent on the repository concern, as components and their roles are referenced. There is also a dependency to variable characterization, as component parameters can be set in assembly contexts.

5.5 SEFF

The SEFF concern contains all classes that are necessary to model the behavior of a component method. For example, the behavior can be modeled in terms of internal actions, external actions, loops and branches. Unlike the current PCM meta-model, the SEFF does not allow the modeling of performance or reliability. This will be realized by additional extensions to the SEFF. Another difference to the current meta-model will be that a SEFF is no longer contained within BasicComponents. This will decouple the repository from the SEFF. In addition, this will have the positive effect that concurrent changes on SEFF and the Repository model will no longer overwrite the other change, if saved. Since SEFFs are contained in a BasicComponent the SEFF concern has a outgoing dependency to the repository.

5.6 Usage

The usage concern contains all classes that are necessary to create a usage model. It matches the current usagemodel package in the current Palladio meta-model. The usage only has a dependency to the Repository concern. In particular it needs the OperationSignature and the OperationProvidedRole that are used in the EntryLevelSystemCall.

5.7 Environment

The environment concern contains all classes that are necessary to model the environment of the system in terms of hardware environment and their linking between one another. The concern consists of the performance independent part of the resourceenvironment package from the current Palladio meta-model. Hence, it will contain the classes Environment and Liniking. The environment concern has no outgoing dependencies.

5.8 Allocation

The Allocation concern contains all classes that are necessary to create an Allocation model. Like the usage model, it matches its former PCM package, the allocation. The main concepts are Allocation and AllocationContext. The AllocationContext matches which component is deployed on which hardware resource. Hence, the allocation concern has dependencies to the composition and the environment concerns.

5.9 Performance

The Performance concern enables the modeling of performance information. It has several sub-concerns. Each of the sub-concerns extend other concerns in order to enable the performance annotations that can be used for performance prediction in a later step.

RDSEFF The RDSEFF (Resource Demanding Service Effect Specification) contains all classes to enable the modeling of a SEFF that interacts with resources. It contains all classes from the seff_performance package in the current Palladio meta-model. The main concepts of the RDSEFF are the ResourceDemandingSEFF class itself as well as ResourceCall and ResourceDemandingBehavior. It has outgoing dependencies to the SEFF and Resource Environment and an incoming dependency from the Infrastructure concern.

Resource Environment The Resource environment is a sub-concern of the performance concern. It contains classes that extends the environment model to enable the performance modeling of hardware resources. These resources are used for the performance prediction

in a later step. The main concepts of the resource environment are the ResourceEnvironment, which extends environment, and the ResourceContainer. The resource environment has outgoing dependencies to the environment and to the Resource Type.

Resource Type repository The resource type concern is a sub-concern of the performance model. The resource type contains the concrete resources, e.g. CPU and HDDs that can be used in the resource environment. Since the resource type repository only contains resources that are used in the resource environment it does not have any outgoing dependencies.

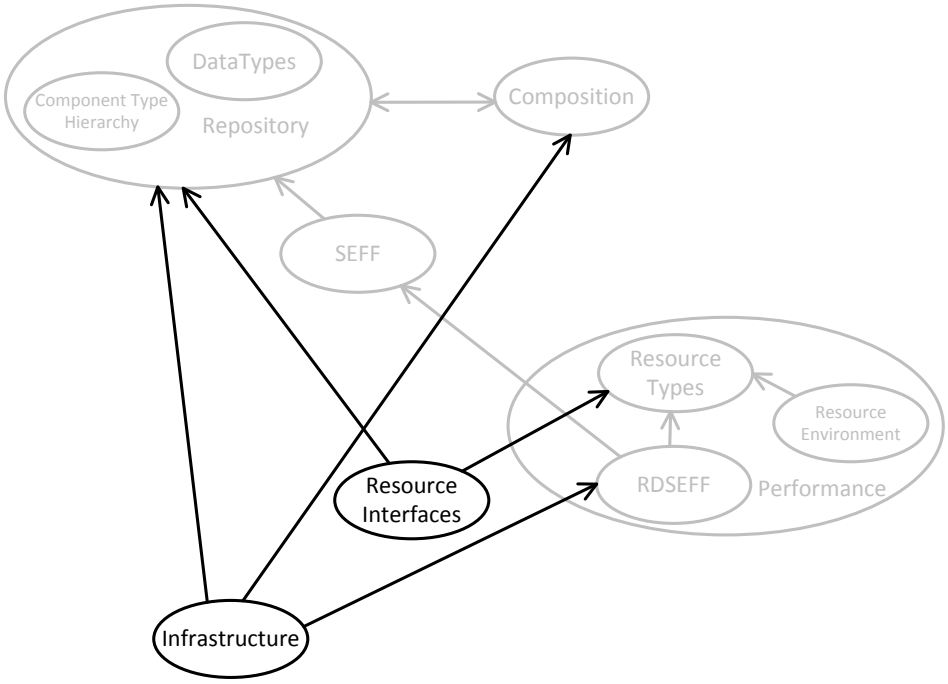


Figure 4: Performance Specific Extending Concerns

5.10 Resource Interface

The resource interface [Hau09] concern extends the resource environment and allows a more detailed performance predictions based on a resource model. The main concepts of the resource interface are ResourceInterface and ResourceSignature. The resource interface has an outgoing dependency to the resource environment. Since the class ResourceSignature uses Parameters from the repository it also has an outgoing dependency to the repository (see Figure 4).

5.11 Infrastructure

The infrastructure concern provides capabilities to model infrastructure software components (e.g. middleware) [Hau09]. Its dependencies are shown in Figure 4. Calls to such infrastructure components have no return values and can be made directly from InternalActions. The infrastructure components therefore provide capabilities to model complex resource demands. The classes infrastructure concern originate in the repository, composition and SEFF package. This subdivision should also be kept in this concern as subconcerns or packages.

The main concepts of this concern are: InfrastructureSignatures, -Interfaces, -Roles, -Connectors and -Calls. Normal and infrastructure Components are distinguished by an attribute in the ImplementsComponentType, this should be replaced by a construct of more annotating nature to be able to factor it out of the repository concern.

The InfrastructureCall has an incoming dependency from SEFF, as it is contained within AbstractInternalControlFlowActions, this should be reversed. Expected outgoing references are therefore: to the repository, because of the inheritance from Interface, Roles, Signature and the annotation of component. There is also a reference to composition, because of the inheritance of the connectors and another reference to SEFF because of the introduction of the InfrastructureCall.

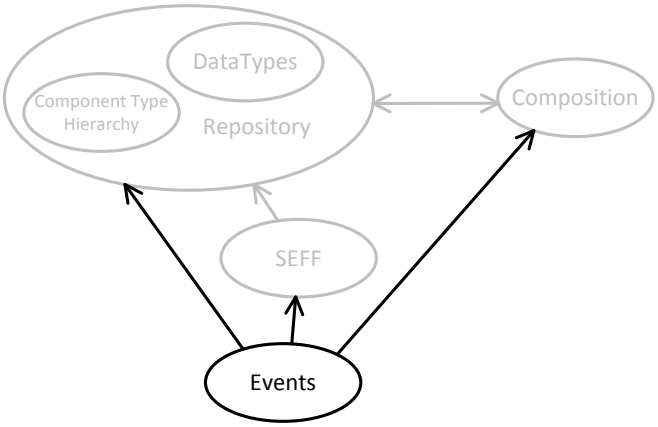


Figure 5: Event Communication Concern

5.12 Events

Currently the Event classes are part of the repository package [Rat13]. For the modularized meta model we propose a own concern for the Event extension of the PCM (see Figure 5. This concern itself will be separated in the three sub-concerns Events-Repository, Events-Composition and Events-SEFF. The Events-Repository contains all classes to model an

Event based system in the repository model, e.g. source and sink roles. The Events-Composition contains the classes that are necessary to compose the events modeled in the repository model. Finally, the Events-SEFF enable the modeling of event based Service Effect Specifications.

The Events-Repository sub-concern has dependency to the repository concern. The Events-Composition sub-concern has dependencies to the Events-Repository and to the composition. The Events-SEFF sub-concern has a dependencies to the Events-Repository and to the SEFF concern.

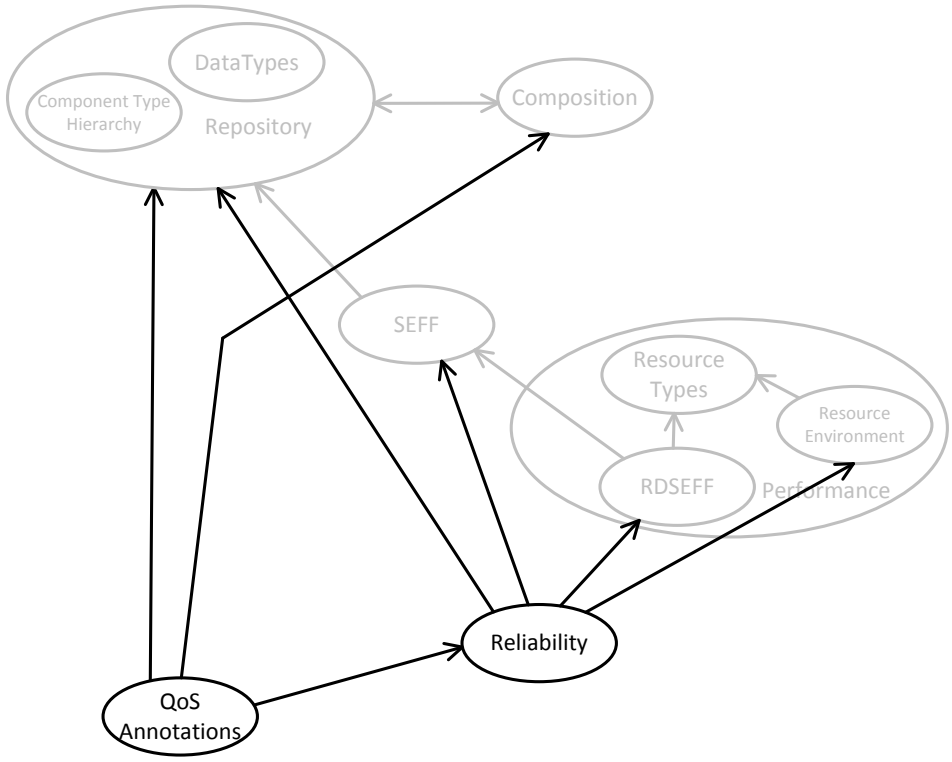


Figure 6: Reliability and QoS Annotations Concerns

5.13 Reliability

The reliability concern enriches the PCM by structure and information which is needed to model and analyze reliability of software systems [Bro12]. Its dependencies are shown in Figure 6. With this extension, InternalActions, hardware resources and external services are allowed to have a chance to fail. This concern stems from the reliability, seff_reliability and resource environment packages. Which should also be reflected in its subconcerns or

package structure.

The main concepts of the reliability concern are: The abstract class FailureType and its subclasses: SoftwareInduced-, ResourceTimeout-, HardwareInduced- and NetworkInducedFailureType. The FailureOccurenceDescription class contains a feature probability and a reference to a FailureType. FailureOccurenceDescriptions can be annotated to system external required interfaces or be contained in internal actions within SEFFs. The RecoveryAction represents a recovery point within a SEFF. A RecoveryAction contains multiple RecoveryActionBehaviors, which are RDSEFFs. The ResourceFailureProbabilityAnnotation is a new class which extracts the hard-coded failure rate from resource containers and LinkingResources.

This concern was retroactively and intrusively added to the meta-model. It is cleanly contained within its own packages. However, it has many incoming dependencies, from packages which are more general. These dependencies should be reversed to decouple these packages and their concerns. The reliability concern will then be an annotating extension. These problematic incoming dependencies are:

- A repository contains FailureTypes. These should be contained within an own container in the reliability concern.
- Signature references failure types.
- The ResourceType concern is dependent, as there are bidirectional relations to HardwareInducedFailureType and NetworkInducedFailureType.
- The SEFF concern is dependent on InternalFailureOccurenceDescription, as they are referenced by internal actions.
- Passive resources reference ResourceTimeoutFailureType (which is not consistent with the type level).

All in all, the reliability concern will be dependent on the following concerns: repository, ResourceType, SEFF and RDSEFF. However, it should be reconsidered, if the extension of the resource type concern is really necessary. The resource types are statically instantiated and do not change during modeling. Can a reliability engineer really model something using these relation, which changes the result of the reliability analysis? If the information provided by these relations is purely conceptual, they could be omitted.

5.14 QoS Annotations

This concern it used to apply Quality of Service (QoS) annotations to system external services or components which are not yet fully specified (e.g. Provides- or CompletesTypes). Its dependencies are shown in Figure 6. These annotations are not a general specification but context dependent. I.e. the resource environment is fixed as well as further dependencies and background or concurrent usage overhead. This concern contains two small

subconcerns for performance and reliability specific annotations. In the scope of the refactoring they may become subpackages of this concern or subpackages of the reliability and performance concern.

The main concepts are: The QosAnnotations class is just a container, which contains all annotation specifications. SpecifiedQoSAnnotation is an abstract super class for all QoS annotations. It references the role and signature, for which an annotation should be applied. SpecifiedOutputParameterAbstraction provides a variable characterization for a return value. Within the subconcerns, SpecifiedQoSAnnotation is specialized to provide annotations of execution time and failure probability and failure type.

Systems contain QoSAnnotations, this should be removed. A QoSAnnotation should reference a repository. The remainder are outgoing references which are legitimate due to their annotating nature. These dependencies reference the following concerns: repository, composition, variable characterization. Further, the reliability subconcern depends on the reliability concern, as the FailureOccurenceDescriptor is used.

6 Conclusion

During the process of inspecting the meta-model, it became clear to us, that containment of multiple concerns in one package and the distribution of concerns over multiple packages, degrade the understandability of the meta-model. The initial meta-model might have been conceptually well formed. However, later modifications and extensions worsened that design.

Another problem, which stems partly from later modifications and extensions, is that dependencies cannot be properly constrained in the package structure. This makes the meta-model more difficult to maintain, as one has to regard unexpected dependencies. E.g. when extending concerns are hard coded into core concepts, changing the extension has to be done carefully. Also it makes a modularization quite difficult.

Further, the lacking modularization makes the application domain of the meta-model very specific. As the PCM was initially designed for performance modeling and analysis, the quality constructs were hard coded into the foundational constructs. If the current Palladio meta-model is used or extended for other purposes and domains, domain unrelated features would decrease the comprehensibly for the developers.

To solve these problems, we need to modularize and refactor the meta-model and we need to set up conventions for future extensions. In the scope of the modularization, hard-coded parts will be factored out into modules. We hope, that the concerns, which we identified, and their dependencies will provide a good basis for future deliberations. The modularization will also bring some more control over the dependencies, as creating a reference between two meta-models, which are not yet dependent in that direction, requires explicit loading of the referenced meta-model. So, extra consideration will be provoked, when inserting such a new dependency.

Conventions will help to ensure, that most extensions will be made externally and not

intrusively. These conventions should also propose and restrict the extension method (e.g. aspect-oriented annotating [JHS+14], profiles [KDH+12]). It may also be beneficial for the understandability of the extensions, that their package structure reflects which concerns of the Palladio meta-model are extended.

7 Continuation of the Migration Process

The immediate next steps will be to discuss and settle for a modularization of the meta-model. We hope that the concerns identified in this paper will provide a solid base for future deliberations. Next, the meta-model will be refactored.

However, to use the modularized meta-model in future releases of the Palladio Bench, further steps have to be performed: A backwards transformation has to be implement, which transforms model instances of the modular meta-model into instances of the classic meta-model. This will enable all the tools to remain functional. The tools which are actively maintained can then be ported to operate on the modular in their own time. Another important step towards release is the implementation of new graphical editors.

The implementation of a forward transformation which transforms instances of the classic meta-model to the modular one, is not immediately mandatory for a release. However, providing such a transformation is nevertheless important, as it migrates legacy model instances.

8 Acknowledgments

This work was partially supported by the Helmholtz Association of German Research Centers.

References

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[Bro12] Franz Brosch. *Integrated Software Architecture-Based Reliability Prediction for IT Systems*. PhD thesis, Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie, Karlsruhe, Germany, June 2012.

[Hau09] Michael Hauck. Extending Performance-Oriented Resource Modelling in the Palladio Component Model. Diploma thesis, University of Karlsruhe (TH), Germany, February 2009.

[JHS+14] Reiner Jung, Robert Heinrich, Eric Schmieders, Misha Strittmatter, and Wilhelm Hasselbring. A Method for Aspect-oriented Meta-Model Evolution. In *Proceedings of the*

2Nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, VAO '14, pages 19:19–19:22, New York, NY, USA, July 2014. ACM.

[KDH+12] Max E. Kramer, Zoya Durdik, Michael Hauck, Jörg Henss, Martin Küster, Philipp Merkle, and Andreas Rentschler. Extending the Palladio Component Model using Profiles and Stereotypes. In Steffen Becker, Jens Happe, Anne Koziolek, and Ralf Reussner, editors, *Palladio Days 2012 Proceedings (appeared as technical report)*, Karlsruhe Reports in Informatics ; 2012,21, pages 7–15, Karlsruhe, 2012. KIT, Faculty of Informatics.

[Koz08] Heiko Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, 2008.

[Rat13] Christoph Rathfelder. *Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation*, volume 10 of *The Karlsruhe Series on Software Design and Quality*. KIT Scientific Publishing, Karlsruhe, Germany, 2013.

[RBB+11] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical report, KIT, Fakultät für Informatik, Karlsruhe, 2011.

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Eclipse series. Addison-Wesley Longman, Amsterdam, second revised edition, December 2008.

[SMRL13] Misha Strittmatter, Philipp Merkle, Andreas Rentschler, and Michael Langhammer. Towards a Modular Palladio Component Model. In Steffen Becker, Wilhelm Hasselbring, André van Hoorn, and Ralf Reussner, editors, *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days, KPDAYS '13*, volume 1083, pages 49–58. CEUR Workshop Proceedings, November 2013.

Predicting Energy Consumption by Extending the Palladio Component Model

Felix Willnecker[1], Andreas Brunnert[1], Helmut Krcmar[2]

[1]fortiss GmbH, Guerickstr. 25, 80805 München, Germany
{willnecker, brunnert}@fortiss.org
[2]Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
krcmar@in.tum.de

Abstract: The rising energy demand in data centers and the limited battery lifetime of mobile devices introduces new challenges for the software engineering community. Addressing these challenges requires ways to measure and predict the energy consumption of software systems. Energy consumption is influenced by the resource demands of a software system, the hardware on which it is running, and its workload. Trade-off decisions between performance and energy can occur. To support these decisions, we propose an extension of the meta-model of the Palladio Component Model (PCM) that allows for energy consumption predictions. Energy consumption is defined as power demand integrated over time. The PCM meta-model is thus extended with a power consumption model element in order to predict the power demand of a software system over time. This paper covers two evaluations for this meta-model extension: one for a Java-based enterprise application (SPECjEnterprise2010) and another one for a mobile application (Runtastic). Predictions using an extended PCM meta-model for two SPECjEnterprise2010 deployments match energy consumption measurements with an error below 13 %. Energy consumption predictions for a mobile application match corresponding measurements on the Android operating system with an error of below 17.2 %.

1 Introduction

Energy efficiency of software systems becomes a growing software engineering challenge [BVD+14]. Energy consumption of Information and Communication Technology (ICT) is rising due to higher demand in data centers, networks, and consumer devices like mobile devices [SNP+09, WBK14]. Therefore, there is need to investigate the energy saving potential of software systems [JGJ+12]. Today, hardware manufactures increase the capabilities of the hardware and simultaneously increase the energy efficiency. Operating system (OS) providers implement energy saving modes to increase the energy efficiency of the overall system. Though, neither optimization on hardware nor on OS level can compensate the rising demand, yet decrease the energy consumption of ICT systems [GJJW12]. Therefore, software optimizations in terms of energy are investigated more recently as the application software ultimately causes the energy demand on hardware and OS level [PLL14].

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

Performance of software systems and energy efficiency are often referred to as contradicting optimization goals [HSB12]. For example, the response time of an enterprise application can be decreased by duplicating the number of replicas of the system. By introducing additional replicas, the energy consumption of the corresponding system is directly increased. In contrast, enhancing the efficiency of a system in terms of performance metrics can also decrease the energy consumption of a system when a components resource demand decreases. Thus, performance metrics and energy consumption rely on the same underlying parameters of resource demand and hardware capabilities. This allows us to adapt and extend technologies used for performance evaluation to simulate the power demand of software systems and therefore predict the energy consumption of such systems.

Predicting performance metrics like response time, throughput or hardware utilization are core capabilities of the PCM modeling environment (Palladio-Bench) [BKR09]. This work proposes an extension of the PCM meta-model called power consumption model and corresponding extensions of the Palladio-Bench to take this meta-model extension into account. The power consumption model describes the power demand of hardware servers that are simulated. Furthermore, we extend the Palladio-Bench to calculate the software's energy consumption based on the resource utilization of the components and the power consumption model [BKR09].

This paper starts by introducing the proposed extension of the PCM meta-model including extensions of the Palladio-Bench for the generation of an energy report. We evaluate this extension with an enterprise application to demonstrate that this extension is accurate for data centers and with a mobile application to show the applicability for mobile devices. This work closes with an outline of related approaches, a summary and future work.

2 Meta-model Extension

The current state of the PCM meta-model cannot predict the energy consumption of a software system. To determine the energy consumption, we specify the power consumption of the hardware resources. A resource's electrical power consumption depends on its usage, caused by the resource demand of the software. Hence, knowing the power consumption P of a certain utilization level of a resource allows to calculate the energy consumption E over the considered time T as presented in equation 1.

$$E = \int_0^T P(t)\,\mathrm{d}t \tag{1}$$

The correlation between resource utilization and energy demand can be approximated for server systems with a linear model [FWB07, RSR+07]. Simple models, in which only the CPU utilization is considered for the power consumption calculation can predict it with high accuracy [RSR+07]. We leverage this correlation and extend the PCM meta-model with a power consumption model to represent such simple models as introduced in our previous work [BWK14]. Power consumption models define the power consumption of a

server as a linear equation using utilization values of the server's resources as independent variables [BWK14]. This linear equation calculates the power consumption P_{pred} for a server based on the constant power consumption of a system in idle state C_0 and the sum of the power consumption values of all other resources [BWK14]. The power consumption of a resource is calculated by multiplying a consumption factor C_i with its utilization as presented in equation 2 [BWK14].

$$P_{pred} = C_0 + \sum_{1 \leq i \leq n} C_i * u_i \tag{2}$$

The power consumption calculation based on the utilization of a resource needs to be modified for some resources used in mobile applications. Power consumption of a screen depends on the brightness or color intensity. The power consumption of sensors for the Global Positioning System (GPS) relies on the demanded time of the resource. A constant factor is not sufficient to calculate the power consumption of such sensors [WBK14]. We developed a generalized equation 3 to take non-constant values into account. Therefore, the utilization factor u_i can not only represent the utilization of traditional resources but also represent brightness of a display or other sensor specific behavior such as the accuracy of a GPS sensor. This utilization factor is multiplied by a function that represents the power consumption of a resource (P_{idlei}). Depending on the attached resource, the function can either be a linear or a probabilistic distribution function. Furthermore, we consider the idle state of a device not as constant, but also as a function (P_{idleo}). This accommodates the fact, that background actives of an Operating System (OS) result in varying power consumption. The generalized equation is formulated as follows:

$$P_{pred} = P_{idle0} + \sum_{1 \leq i \leq n} P_{idlei} * u_i \tag{3}$$

In PCM, servers and mobile devices are represented as resource containers. The new power consumption model element is added to the existing resource container meta-model element. This element contains the power consumption characteristics of a server, a mobile device or a network adapter and represents the power consumption as outlined in equation 3.

The same formula can be applied to the power consumption of a network adapter. For mobile devices, one of the largest power consumer is the cellular network adapter [WBK14]. The utilization factor u_i depends on the throughput of the adapter. A typical mobile device is linked to three different network adapters: Wi-Fi, Cellular and Bluetooth each with an independent factor function and an offset function for the power consumption of the adapter in idle state. Therefore, each linking resource is attached to a power consumption model representing the factor function that is multiplied with the throughput of the adapter and offset function for the adapter in idle state for the resource.

To model the power consumption of network adapters and processing resources we extend the PCM resource meta-model. Figure 1 presents our extensions for the PCM meta-model. The extension contains a *Power Consumption Model* and two types of *Power Consumption*

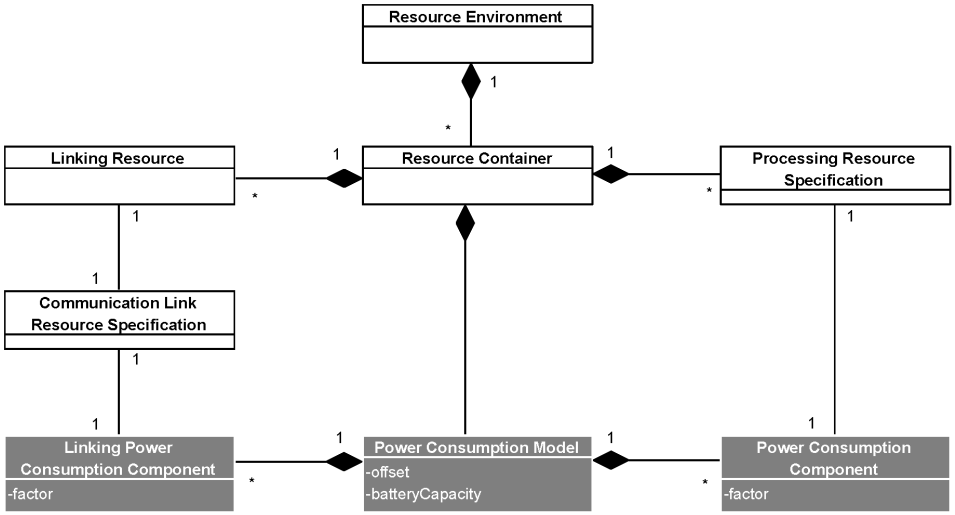


Figure 1: Meta-model Extension for predicting Energy Consumption

Components. The central power consumption model contains the offset function representing the power consumption of the device or linking resource. Furthermore, this model element contains the device's battery capacity in milliWatt-hours (mWh) to calculate the energy consumption of battery depended devices like mobiles and the discharging of its battery. The result of equation 1 can be subtracted from the battery capacity. This calculation provides the remaining battery capacity respectively the loading state.

The power consumption model can reference N *Power Consumption Components* and N *Linking Power Consumption Components*, one for each processing resource respectively one for each linking resource. Each of these components consists of a power consumption factor multiplied with the utilization factor (depending on e.g, utilization, brightness, throughput, demanded time) during simulation. The factor can be constant or a probabilistic function to simulate varying power consumptions.

Figure 2 shows an example of such a power consumption model element for one server with 16 Core Processing Units (CPUs) and one Hard Disc Drive (HDD) [BWK14]. We added a power consumption element to the resource container containing a constant (C_0) representing the idle power consumption of the server [BWK14]. To represent the power consumption of the resources CPU and HDD we added two power consumption component elements [BWK14]. Each of these processing resources is described with the variables of equation 2 [BWK14]. A power consumption component contains a factor (C_i) and a reference to the utilization (u_i) of the resource [BWK14]. Figure 2, shows a power consumption model with a constant C_0 of 200 Watts (W) for the server in idle state, a factor C_{CPU} of 300 W for the CPU and a factor C_{HDD} of 50 W for the HDD [BWK14].

In order to model elements like the GPS sensor or display usage in the resource demanding service effect specifications (RDSEFFs) additional resource types can be added to the

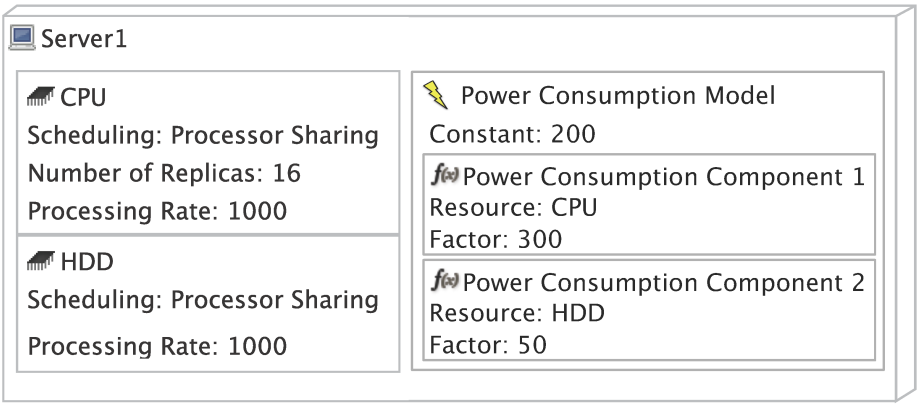


Figure 2: Power consumption model [BWK14]

resource repository of the PCM meta-model. Two key resources that we found necessary for this are purpose are DISPLAY and GPS. The resource demand placed on both resources during a simulation is used for the power consumption calculations depending on a discrete utilization of either 100 % (on) or 0 % (off).

Calculation of the energy consumption is conducted after the simulation run. We extend the Palladio-Bench with an energy consumption report. This report is based on the utilization of all resource. After a simulation run, we calculate the power demand for each resource. We build a sum over all resource-specific power demands of a single resource container and add the power demand of the resource container itself. The total power demand is integrated over the simulation time as described in equation 1. The result of this integral is the energy consumption for a single resource container in a PCM resource environment model. For constant power demands this can be simplified by just multiplying the power demand by the simulation time. The result is a report for the energy consumption of each resource container containing its total power consumption, its total simulation time, its energy consumption and a simple cost calculation.

3 Evaluation

Two experiments are performed to validate the extensions proposed in this work. We evaluate the accuracy for server systems by running a SPECjEnterprise2010[1] benchmark. For mobile devices we use the sports app Runtastic[2] and simulate its energy consumption.

[1]SPECjEnterprise is a trademark of the Standard Performance Evaluation Corp. (SPEC). The official web site for SPECjEnterprise2010 is located at http://www.spec.org/jEnterprise2010.

[2]Runtastic is a trademark of the runtastic GmbH. The official website is located at https://www.runtastic.com/

Table 1: Evaluation environment [BWK14]

Component	AMD-Based Server	Intel-Based Server
Base System	IBM System X3755M3	IBM System X3550M3
CPU	4 x AMD Opteron 6172	2 x Intel Xeon E5645
CPU Cores	2 x 2.1 GHz	6 x 2.4 GHz
Random Access Memory	256 GB	96 GB
OS	openSuse 12.2	openSuse 12.3
Application Server	6 x JBoss Application Server 7.1.1	
Application	SPECjEnterprise2010	
Database	Apache Derby DB version 10.9.1.0	
Java Virtual Machine	64 bit Java OpenJDK version 1.7.0	

3.1 SPECjEnterprise2010

This section is based on our previous work "Using Architecture-Level Performance Models as Resource Profiles" [BWK14]. Two power consumption models based on the presented PCM meta-model extension were generated for the SPECjEnterprise2010 benchmark application on an AMD-based server and an Intel-based server [BWK14]. To construct these power consumption models we used an application that conducted a calibration run on the target hardware as proposed by Economou et al. [ERKR06]. The run charged the resources independently from each other with varying intensity. While the resources were stressed, resource utilization and power consumption values were collected simultaneously. To collect the power consumption of the server systems, we used the Intelligent Platform Management Interface (IPMI[3]).

After the calibration run, we executed different workloads using varying amounts of users and conducted a simulation using the power consumption model for both hardware environments. We measured and compared the energy consumption and calculated the error of the simulated energy consumption. The used environment for this evaluation is described in table 1 [BWK14].

The power consumption of both servers was predicted with an error below 13 %. Table 2 shows the results for the AMD-based server. The load test ran with 1300 - 3500 clients (C) and caused between 367.55 W and 436.47 W Measured Mean Power Consumption (MMPC). The Simulated Mean Power Consumption (SMPC) lied within 320.26 W and 390.95 W resulting in a Power Consumption Prediction Error (PCPE) between 10.43 % and 12.87 %.

Table 3 shows the results for the Intel-based server. Between 1300 and 4300 clients were used by the load test and caused between 197.05 W and 264.29 W MMPC. The SMPC lied within 175.94 W and 232.69 W resulting in a PCPE between 10.71 % and 11.96 %. The power consumption was relatively stable during the steady state of all load levels, therefore the energy consumption was simply calculated by multiplying the mean power consumption values by the simulation time [BWK14].

[3] http://www.intel.com/design/servers/ipmi/

Table 2: Measured and simulated results for the AMD-based server [BWK14]

C	MMPC	SMPC	PCPE
1300	367.55 W	320.26 W	12.87 %
2300	403.87 W	352.22 W	12.79 %
3300	433.76 W	384.52 W	11.35 %
3500	436.47 W	390.95 W	10.43 %

Table 3: Measured and simulated results for the Intel-based server [BWK14]

C	MMPC	SMPC	PCPE
1300	197.05 W	175.94 W	10.71 %
2300	220.47 W	194.93 W	11.58 %
3300	241.67 W	213.91 W	11.49 %
4300	264.29 W	232.69 W	11.96 %

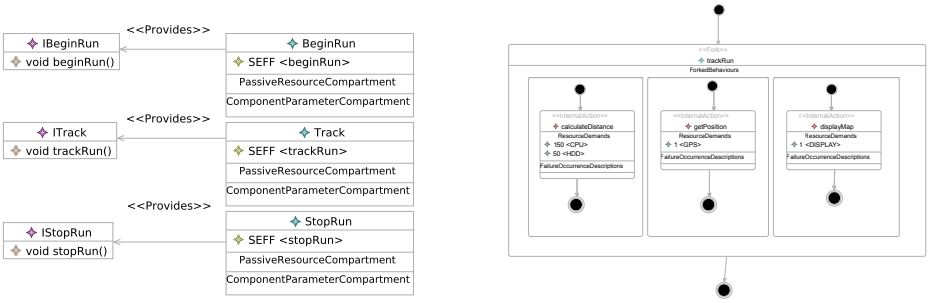
3.2 Runtastic for Android

A power consumption model for two devices running the Android OS is generated and used for the evaluation of power consumption models for mobile devices. Power consumption models for mobile devices can either use vendor profiles[4] provided for the Android OS or stress the resources independently and measure the discharging current. The discharging current C multiplied with the battery voltage V results in the power demand of a hardware resource P as presented in equation 4 [Lei14].

$$P = V * C \tag{4}$$

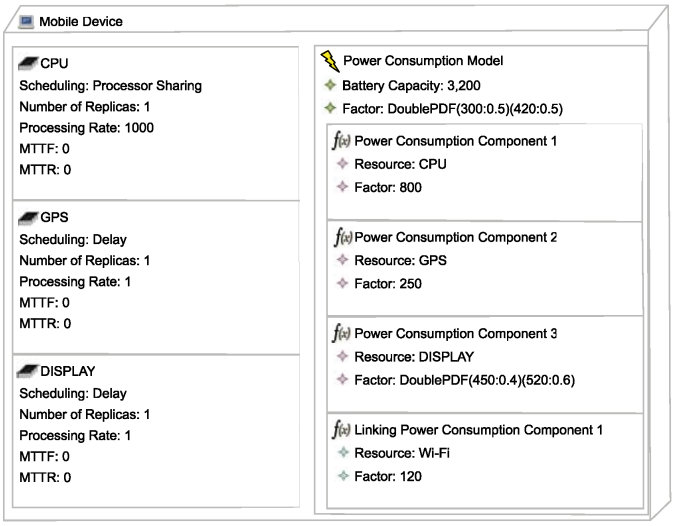
We use a calibration app to stress the resources of the mobile devices as the vendor profiles accuracy and completeness varied between the devices. The power consumption model is created after the calibration by running a regression on the measured data. An example of such a model with a CPU, display and a GPS sensor for a mobile device is presented in figure 3(c). To calculate the power demand of this device, the utilization of CPU, GPS and display is considered. To calculate the power demand of the CPU its utilization is multiplied with a factor of 800 milliWatts (mW). As soon as the GPS sensor is used, a utilization of 100 % is assumed. The GPS sensor in figure 3(c) therefore consumes 250 mW as soon as it is used. Similarly to the GPS sensor, the display can either be used or not. To take different brightness or color intensity levels into account, the power demand is either 300 mW in 50 % of the cases or 420 mW in the other 50 %. To calculate the power consumption P_{pred} we use the equation: P_{pred} = ((300 * 0.5) + (420 * 0.5)) + 800 * u_{CPU} + 250 * u_{GPS} + ((450 * 0.4) + (520 * 0.6)) * $u_{DISPLAY}$. A CPU utilization of 60 % and a utilization of the GPS and Display of 100 % would thus lead to a predicted power consumption P of 1582 mW. According to equation 1 we build the integral for the predicted power consumption P with the time T of the simulation to determine the energy

[4] https://source.android.com/devices/tech/power.html



(a) Repository model

(b) RDSEFF



(c) Power Consumption Model

Figure 3: Power Consumption Example for Mobile

consumption E of the system [BWK14]. The battery capacity of the modeled devices is 3.200 mWh. A constant use of the display and GPS sensor while the CPU is active at 60 % utilization would therefore reduce the battery capacity by about 50 % per hour.

The example in figure 3(a) and 3(b) shows a representation of a simplified sports tracking application in PCM that tracks the position via GPS, displays a map and calculates the running distance. Display and GPS are accessible as processing resource. Display and GPS run concurrent to the calculations representing three different threads: A thread for the user interface, one thread for the location tracking and one thread for calculating the distance.

We used Runtastic for Android for this evaluation as this application uses a broad number of resource types. We predicted and measured the power consumption of a 30-minute-

Table 4: Measured and simulated results for Runtastic Android application [Lei14]

Device	MMPC	SMPC	PCPE	BLPE
Samsung Galaxy Tab	1.251 W	1.084 W	13.35 %	0.67 %
LG Google Nexus 5	0.883 W	0.732 W	17.12 %	1.01 %

long Runtastic run on a Samsung Galaxy Tab and a LG Google Nexus 5. The Nexus device runs Android 4.4.4 and the Galaxy Tab has Android 4.3 installed. The evaluation run was conducted with both devices simultaneously. Additionally, both devices were connected to the same network carrier in order to reduce power-relevant variables (e.g, different signal strengths). During the run we collected hardware utilizations data with the Qualcomm Trepn-Profiler[5]. The utilization data is used to build a simple PCM usage, system, repository and allocation model. The repository model was constructed similar to the one shown in figure 3(a) and 3(b) and thus, simply represents the distribution of resource demands placed by the Runtastic application on mobile device hardware. We chose this approach as we did not have access to the source code or debugging interface of the Runtastic application. The usage model contains only one user starting and tracking a run. The power consumption during the run was logged for comparing it with the simulation. We conducted two simulations for the two devices. Both simulations used the same PCM repository model only with different resource environment models and therefore different power consumption models. Afterwards, we calculated the error between the power consumption prediction and the power consumption measured during the run [Lei14].

Table 4 shows the results for the mobile devices. The application causes between 1.251 W and 0.883 W MMPC. The SMPC lies within 1.084 W and 0.732 W resulting in a PCPE between 13.35 % and 17.12 %. The accuracy quality decreases for lower power consumptions. For a 30-minutes run we predicted the battery level of the device with an Battery Level Prediction Error (BLPE) of 0.7 % to 1 % [Lei14].

4 Related Work

This chapter outlines related approaches that measure, compare or predict the energy consumption of software systems.

Capra et al. [CFFG10] compared the energy consumption of customer relationship management (CRM) and database management systems (DBMS). The energy efficiency significantly varied between the compared systems when processing the same workload. They reasoned that energy efficiency should be considered when buying or developing software. The extension for the PCM meta-model proposed in our work takes energy efficiency into account as a key quality metric of a software system.

Jwo et al. [JWH+11] proposed to calculate the energy consumption of an enterprise application by multiplying the time a transaction is processed by mean power consumption

[5] https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler

of the host. The consumption therefore relies on the workload but still depends on the deployment environment and can therefore not convey a general energy efficiency metric. The extension in our work proposes a similar concept but based on resource demand values instead of response time values.

Johann et al. [JDNK12] proposed energy efficiency measurement methods for software systems. They define energy efficiency as the ratio of "useful" work relative to the energy required to process it. The calculation is based on single methods or components and is evaluated during the development process. They conclude that this method supports the creation of energy efficient applications.

Hönig et al. [HEKSP11] suggest a model-based approach for energy-aware software development for mobile applications. The energy consumption is based on vendor profiles provided by Android device manufactures. The accuracy of these models varies between different manufactures and devices but provides a baseline for the energy consumption calculation. Vendor profiles can be used to create resource environment models including power consumption models for the extension presented in our work.

Josefiok et al. [JSW+13] compared power measurements on different Android devices and OS versions. They discovered that the measurements and the Application Programming Interfaces (API) for monitoring the power consumption differ between manufactures and OS versions. They propose an energy abstraction layer to handle the multitude of APIs and granularity levels. Such a generalized API would help to create resource environment models including power consumption models and therefore predict the energy consumption of mobile applications on a broader scale.

These different approaches show the growing importance of evaluating energy consumption of software systems. Better measurements can help developers to decrease the energy consumption of their systems and increase the overall efficiency. Comparisons help users to choose the software with the best energy efficiency, which can also lead to lower operations costs. Energy consumption prediction capabilities can help to estimate the energy consumption and subsequently efficiency of a system without owning the target environment.

5 Conclusion

This work proposed a PCM meta-model extension to predict the energy consumption of software systems. This extension has been validated for server and mobile systems, for different hardware environments and workloads. The results show that the power consumption of these systems can be predicted with an error below 17.2 %. The evaluation used the SPECjEnterprise2010 benchmark for server systems and the sports tracking application Runtastic for mobile devices. The extension allows to specify the power consumption of hardware resources relative to their utilization. Two additional processing resource types have been added to the meta-model to represent mobile applications.

Constructing and analyzing energy efficient software is becoming an important research area for the software engineering community. The extension proposed in this work allows

predicting energy consumption of an application and to predict battery life of a device running a mobile application. The model extensions help developers to understand the varying power demands of different devices and to optimize applications in order to save battery power and reduce profiling effort [WBK14]. PCM meta-model instances for mobile devices are created manually and with a limited number of resources. The calibration application for mobile devices used in this work can create the power consumption model automatically but lacks the means to automatically generate performance models. Such model generators are already available for enterprise applications [BVK13, BHK11]. A future challenge in this research area is to create accurate models for mobile applications automatically.

References

[BHK11] F. Brosig, N. Huber, and S. Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-based Systems. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 183–192, Nov 2011.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009.

[BVD+14] Andreas Brunnert, Christian Vögele, Alexandru Danciu, Matthias Pfaff, Manuel Mayer, and Helmut Krcmar. Performance Management Work. *Business & Information Systems Engineering*, 6(3):177–179, 2014.

[BVK13] Andreas Brunnert, Christian Vögele, and Helmut Krcmar. Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications. In *Computer Performance Engineering*, pages 74–88. Springer, 2013.

[BWK14] Andreas Brunnert, Kilian Wischer, and Helmut Krcmar. Using Architecture-level Performance Models As Resource Profiles for Enterprise Applications. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 53–62, New York, NY, USA, 2014. ACM.

[CFFG10] Eugenio Capra, Giulia Formenti, Chiara Francalanci, and Stefano Gallazzi. The Impact of MIS Software on IT Energy Consumption. In *ECIS 2010 Proceedings*, 2010.

[ERKR06] Dimitris Economou, Suzanne Rivoire, Christos Kozyrakis, and Partha Ranganathan. Full-system Power Analysis and Modeling for Server Environments. In *Workshop on Modeling, Benchmarking, and Simulation*, Boston, Massachusetts, USA, 2006.

[FWB07] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power Provisioning for a Warehouse-sized Computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM.

[GJJW12] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing Energy Code Smells with Reengineering Services. In *GI-Jahrestagung*, pages 441–455, 2012.

[HEKSP11] Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. SEEP: Exploiting Symbolic Execution for Energy-aware Programming. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11, pages 4:1–4:5, New York, NY, USA, 2011. ACM.

[HSB12] Hagen Höpfner, Maximilian Schirmer, and Christian Bunse. On Measuring Smartphones' Software Energy Requirements. In *ICSOFT*, pages 165–171, 2012.

[JDNK12] Timo Johann, Markus Dick, Stefan Naumann, and Eva Kern. How to Measure Energy-efficiency of Software: Metrics and Measurement Results. In *Proceedings of the International Workshop on Green and Sustainable Software*, pages 51–54, Zurich, Switzerland, 2012.

[JGJ+12] Jan Jelschen, Marion Gottschalk, Mirco Josefiok, Cosmin Pitu, and Andreas Winter. Towards Applying Reengineering Services to Energy-Efficient Applications. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 353–358, March 2012.

[JSW+13] Mirco Josefiok, Marcel Schröder, Andreas Winter, et al. An Energy Abstraction Layer for Mobile Computing Devices. In *2nd Workshop EASED@ BUIS 2013*, page 17, 2013.

[JWH+11] Jung-Sing Jwo, Jing-Yu Wang, Chun-Hao Huang, Shyh-Jon Two, and Hsu-Cheng Hsu. An Energy Consumption Model for Enterprise Applications. In *Proceedings of the IEEE/ACM International Conference on Green Computing and Communications*, pages 216–219, Washington, DC, USA, 2011. IEEE.

[Lei14] Johannes Leimhofer. Predicting the Energy Consumption of Mobile Applications using Simulations. Bachelor's thesis, Technische Universität München, October 2014.

[PLL14] Giuseppe Procaccianti, Patricia Lago, and Grace A Lewis. A Catalogue of Green Architectural Tactics for the Cloud. In *Proceedings of 8th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 29–36. IEEE, 2014.

[RSR+07] S. Rivoire, M.A. Shah, P. Ranganathan, C. Kozyrakis, and J. Meza. Models and Metrics to Enable Energy-Efficiency Optimizations. *Computer*, 40(12):39–48, 2007.

[SNP+09] Lutz Stobbe, Nils Nissen, Marina Proske, Andreas Middendorf, Barbara Schlomann, Michael Friedewald, Peter Georgieff, and Timo Leimbach. Abschätzung des Energiebedarfs der weiteren Entwicklung der Informationsgesellschaft. *Abschlussbericht an das Bundesministerium für Wirtschaft und Technolgogie. Berlin, Karlsruhe*, 12, 2009.

[WBK14] Felix Willnecker, Andreas Brunnert, and Helmut Krcmar. Model-based Energy Consumption Prediction for Mobile Applications. In Jorge Marx Gómez, Michael Sonnenschein, Ute Vogel, Andreas Winter, Barbara Rapp, and Nils Giesen, editors, *Proceedings of Workshop on Energy Aware Software Development (EASED) @ EnviroInfo 2014*, pages 747–752. BIS-Verlag, 2014. ISBN 978-3-8142-2317-9.