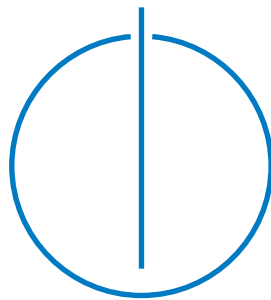


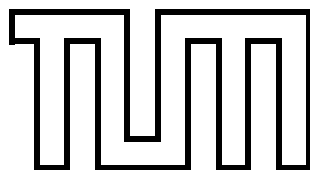
TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik  
Lehrstuhl für Wirtschaftsinformatik (I 17)  
Prof. Dr. Helmut Krcmar

**Memory-aware Multi-Objective  
Optimization of Deployment  
Topologies for Distributed  
Applications**

Felix Markus Willnecker





TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik  
Lehrstuhl für Wirtschaftsinformatik (I 17)  
Prof. Dr. Helmut Krcmar

**Memory-aware Multi-Objective  
Optimization of Deployment  
Topologies for Distributed  
Applications**

Felix Markus Willnecker

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. *Alexander Pretschner*  
Prüfer der Dissertation: 1. Prof. Dr. *Helmut Krcmar*  
2. Prof. Dr. *Ralf H. Reussner*  
Karlsruhe Institute of Technology (KIT)

Die Dissertation wurde am 13.10.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18.01.2018 angenommen.

## Acknowledgements

*„Es ist ein lobenswerter Brauch: Wer was Gutes bekommt, der bedankt sich auch“ - Wilhelm Busch (1832-1908)*

During the last four years of my research at fortiss I received a lot. A lot of good ideas, a lot of help, a lot of resources for my research. Now is the time to say thank you to the ones that contributed to this thesis, who supported and encouraged me.

First I want to thank Prof. Dr. Helmut Krcmar for the opportunity to work in his group and for supervising this interesting and challenging research topic. I enjoyed the freedom of research you granted me, as well as the guidance you offered. During this time I had some great experiences and got the chance to evolve not only as a scientist, but also personally.

My thanks also goes to my former and current colleagues Christian Vögele, Andreas Brunnert, Johannes Kroß, and Alexandru Danciu of the Performance Management Group at the fortiss institute for their valuable advices, ground work they created, suggestions and fruitful discussions around the topic of my dissertation. This work would have taken a lot longer and achieved less without this group. I also like to say thank you to the former FB 3 colleagues at fortiss.

Several colleagues of other institutions collaborated with me throughout the years and I am very grateful for their support. First of all, Dr. André van Hoorn from the University of Stuttgart, who worked with me in the SPEC DevOps Performance Working Group. Prof. Dr. Samuel Kounev, Dr. Simon Spinner and Jürgen Walter from the University of Würzburg, whose tools and techniques contributed significantly to the results of my work. Last but not least, Prof. Dr. Ralf Reussner and Prof. Dr. Steffen Becker, whose work on the Palladio Component Model lay the foundation of my research.

This work was supported by many dedicated and ambitious students. I would like to thank Johannes Leimhofer, Julia Kindelsberger, Bernhard Koch-Kemper, Carmen Carlan, Thomas Zwickl, and Sören Gunia for their excellent work, their ideas, and clever thoughts.

Finally, my biggest thank goes to my wonderful wife Eva. You constantly supported me during these years. Thank for enduring long working hours at night, at Christmas, during our vacations. Whenever a paper deadline called you were patiently waiting for me. Without you, I could not have done it.

## Abstract

**Problem** Distributed applications are composed of components and distributed throughout multiple container, Virtual Machines (VMs) or bare-metal server. These applications often utilize corresponding resources far below available capacity. Especially in managed environments, such as cloud Infrastructure as a Service (IaaS) environments, this induces unnecessary costs. Changing the deployment topology in order to increase resource utilization demands several tests on environments comparable to the production system. The effort to test a certain topology can be reduced by using performance models. Such models can be used to landscape system architectures and to simulate changes in the deployment topology or resource environment without utilizing actual resource environments. Therefore, this work aims on generating performance models for distributed applications to generate models of such applications and to optimize their deployment topologies. Furthermore, memory is typically disregarded performance meta-models leading to inaccurate performance models of distributed applications. Therefore, our approach adds a memory model to architecture-level performance models to optimize deployment topologies.

**Research Method** The research goals were achieved by following the design-science paradigm. During the work on this thesis, several artifacts were designed, implemented, and evaluated such as the deployment topology optimizer and an automatic memory management model and corresponding simulation methods. Our research is based on previous results in the area of application performance management and software performance engineering such as application monitoring solutions and performance model generators (PMGs). Our artifacts utilize and extend this previous research that was identified in multiple literature reviews. We used prototyping and simulation, as well as controlled experiments to implement, test, and evaluate our artifacts.

**Results** This work presents an memory-aware deployment topology optimizer for distributed applications. The key results are a scalable simulation cluster, an automatic memory management performance model, and a deployment topology optimization. The cluster is designed to be integrated it into other tool-chain. In this work, the cluster enables us to simulate multiple design variants such as different deployment topologies in parallel. With the use of the automatic memory management performance model, performance engineers and researchers are enabled to consider memory and garbage collection as an important factor of their applications and runtimes. Furthermore, our deployment topology optimizer can select good topologies by the use of evolutionary algorithms to conduct multi-objective optimization. The corresponding process presented in this work uses small test environments to generate holistic performance models and to predict the effects in target environments. In addition, we present a flexible cost-model that allows



---

to consider runtime costs in on-premise, cloud, and hybrid environments. The results of this cost-model are regarded in the topology optimization.

**Research Implications** This work combined two separated research fields: architecture optimization and performance model generation/extraction. This combinations allows researchers to use real world applications and connect those to academic research regarding architecture optimization. Furthermore, academic research benefits from the meta-model approaches presented in this work. Especially, the probabilistic memory model allows to research in memory intensive areas such as in-memory databases. With the contributions of this work, automatic, and dynamic memory management is integrated in architecture-level performance models. Additionally, large simulation-based experiments with a huge amount of variants become possible with the simulation service proposed in this work.

**Practical Implications** This work contributes to practice by providing a holistic tool that can derive performance models and optimize their deployment topology. Especially for cloud environments, this allows companies to save operation costs by providing similar quality of service. Unused resources are utilized by the application operator instead of the IaaS provider. The memory management model provides an easy to understand heuristic for profiling memory behavior of an application. Developers and architects can use this as metrics to understand, monitor, and improve the memory footprint of their distributed applications. The improvements of PMGs presented in this work lead to more accurate performance models. These models cover more aspects of the runtime environment and of the application itself. This comprises memory and garbage collection behavior and the use of resource demand measurements and estimations. The latter increases the supported technologies for PMG the number of potential monitoring solutions is higher when combining both approaches. Tools for capacity planning and in continuous delivery pipelines that use PMG benefit from these improvements.

**Limitations** The presented methods, models, and techniques have certain limitations. The PMG approach, requires a transaction ID that is unique through the complete system. If such an ID is not available and not injectable, our approach is blind at the boundaries of the system. In our case, this allows us to consider databases as blackboxes and derive their resource demands using estimation techniques. However, our model generation is blind if any further cascades occur, like a secondary database system, a replication, or a shard.

Even though, we use a combination of recombination and mutation for our deployment optimizer, it is still possible that not the best solution is found. The optimal solution might never be found as the search space can be huge. We try to conduct a broad search, by using recombinations in every generation. This might not lead to the best solution, but usually to a deployment topology that makes use of underutilized resources in contrast to naive deployments.

## Zusammenfassung

**Problem** Verteilte Anwendungen bestehen aus Komponenten die über verschiedenste Container, virtuelle Maschinen oder Hardware Server verteilt betrieben werden. Diese Applikationen nutzen die zur Verfügung stehenden Ressourcen üblicherweise nicht aus. Insbesondere in gemanagten Umgebungen, wie IaaS Cloud Umgebungen, führt dies zu unnötigen Kosten. Die Topologie anzupassen, um die zur Verfügung stehenden Ressourcen besser auszunutzen, erfordert eine Reihe von Tests auf produktionsnahen Systemen. Der Aufwand der dabei entsteht kann durch Performance Modelle reduziert werden. Solche Modelle können genutzt werden, um die Systemarchitektur und ihre Topologie abzubilden und Änderungen mit Hilfe von Simulationen zu bewerten. Diese Arbeit benutzt Performance Model Generierung für verteilte Unternehmensanwendungen zur Erstellung solcher Modelle, um anschließend die Topologie dieser Anwendungen zu optimieren. Darüber hinaus führen wir ein Model für die Simulation des Arbeitsspeichers ein, da dies in aktuellen Performance Modellen typischerweise vernachlässigt wird, dennoch großen Einfluss auf die Performance der Applikation hat.

**Forschungsmethode** Zur Erreichung der Forschungsziele wurde das Design-Science Paradigma verwendet. Während der Arbeit an dieser Dissertation sind verschiedene Artefakte designed, implementiert und evaluiert worden, wie beispielsweise ein Optimierer für die Topologie von verteilten Applikationen oder ein Model für die Simulation von automatischen Memory Management. Diese Arbeit fußt auf Arbeiten aus dem Bereich des Application Performance Management und Software Performance Engineering, wie beispielsweise Application Monitoring oder Performance Model Generatoren. Unsere Artefakte bauen auf diese vorherigen Arbeiten auf und erweitern diese durch zusätzliche Komponenten. Zur Identifikation relevanter Vorarbeiten wurden mehrere systematische Reviews der wissenschaftlichen Literatur durchgeführt. Zur Evaluierung unserer Artefakte kommen Methoden wie Prototyping, Simulation und Experimente zum Einsatz. Dadurch wurden unsere Artefakte in mehrere Iterationen getestet, auf Basis der Ergebnisse verbessert und erneut evaluiert.

**Ergebnisse** Diese Arbeit enthält einen Topologie Optimierer für verteilte Applikationen der die Arbeitsspeicher Ressource berücksichtigt. Die wichtigsten Ergebnisse und Artefakte sind ein skalierbarer Simulationscluster, ein Performance Meta-model für automatisches Memory Management, sowie der genannte Topologie Optimierer. Mit Hilfe des Simulationsclusters können unterschiedliche Modelinstanzen gleichzeitig verteilt simuliert werden. Der Cluster ist für die Integration in andere Tools mit Hilfe einer REST API designed. Wir nutzen den Cluster um eine große Anzahl von Topologien parallel zu simulieren. Durch die Einführung eines Models für automatisches Memory Management, kann die Arbeitsspeicher Ressource unter Berücksichtigung von Garbage Collection als wichtiger Faktor für die Performance von verteilten Anwendungen in Performance Modellen genutzt werden. Der Optimierer erlaubt die automatische Auswahl von guten

---

Topologien für verteilte Applikationen mit Hilfe von evolutionären Algorithmen und durch mehrdimensionale Optimierung. Der in dieser Arbeit vorgeschlagene Prozess sieht vor, dass ganzheitliche Performance Modelle in kleinen Testumgebungen erstellt werden und von da aus die Performance Metriken in der Zielumgebung simuliert und optimiert werden. Darüber hinaus führen wir ein flexibles Kostenmodell ein, welches die Laufzeitkosten in On-premise, Cloud und hybriden Umgebungen berücksichtigt und in die Optimierung mit einfließt.

**Beitrag zur Forschung** Diese Arbeit kombiniert zwei Forschungsfelder: Architekturoptimierung und Performance Model Generierung oder Extraktion. Die Kombination dieser beiden Felder erlaubt Forschern die Nutzung von Modellen echter verteilter Applikationen zur Evaluierung und Weiterentwicklung ihrer Forschungsergebnisse, insbesondere im Bereich der Architekturoptimierung. Weiterhin profitiert die akademische Forschung durch die eingeführten Erweiterungen für Performance Modelle hinsichtlich der Arbeitsspeicher Ressource. Das eingeführte Model basiert auf Wahrscheinlichkeit und ist dadurch flexible einsetzbar. Denkbar wäre beispielsweise der Einsatz zur Untersuchung von In-memory Datenbanken. Darüber hinaus, große Simulationsexperiment mit einer großen Anzahl und Varianten können mit Hilfe des eingeführten Simulationsclusters schneller, parallel durchgeführt werden.

**Beitrag zur Praxis** Durch die Einführung einer ganzheitlichen Toolkette zur Erstellung von Performance Modellen und Optimierung der Topologie von verteilten Anwendungen bietet eine Reihe von Vorteilen für die Praxis. Diese Arbeit zeigt, dass insbesondere in Cloud Umgebungen Einsparpotential besteht und Unternehmen die Kosten für den Betrieb von Anwendungen bei ähnlicher Servicequalität reduzieren können. Eingekaufte Ressourcen können dadurch vom Betreiber einer Applikation besser ausgenutzt werden, anstatt durch den IaaS Anbieter überprovisioniert zu werden. Das Memory Management Model erlaubt die Bewertung der Speichereffizienz einer Applikation auf Basis von einfachen Metriken. Dadurch können Entwickler und Architekten den Speicherbedarf leichter abschätzen, bewerten, und überwachen. Darüber hinaus sind akkuratere Vorhersagemodelle für die Performance einer Applikation möglich, da diese nun die wichtigsten Aspekte der Laufzeit der Applikation umfassen. Weiterhin konnten wir durch den kombinierten Einsatz von Resource Demand Messungen und Abschätzmethoden, die Anzahl der Monitoringlösungen für die Performance Model Generierung erhöhen und dadurch eine breitere Anzahl an Technologien für die Generierung nutzen. Dies hilft Anwendungsszenarien wie Kapazitätsplanung und die automatische Performance Überwachung in Continuous Delivery Pipelines zu verbessern.

**Limitationen** Die eingeführten Methoden, Modelle und Techniken unterliegen einigen Limitationen. Zur Erstellung von Performance Modellen ist eine einheitliche Transaktions-ID über die komplette Anwendung notwendig. Ohne diese ID ist die Erkennung und Zuordnung einer Transaktion über mehrere Systeme hinweg nicht möglich und dadurch auch die Erstellung entsprechender Performancemodelle. In unserem Fall betrifft dies die Datenbank. Wir betrachten die Datenbank als Blackbox am Ende einer Transaktion und konnten mit Hilfe von Resource Demand Abschätzungen dennoch akkurate Performance Modelle erstellen. Entstehen allerdings weitere Aufrufkaskaden, durch beispielsweise eine zweite Datenbank oder Replikat, wird eine bessere Monitoringlösung benötigt, die eine Transaktions-ID für diese Art von Transaktionen einführt.

Obwohl wir eine große Anzahl von Topologien mit Hilfe von Rekombinationen und Mutationen testen, ist es dennoch möglich, dass der Optimierer nicht die beste mögliche Lösung findet. Je nach Komplexität der Applikation ist der Suchraum dafür einfach zu groß. Wir versuchen lokale Optima zu vermeiden, indem wir häufige Rekombinationen einsetzen. Häufig fanden wir aber eine deutlich bessere Lösung gegenüber einem naiven Deployment und konnten dadurch eine Optimierung erzielen und die Topologie einer verteilten Applikation verbessern.

# Contents

Acknowledgement . . . . .	iii
Abstract . . . . .	iv
Zusammenfassung . . . . .	vi
Contents . . . . .	ix
List of Figures . . . . .	xii
List of Tables . . . . .	xiv
List of Abbreviations and Acronyms . . . . .	xvi
<b>Part A</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Problem Statement and Motivation . . . . .	3
1.2 Research Goal and Research Questions . . . . .	3
1.3 Thesis Structure . . . . .	6
<b>2 Conceptual Background</b>	<b>8</b>
2.1 Performance Metrics . . . . .	8
2.2 Application Performance Management . . . . .	10
2.3 Model-based Performance Evaluation . . . . .	12
2.3.1 Queuing Networks . . . . .	13
2.3.2 Layered Queuing Networks . . . . .	14
2.3.3 Palladio Component Model . . . . .	16
2.4 Automatic Architecture Optimization . . . . .	21
2.4.1 Multi-objective Optimization Problems . . . . .	22
2.4.2 Basic Concepts of Evolutionary Optimization . . . . .	23
2.5 Memory Management . . . . .	25
<b>3 Research Methodology</b>	<b>28</b>
3.1 Research Design . . . . .	28
3.2 Research Methods . . . . .	29
3.3 Publications . . . . .	32
<b>Part B</b>	<b>36</b>
<b>4 Using Dynatrace Monitoring Data for Generating Performance Models of Java EE Applications</b>	<b>37</b>
4.1 Introduction . . . . .	38
4.2 Automatic Performance Model Generation Framework . . . . .	38
4.3 Conclusion & Future Work . . . . .	40

---

<b>5</b>	<b>Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques</b>	<b>41</b>
5.1	Introduction . . . . .	42
5.2	Extracting Resource Demands . . . . .	43
5.2.1	Performance Management Work - Tools Monitoring . . . . .	45
5.2.2	Dynatrace Application Monitoring . . . . .	45
5.2.3	Library for Resource Demand Estimation . . . . .	45
5.2.3.1	Demand estimation approaches . . . . .	45
5.2.3.2	Estimation approach selection . . . . .	46
5.3	Evaluation . . . . .	47
5.3.1	Standalone evaluation . . . . .	48
5.3.2	Distributed Setup . . . . .	50
5.4	Related Work . . . . .	54
5.5	Conclusion and Future Work . . . . .	55
<b>6</b>	<b>Full-Stack Performance Model Evaluation using Probabilistic Garbage Collection Simulation</b>	<b>56</b>
6.1	Introduction . . . . .	57
6.2	Garbage Collection Model . . . . .	57
6.3	Evaluation . . . . .	58
6.4	Conclusions . . . . .	61
<b>7</b>	<b>Model-based Prediction of Automatic Memory Management and Garbage Collection Behavior</b>	<b>62</b>
7.1	Related Work . . . . .	65
7.1.1	Memory management . . . . .	65
7.1.2	Performance Model Generation . . . . .	66
7.1.3	Performance Management . . . . .	67
7.2	Use Cases . . . . .	67
7.3	Research Method . . . . .	68
7.3.1	Memory Management and Garbage Collection . . . . .	70
7.3.2	Memory Meta-Model . . . . .	73
7.3.3	Observing memory demands and Garbage Collection (GC) . . . . .	78
7.3.4	Memory model generation . . . . .	81
7.3.5	Limitations . . . . .	87
7.4	Evaluation . . . . .	88
7.4.1	Experimental Setup . . . . .	88
7.4.2	Evaluation process . . . . .	90
7.4.3	Evaluation Results . . . . .	92
7.4.4	Discussion . . . . .	99
7.5	Conclusion . . . . .	99
<b>8</b>	<b>Optimization of Deployment Topologies for Distributed Enterprise Applications</b>	<b>101</b>
8.1	Introduction . . . . .	102
8.2	Related Work . . . . .	104
8.3	Enterprise application components . . . . .	105
8.4	Deployment Topology Optimization Process . . . . .	107

---

8.5	Performance Model Generator . . . . .	108
8.5.1	Monitoring . . . . .	109
8.5.2	Aggregation . . . . .	110
8.5.3	Model Generation . . . . .	110
8.6	Architecture Optimizer . . . . .	113
8.7	Evaluation . . . . .	116
8.8	Conclusion . . . . .	118
<b>9</b>	<b>SiaaS: Simulation as a Service</b>	<b>121</b>
9.1	Introduction . . . . .	122
9.2	Related Work . . . . .	122
9.3	Simulation Service . . . . .	123
9.4	Evaluation . . . . .	125
9.5	Conclusions . . . . .	126
<b>10</b>	<b>Multi-Objective Optimization of Deployment Topologies for Distributed Applications</b>	<b>127</b>
10.1	Introduction . . . . .	128
10.2	Related Work . . . . .	130
10.3	Distributed application components . . . . .	132
10.4	Deployment Topology Optimization Process . . . . .	134
10.5	Performance Model Generator . . . . .	135
10.5.1	Monitoring . . . . .	135
10.5.2	Aggregation . . . . .	137
10.5.3	Model Generation . . . . .	138
10.5.4	Cost model . . . . .	140
10.6	Architecture Optimizer . . . . .	141
10.7	Evaluation . . . . .	144
10.7.1	Evaluation System . . . . .	144
10.7.2	Evaluation Approach . . . . .	145
10.7.3	On-premise Evaluation . . . . .	146
10.7.4	Cloud Environment Evaluation . . . . .	149
10.8	Conclusion . . . . .	151
<b>Part C</b>		<b>153</b>
<b>11</b>	<b>Discussion</b>	<b>154</b>
11.1	Summary . . . . .	154
11.2	Limitations . . . . .	156
11.3	Contribution to Research . . . . .	158
11.4	Contribution to Practice . . . . .	159
11.5	Future Research . . . . .	159
	<b>References</b>	<b>160</b>

# List of Figures

1.1	Structure of this dissertation . . . . .	7
2.1	Resources and state causing the response time of a certain operation. (Adapted from Koch-Kemper (2015)) . . . . .	10
2.2	Example of a simple queuing network. (Adapted from Balsamo (2007); Koch-Kemper (2015)) . . . . .	13
2.3	Example of a layered queuing network. (Adapted from Cortellessa/Di Marco/Inverardi (2011)) . . . . .	15
2.4	Components of Palladio Component Model (PCM). (Adapted from Becker/Koziolok/Reussner (2009)) . . . . .	16
2.5	Repository model graph and example Resource Demanding Service Effect Specifications (RDSEFF). (Adapted from Brunnert/Vögele/Krcmar (2013))	17
2.6	System model graph (Adapted from Koch-Kemper (2015)) . . . . .	18
2.7	Resource environment model example . . . . .	19
2.8	Usage model example . . . . .	20
2.9	Software Architecture Optimization Approaches Taxonomy Aleti et al. (2013)	22
2.10	Exemplary multi-objective optimization problem with two objective functions that should be minimized (Adapted from Coello/Lamont/Van Veldhuisen (2007)) . . . . .	23
2.11	Memory space organization in Java adapted from Honk (2014) . . . . .	26
2.12	Memory trace of a system using Dynamic Memory Management . . . . .	26
2.13	Memory trace of a system using Automatic Memory Management . . . . .	27
3.1	Memory trace of a system using Dynamic Memory Management . . . . .	31
4.1	PMWT Performance Model Generation Framework . . . . .	39
5.1	Performance model generator framework (adapted from Brunnert/Neubig/Krcmar (2014); Willnecker/Brunnert et al. (2015a)) . . . . .	44
5.2	Measured and simulated response times on system entry point level . . . . .	49
5.3	Measured and simulated response times on component operation level . . . . .	51
5.4	Measured and simulated response times . . . . .	53
6.1	SPECjEnterpriseNEXT deployment . . . . .	59
7.1	Research method and evaluation process . . . . .	69
7.2	Memory space organization in Java adapted from Honk (2014) . . . . .	71
7.3	Memory trace of a system using Dynamic Memory Management . . . . .	72
7.4	Memory trace of a system using Automatic Memory Management . . . . .	73



7.5	Software Engineering concepts of a distributed application,PCM elements, and equivalent Java components adapted from Willnecker/Krcmar (2016) .	74
7.6	PCM extension for memory resources . . . . .	76
7.7	PCM extension accessing the newly introduced memory resource . . . . .	78
7.8	Example of growing committed memory using automatic memory management . . . . .	80
7.9	Performance model generation process adapted from Willnecker/Krcmar (2016); Brunnert/Krcmar (2017) . . . . .	81
7.10	Example of an automatic generated <i>Resource Environment</i> model for the SPECjEnterpriseNEXT Enterprise Application (EA) . . . . .	84
7.11	PCM RDSEFF representation of memory demands . . . . .	86
7.12	SPECjEnterprise2010 Orders Domain as an example EA . . . . .	88
7.13	SPECjEnterpriseNEXT Insurance Domain as example EA . . . . .	89
7.14	Experiment design to evaluate memory model accuracy . . . . .	90
7.15	Response times SPECjEnterprise2010 Experiment 1 (Replay) & 2 (Cloud)	93
7.16	Response times SPECjEnterprise2010 Experiment 3 (G1) & 4 (Alternate Workload) . . . . .	94
7.17	Response times SPECjEnterpriseNEXT Experiment 5 (Replay) . . . . .	95
7.18	Response times SPECjEnterpriseNEXT Experiment 6 (Cloud) . . . . .	96
7.19	Response times SPECjEnterpriseNEXT Experiment 7 (G1) . . . . .	97
7.20	Response times SPECjEnterpriseNEXT Experiment 8 (Alternative Workload) . . . . .	98
8.1	Enterprise application components (adapted from Becker/Koziolek/Reussner (2009); Brunnert/Krcmar (2017)). . . . .	106
8.2	Deployment topology optimization process. . . . .	107
8.3	Performance model generator framework (adapted from Willnecker/Dlugi et al. (2015c)). . . . .	109
8.4	PCM extension for memory resources. . . . .	111
8.5	Deployment Unit Optimizer. . . . .	114
8.6	SPECjEnterpriseNEXT test deployment. . . . .	117
8.7	Response time evaluation. . . . .	120
9.1	Simulation cluster architecture . . . . .	123
9.2	Simulation cluster in kubernetes instance . . . . .	125
10.1	Distributed application structure adapted from (Becker/Koziolek/Reussner, 2009; Brunnert/Krcmar, 2017) . . . . .	132
10.2	Deployment topology optimization process. . . . .	134
10.3	Core subsystem of deployment topology optimization . . . . .	136
10.4	PCM extension for memory resources. . . . .	138
10.5	Cost model extension . . . . .	140
10.6	Response time evaluation - On-premise environment . . . . .	146
10.7	Pareto-fronts along optimal deployments in both environments. X-Axis: costs, Y-Axis: average response times, Z-Axis: average resource utilization. Dots represent actual measurements, blue plane represents calculated Pareto-front) . . . . .	148
10.8	Response time evaluation - Cloud environment . . . . .	149

# List of Tables

2.1	Resource utilization definition . . . . .	9
3.1	Publications embedded in this thesis . . . . .	32
3.2	Further publications during the work on this dissertation . . . . .	34
4.1	Bibliographic details for P1 . . . . .	37
5.1	Bibliographic details for P2 . . . . .	41
5.2	Measured and simulated Central Processing Unit (CPU) utilization for system entry point level . . . . .	49
5.3	Measured and simulated CPU utilization for component operation level . . . . .	50
5.4	Software and hardware configuration of the system under test (SUT) . . . . .	52
5.5	Measured and simulated CPU utilization using Performance Management Work (PMW)-Tools monitoring . . . . .	53
5.6	Measured and simulated CPU utilization using Dynatrace AM . . . . .	53
6.1	Bibliographic details for P3 . . . . .	56
6.2	Measurement and simulation results . . . . .	60
7.1	Bibliographic details for P4 . . . . .	62
7.2	GC measurement data collection . . . . .	83
7.3	Software and hardware configuration for model generation . . . . .	91
7.4	List of all conducted experiments . . . . .	91
7.5	Measurement and simulation results for SPECjEnterprise2010 accessed by 200 concurrent users . . . . .	92
7.6	Measurement and simulation results for SPECjEnterpriseNEXT in an on-premise environment accessed by 140 concurrent users . . . . .	95
7.7	Measurement and simulation results for SPECjEnterpriseNEXT in a cloud environment accessed by 140 concurrent users . . . . .	96
7.8	Measurement and simulation results for SPECjEnterpriseNEXT using the G1 GC accessed by 140 concurrent users . . . . .	97
7.9	Measurement and simulation results for SPECjEnterpriseNEXT using an alternated workload accessed by 140 concurrent users . . . . .	98
8.1	Bibliographic details for P5 . . . . .	101
8.2	Software and hardware configuration of the deployment . . . . .	116
8.3	Measurement and simulation results . . . . .	118
9.1	Bibliographic details for P6 . . . . .	121

---

10.1	Bibliographic details for P7 . . . . .	127
10.2	Software and hardware configuration for model generation . . . . .	145
10.3	Measurement and simulation results for a selected topology on-premise accessed by 500 users . . . . .	147
10.4	Measurement and simulation results for a selected cloud topology accessed by 500 users users . . . . .	150
11.1	Key results of embedded publications . . . . .	157

# List of Abbreviations and Acronyms

<b>AM</b>	Application Monitoring	137
<b>API</b>	Application Programming Interface	158
<b>APM</b>	Application Performance Management	154
<b>AS</b>	Application Server	142
<b>AWS</b>	Amazon Web Services	156
<b>CDI</b>	Contexts and Dependency Injection	123
<b>CLS</b>	Command Line Simulator	156
<b>CPU</b>	Central Processing Unit	157
<b>CPUPE</b>	CPU utilization prediction error	49
<b>DA</b>	distributed application	156
<b>DB</b>	database	155
<b>DBMS</b>	Database Management System	144
<b>DML</b>	Decartes Modeling Language	13
<b>DUO</b>	Deployment Unit Optimizer	156
<b>EA</b>	Enterprise Application	155
<b>EC2</b>	Elastic Compute Cloud	156
<b>EE</b>	Enterprise Edition	133
<b>EJB</b>	Enterprise JavaBean	45
<b>GB</b>	Gigabyte	141
<b>GBit/s</b>	Gigabit-per-second	145
<b>GC</b>	Garbage Collection	155
<b>GHz</b>	Gigahertz	145
<b>HDD</b>	Hard Disk Drive	157
<b>HTTP</b>	Hypertext Transfer Protocol	89
<b>IaaS</b>	Infrastructure as a Service	159
<b>kB</b>	kilobyte	21
<b>LQN</b>	layered queueing network	130
<b>LTS</b>	Load Test Selector	156
<b>IQR</b>	Interquartile range	92
<b>JAX-RS</b>	Java API for RESTful Web Services	123

---

<b>JDBC</b>	Java Database Connectivity .....	145
<b>JSP</b>	JavaServer Page .....	48
<b>JMX</b>	Java Management Extensions .....	137
<b>JPA</b>	Java Persistence API .....	145
<b>JVM</b>	Java Virtual Machine .....	158
<b>LibReDE</b>	Library for Resource Demand Estimation .....	154
<b>MMCPU</b>	measured mean CPU utilization .....	49
<b>MRT</b>	measured response time .....	117
<b>PCM</b>	Palladio Component Model .....	155
<b>PET</b>	Performance Evaluation Tool .....	70
<b>PMG</b>	performance model generator .....	154
<b>PMW</b>	Performance Management Work .....	156
<b>PMWT</b>	Performance Management Work Tools .....	135
<b>RAM</b>	Random Access Memory .....	48
<b>RDSEFF</b>	Resource Demanding Service Effect Specifications .....	xii
<b>REST</b>	Representational State Transfer .....	145
<b>SaaS</b>	Software as a Service .....	156
<b>SAR</b>	System Activity Reporter .....	137
<b>SMCPU</b>	simulated mean CPU utilization .....	49
<b>SPE</b>	Software Performance Engineering .....	67
<b>SPEC</b>	Standard Performance Evaluation Corp. ....	43
<b>SRT</b>	simulated response time .....	117
<b>SUT</b>	system under test .....	69
<b>OS</b>	operating system .....	132
<b>UML</b>	Unified Modeling Language .....	17
<b>UI</b>	User Interface .....	155
<b>VM</b>	Virtual Machine .....	128
<b>WS</b>	Web Service .....	155

# Part A

# Chapter 1

## Introduction

Enterprise Applications (EAs) are typically implemented as distributed systems composed of components and distributed throughout multiple Virtual Machines (VMs) or hosts. These systems often utilize corresponding resources far below available capacity. Especially in managed environments, such as cloud Infrastructure as a Service (IaaS) environments, this induces unnecessary costs. Changing the deployment topology in order to increase resource utilization demands several tests on environments comparable to the production system. The effort to test a certain topology can be reduced by using performance models.

Such models can be used to landscape system architectures and to simulate changes in the deployment topology or resource environment without utilizing actual resource environments. Therefore, this work aims on generating performance models for distributed applications (DAs) focusing on EAs. One of the main challenges is to consider all major resource types regarding the performance of an DA: Central Processing Unit (CPU), Hard Disk Drive (HDD), network, and memory. Especially automatic memory management has been disregarded by modern architecture-level performance models. However, leaving out one major resource leaves unrepresentative models and inaccurate simulation results. Therefore, we further introduce an automatic memory management model and a corresponding performance model generator (PMG) which creates holistic performance models.

Based on these models, we propose an architecture optimizer that searches for optimal deployment topologies by conducting a large amount of simulations. These simulations are executed in parallel using a distributed simulation service. We evaluate the prediction accuracy compared to actual deployments and the quality of the selected topologies in terms of resource utilization, response times, and costs. This allows architects to evaluate component changes and topology variations without replicas of the production system.

## 1.1 Problem Statement and Motivation

Distributed system architectures are state of the art in large scale EAs (Brunnert/Wischer/Krcmar, 2014). These systems are composed of components that can be moved and replicated running on multiple instances assigned by deployment management software (Woolf, 2009). Deployment topologies based on different allocations of multiple components and their relationship form a distributed system architecture. Selecting the right topology is a complicated task that today is merely assisted by logical topology recommendations (Woolf, 2009; Koziolok/Koziolok/Reussner, 2011).

Logical topologies often utilize the hardware below their possible capacity as the average processing load of data centers today is less than 20% (Pawlish/Varde/Robila, 2012). Virtualized server environments already reduce this over-provisioning, thus increasing the hardware utilization (Speitkamp/Bichler, 2010). However, virtualized server environments limit the optimization opportunities to the granularity level of single virtual machines. Furthermore, the benefits of virtualization are today used by IaaS providers to load their servers to capacity. Application operators in their very own interest should increase the load per VM they rent, in order to get the best value for the cost of running server instances. Distributed systems, especially ones using the Microservice pattern rely on fine-grained deployment units, allowing operation engineers and architects to utilize unused capacity more efficiently (Fowler/Lewis, 2014). Planning and testing such changes in productive environments comprises risks for the stability. In addition, productive alike test environments and productive systems have comparable prices. Furthermore, such environments are usually used to capacity by several projects executing load tests. Therefore, we propose to use a purely software-based solution for planning, optimizing, and evaluating deployment topologies using performance models.

Performance models can be used to optimize resource utilization, response times, and costs by evaluating alternative deployment topologies using simulations (Koziolok/Koziolok/Reussner, 2011; Brunnert/Vögele/Krcmar, 2013). Even for small businesses and their DA the potential savings in IaaS environments can exceed millions of annual payments (Roussel/Branson, 2017). This thesis aims on automatically generating performance models for DAs in order to optimize their deployment topologies and to analyze the impact of changing or introducing new components.

## 1.2 Research Goal and Research Questions

This section provides an overview of the main research questions (RQ) and how this dissertation tries to answer them.

**RQ1: Which model generation techniques and monitoring approaches are necessary to generate full-stack performance models of DAs**

Automatic performance model generator technologies and resource estimation approaches have been proposed to the scientific community (Brunnert/Krcmar, 2017; Spinner et al.,



2015). The proposed approaches focus on single systems and partly ignore the distribution aspect of modern EAs. The available model generators disregard the topology aspect and mainly focus on application servers, their resource demands, and relationships. Research on full-stack distributed systems including the deployment topology of their components was barely conducted.

This change of scope requires reconsidering the level of granularity feasible for the performance model generation. Currently available approaches generate fine-grained performance models (Brunnert/Krcmar, 2017). Computing power and a processable complexity level limit the granularity of larger performance models. The granularity decision also depends on the entailed monitoring solutions. Industry as well as scientific solutions are available to monitor full-stack distributed systems (van Hoorn/Waller/Hasselbring, 2012; Greifeneder, 2011).

Selecting an appropriate monitoring solution, choosing the right level of granularity for the performance models, as well as adapting and extending available performance model generators are the main challenges for this research question. This work will identify, evaluate, and synthesize available monitoring solutions and model generators. We select applicable monitoring solutions and conduct a series of controlled experiments using SPEC benchmarks (Henning, 2006). These experiments identify the combination of monitoring and model generator solutions that are best suited for full-stack DA models.

**RQ2: Which modeling and simulation techniques can represent automatic memory management of DAs without excessive computational requirements?**

Memory resources have been widely disregarded in current performance models and corresponding generators (Brunnert/Krcmar, 2017; Brosig/Huber/Kounev, 2014; Walter et al., 2017). If memory resources are applied, only simple dynamic memory management features are available (Brunnert/Krcmar, 2017). Modern runtimes like the Java Virtual Machine (JVM) or .NET and scripting environments rely heavily on automatic memory management (Libič et al., 2014; Schildt, 2014). Especially the clean-up of memory, so called Garbage Collection (GC) is barely represented in performance models (Libič et al., 2014). Available solutions representing memory resources, automatic memory management, and GC are more complex than executing the application itself (Libič et al., 2014).

This research questions aims on designing, implementing, and evaluating a memory resource representation that allows simulations with accurate prediction quality while requiring less computational resources than executing the actual system. We extend established performance models with a memory resource representation that allows to simulate dynamic and automatic memory management. Furthermore, we provide a monitoring and model generation solution that can detect GC behavior and generate model instances simulating the observed behavior. Therefore, we can predict how changes to the system, the workload, or the resource environment change the GC behavior and the influence of these changes on the performance metrics.

**RQ3: Which deployment optimization approaches can be adapted for architecture-level performance models for DA and how accurate are these opti-**

**mizations compared to the actual system?**

This research question is settled between the research domains of architecture optimization and performance model generation and tries to adapt and extend available optimization algorithms to optimize the deployment topologies of a productive DA (Koziolk/Koziolk/Reussner, 2011; Salehie/Tahvildari, 2009; Huber/Brosig et al., 2016).

Architecture optimizations have been proposed to improve logical component topologies and support architecture decisions during design time (Koziolk/Koziolk/Reussner, 2011). Especially deployment topology decisions are a complex and time consuming activity (Koziolk/Koziolk/Reussner, 2011). Unfortunately, during design time resource demands and the infrastructure hosting the system can only be estimated. The accuracy of the model as well as the optimization recommendations depend on these assumptions.

Optimization of running systems is also covered in the domain of self-adaptive software systems. Such system require the existence of accurate (performance) models of real systems (Salehie/Tahvildari, 2009; Huber/Brosig et al., 2016). Such models in regard to performance are generated by PMGs (Brosig/Huber/Kounev, 2014; Brunnert/Krcmar, 2017; Walter et al., 2017). As of today, several approaches for automatic PMGs including resource demand estimations exist (Brunnert/Krcmar, 2017; Brosig/Kounev/Krogmann, 2009; Spinner/Casale et al., 2014; Walter et al., 2017). This work adapts and extends existing automatic PMG approaches to work with distributed EAs. The generated models are used to optimize deployment topologies. These systems act automatically to a certain degree and can react within a limited set of rules. These rules can include increasing the number of available servers to compensate an unusually high amount of user requests or detecting security breaches and close certain system accesses (Salehie/Tahvildari, 2009). To detect the need of action self-adaptive software monitors performance data like hardware utilization and network throughput (Huber/Brosig et al., 2016). This research question aims on selecting a suitable optimization algorithm that is applied on the generated performance models of RQ1 and RQ2.

Evaluating the selected and extended model generator and optimization algorithm requires three steps: Validating the accuracy of the generated architecture-level performance models, the change prediction capabilities, and the selected optimization algorithm.

The evaluation of the performance model is conducted by a series of controlled experiments. Hence, we generate such models from real DAs. These generated models are used to simulate different load intensity and usage scenarios. The same scenarios are processed on real systems. When processing such test runs, performance metrics like throughput, response times, or utilization are measured and compared with the simulation results. The accuracy of the model is assessed based on the error among simulation results and measurements of the real system.

The last step is the evaluation of the optimization algorithm. The algorithm calculates alternative deployment topologies by predicting deployment topology changes. The topologies are optimized in terms of response times, resource utilization, and/or costs. The optimization algorithm presents alternative topologies that improve the architecture of

the evaluated DA based on the optimization goal(s). The results help to understand the optimization potential of running systems, the dependencies of distributed components in terms of performance, and suggests optimization of currently applied topologies.

## 1.3 Thesis Structure

The structure of this dissertation is shown in Figure 1.1. The thesis is partitioned in three major parts (Part A, Part B, and Part C)

**Part A** contains the introduction, the conceptual background, and a description of the used research methodology. The introduction (chapter 1) is structured into the problem statement, the motivation, and the derived research questions of this dissertation. The next chapter covers the conceptual background (chapter 2) and contains preliminaries about performance metrics, application performance management, model-based performance engineering, automatic architecture optimization, and memory management techniques. Furthermore, we elaborate on the research methodology (chapter 3) introduce the research design, the research methods, and present a list of the embedded publications.

**Part B** contains the seven embedded publications P1 to P7 (chapter 4 - chapter 10). These publications resulted from research done by the author as part of this dissertation. A short summary of the content of these publications is presented in Section 3.3.

**Part C** concludes this dissertation with a discussion (chapter 11). First, the findings of the publications are summarized. Afterwards, limitations and the contribution to research and practice are explained. This dissertation concludes with future research opportunities and open questions.

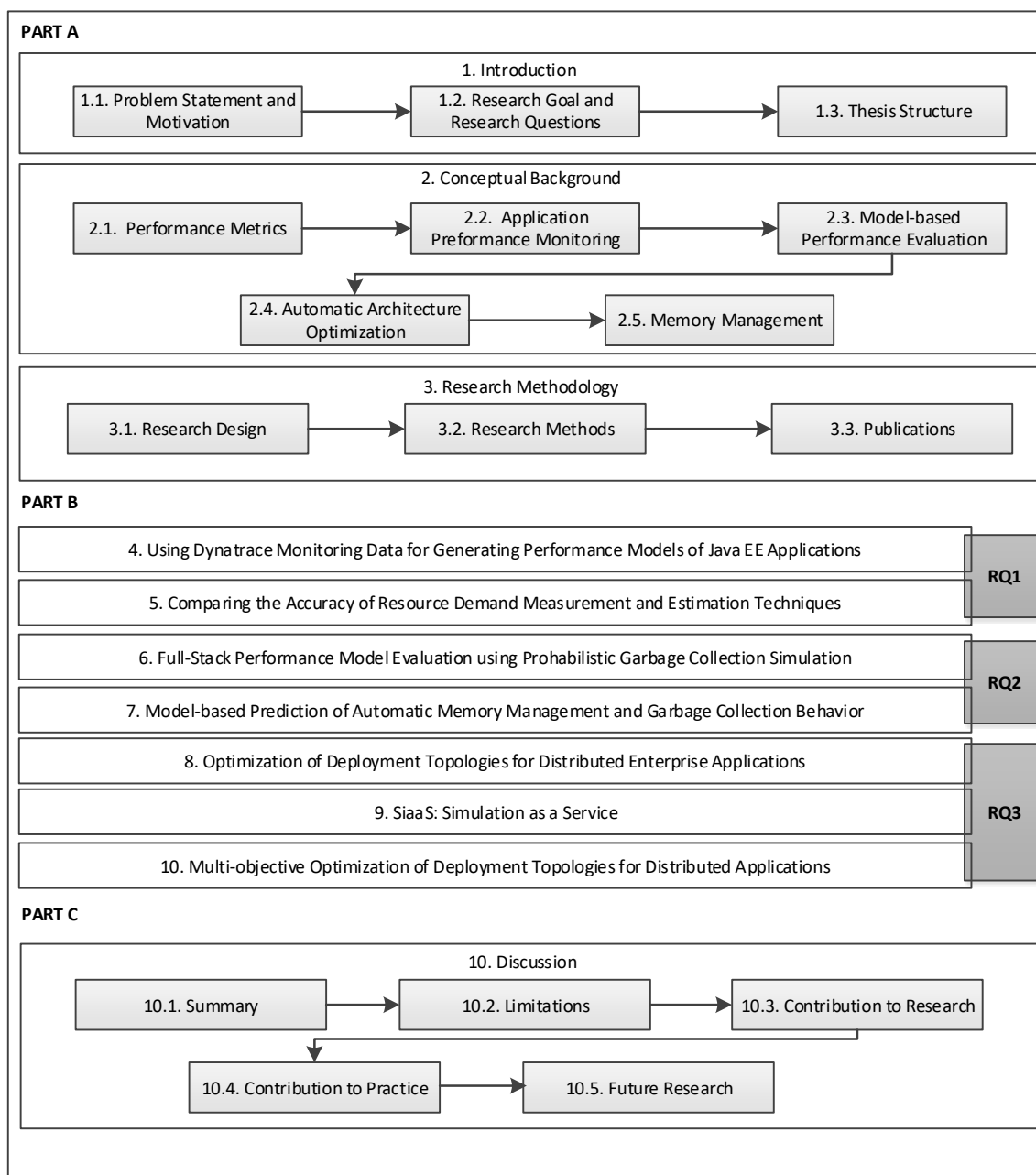


Figure 1.1: Structure of this dissertation

# Chapter 2

## Conceptual Background

This work is based on research in the field of application performance management, model-based performance evaluation, and architecture optimization, and memory management approaches. In this chapter the basic principles and the relations between these are presented, which are required to understand the concepts and contributions of this dissertation.

### 2.1 Performance Metrics

This work focuses on the utilization of four major resources of a typical EA, the throughput, and the response times of business transactions. We consider the following major resource type in our research:

- (i) CPU as the major processing unit for EA executing all calculations and usually the resource with the major portion in origination response times.
- (ii) Network in terms of bandwidth and latency, which play an important role in the communication between client and server but also between services in a distributed architecture.
- (iii) Memory in terms of Random Access Memory (RAM) holding application data and program code for the EA. We consider application data or the heap space as this is the variable part of memory that needs to be considered for capacity planning and the part that leads to application crashes if not managed properly.
- (iv) HDD as storage resource for larger datasets, files, and images used and provided by EAs.

We consider the utilization regarding the above mentioned resource types. Depending on the type the utilization is calculated differently as depicted in Table 2.1. The utilization of these resources are measured, simulated, and compared in multiple evaluations of this work.

Resource	Utilization calculation
CPU	$\frac{\text{Number of cycles necessary to execute an operation}}{\text{Number of totally available cycles}}$
Network	$\frac{\text{Send and received blocks (bytes)}}{\text{Total bandwidth}(\frac{\text{Bytes}}{\text{Time}})}$
Memory	$\frac{\text{Mean allocated memory (bytes)}}{\text{Mean committed memory (bytes)}}$
HDD Read	$\frac{\text{Read blocks (bytes)}}{\text{Max read rate}(\frac{\text{Bytes}}{\text{Time}})}$
HDD Write	$\frac{\text{Written blocks (bytes)}}{\text{Max write rate}(\frac{\text{Bytes}}{\text{Time}})}$

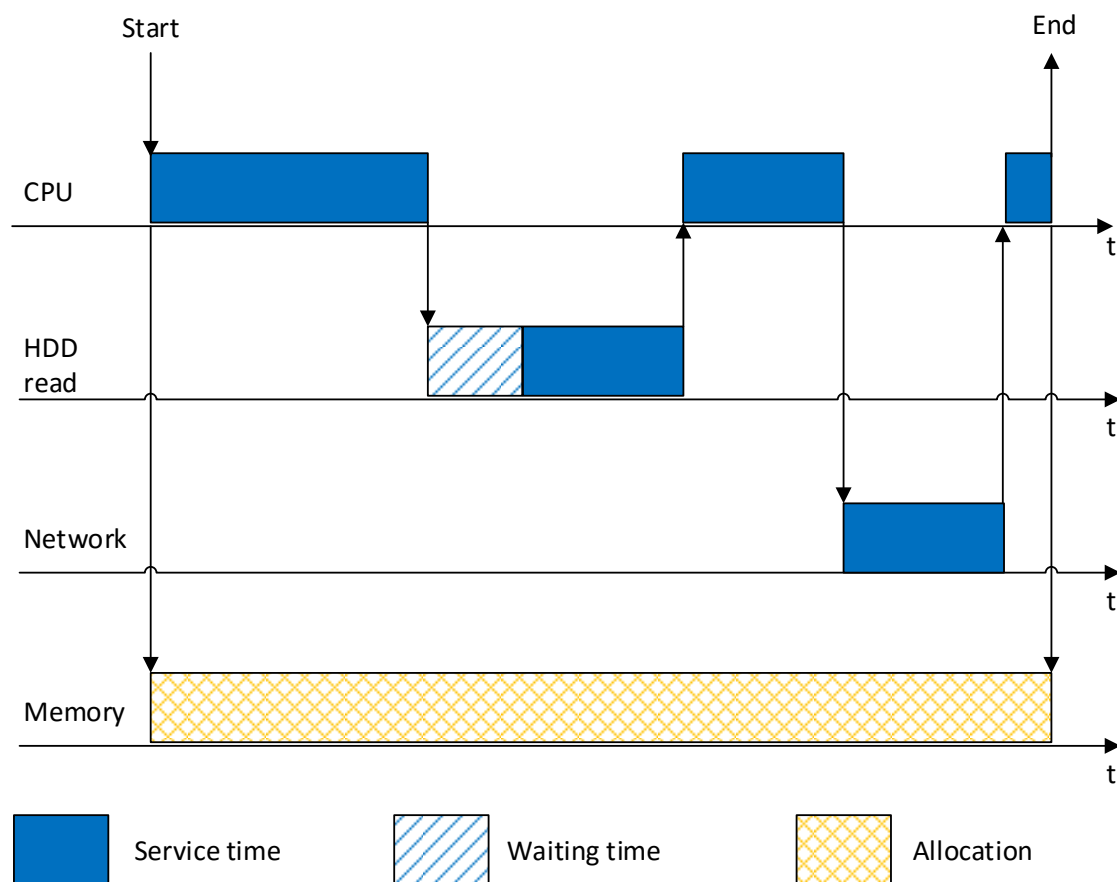
**Table 2.1:** *Resource utilization definition*

As a second important metric beside resource utilization, we consider response times. The response time of a certain operation or (business) transaction means the length of time take for the system to react. In EA this usually means a full round trip from a user action to a visible response on the client or the complete length of time from starting till completing a certain (business) transaction. The response time is usually influence by the computation time, resource contention in shared environments, availability of HDD and memory, as well as network bandwidth and latency. Furthermore, the availability of application specific pools, mutual exclusion flags (e.g., mutex), and GC runs can delay the execution and thus increase the response time of an operation or transaction (Forouzan, 2013).

Figure 2.1 shows an example of an operation and how different components lead to a certain response time. Service time, meaning execution in terms of CPU, as well as waiting times caused by resource contention lead to the overall response time here. Furthermore, a certain amount of memory is allocated for processing the operation. This does not necessarily increase the response times, nevertheless the number of parallel executions is limited by the amount of memory currently available. Faster operation cycles release memory earlier, allowing to process more transactions.

Furthermore, we consider throughput as another important performance metric that is evaluated in this work. We observe throughput as the amount of bytes send or received through network in a certain time, but also the amount of users and/or transactions processes in a certain time frame. We calculate throughput using Equation 2.1. For example, after executing a 10 minute load test, we divide the number of processes transactions by the total time in seconds and receive the mean amount of transactions processed per second. Besides measuring and calculating the throughput this approach is also applied to calculate the inter arrival time between two transactions. We use the inter arrival time to define and configure the workload of certain business transactions in our research.

$$\text{throughput} = \frac{\text{Total Number Of Items Processed}}{\text{Time}} \quad (2.1)$$



**Figure 2.1:** Resources and state causing the response time of a certain operation. (Adapted from Koch-Kemper (2015))

## 2.2 Application Performance Management

Multiple definitions for the term Application Performance Management (APM) exist. Mendel (2013) defines Application Performance Management (APM) as a tool or set of tools to monitor and manage the performance and availability of a software application. A more complex classification is provided by Haight/De Silva (2016), who define five core functionalities of APM:

- (i) Tracing (in real-time) of algorithm invocations that compound the monitored application.
- (ii) Measurements and presentation of hard- and software resources, which were used to execute these algorithms.
- (iii) Analysis and determination of successful (depending on the applications purpose) execution of its transactions.
- (iv) Recording of latencies and response times caused by the sequences of the algorithms of the monitored application.
- (v) Root-cause analysis and determination of errors, resource allocation, and latencies/response times.

The abbreviation APM is sometime used in the context of Application Performance Monitoring, which is a subset of Application Performance Management. Monitoring is an important part, but only deliver a large number of data points, which are worthless without the right presentation. APM solutions in the sense of this work, provide a larger set of tools that go beyond monitoring and support the analyst in managing (monitoring, maintaining, improving) the performance and availability of complex distributed EA.

We distinguish between two major categories of monitoring data in our work. Coarse-grained system monitoring and fine-grained application monitoring (Spinner et al., 2015). Coarse-grained monitoring data has long been standard in performance and availability monitoring and management. System or infrastructure monitoring solutions like Nagios<sup>1</sup> and Hyperic<sup>2</sup> use such coarse-grained data to manage the health of hosts. Standard linux system tools like System Activity Reporter (SAR), TOP<sup>3</sup>, or procfs<sup>4</sup> provide resource metrics on system and process level. Other sources might be application or server specific logs like the Apache HTTP server<sup>5</sup> access logs. However, the main disadvantage of this approach is that the performance management is limited to the process level of an application and does not provide information about the transaction flow, the algorithms, their resource usage, and their latency.

Modern commercial APM solutions like Dynatrace<sup>6</sup> or AppDynamics<sup>7</sup> and open-source APM suites like Kieker<sup>8</sup> or InspectIT<sup>9</sup> provide detailed application and transaction specific monitoring data (van Hoorn/Waller/Hasselbring, 2012; Okanović et al., 2016). This allows to detect, analyze, and resolve performance issues on the component or operation level, whilst classical coarse-grained APM solutions required secondary information sources like application logs to improve performance and availability of applications. The main disadvantages compared to coarse-grained system monitoring is the amount of data to be managed and the instrumentation overhead that is usually higher compared to system monitoring.

Even though all the above mentioned APM solutions provide details about the application that is monitored, the instrumentation technique collection the data might be different depending on the selected solution. Solutions like Dynatrace use Byte Code instrumentation and actually detect performance metrics at the boundaries of instrumented operations and components (Angel et al., 2001). This provides very accurate instrumentation data but requires to alter the actual deployed system as the byte code is altered in order to measure and collect data about resource utilization and the current transaction. Other solutions like AppDynamics use distribution techniques based on frequent samples collected on system and application level. Information about the active transactions and the current resource utilization of the monitored processes are collected. Afterwards, the monitoring solution distributes the resource utilization amongst the active transactions.

---

<sup>1</sup><https://www.nagios.org/>

<sup>2</sup><https://github.com/hyperic>

<sup>3</sup><https://linux.die.net/man/1/top>

<sup>4</sup><https://www.kernel.org/doc/Documentation/filesystems/proc.txt>

<sup>5</sup><https://httpd.apache.org/>

<sup>6</sup><https://www.dynatrace.com/>

<sup>7</sup><https://www.appdynamics.com/>

<sup>8</sup><http://kieker-monitoring.net/>

<sup>9</sup><http://www.inspectit.rocks/>



This provides a blurry view on the actual resource consumption compared to byte code instrumentation, but requires usually less resources than byte code instrumentation (Spinner et al., 2015). A third way of collecting this fine-grained monitoring data is to intercept calls using application or framework specific Application Programming Interfaces (APIs). The Performance Management Work Tools (PMWT) Java Enterprise Edition (EE) agent and the RETIT<sup>10</sup> Java EE agent use this technique based on interception APIs of Java EE (Brunnert/Vögele/Krcmar, 2013). This instrumentation technique can be seen as a compromise. It does not require to alter the code of the complete application but only to add certain interceptors, but requires APIs that allow to hook into the transaction process.

In this work all above mentioned approaches are used in different contexts. We use fine-grained monitoring data to generate performance models using leading APM solutions like Dynatrace, custom solutions like PMWT and RETIT Java EE agent, but also coarse-grained monitoring data in certain contexts (Brunnert/Vögele/Krcmar, 2013; Willnecker/Brunnert et al., 2015a; Spinner et al., 2015). In general fine-grained monitoring data is necessary for the model generation, but we developed and integrated techniques to use coarse-grained monitoring data to generate models representing database access (Willnecker/Brunnert et al., 2015a). Coarse-grained monitoring data plays an important role in our evaluation approaches. We compare simulation runs with total resource utilization, throughput, and response times in order to show the accuracy of our models and their prediction quality.

## 2.3 Model-based Performance Evaluation

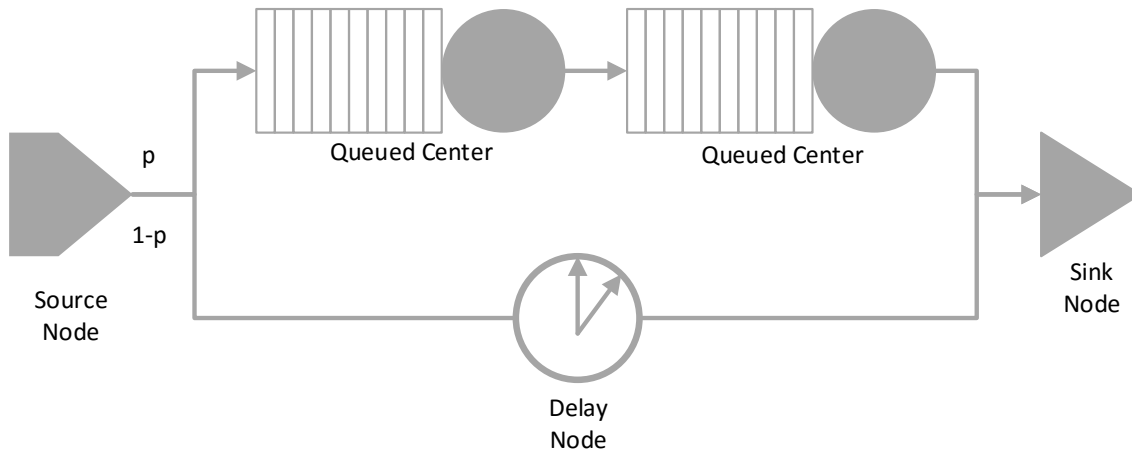
Model-based performance evaluation is a technique that is usually applied to predict the performance of software systems, often at design time (Brunnert et al., 2015). Recent research generates such models based on APM or coarse-grained monitoring in order to create and use accurate models (Brunnert/Krcmar, 2017; Brosig/Huber/Kounev, 2014; Walter et al., 2017). Our research uses software performance models in all stages and in an integrative way. This means, we do not create such models at design-time manually, as this often outweighs their benefits (Kounev/Brosig/Huber, 2014). We generate performance models out of APM data, alter certain model parts, and predict the performance metrics resource utilization, response times, and throughput.

Performance models represent application systems in an abstract way. The model uses parameters and abstract control-flow blocks to specify the behavior of an application. The level of detail is usually limited to the parts that have influence on the performance of the system. Other parts are usually not specified in a performance model. This still comprises a lot of hard- and software components like the resource environment, operations and their resource consumption, or the usage model.

Simulation engines or solvers allow to predict performance metrics of such performance models. This allows to for instance predict the resource utilization under a certain work-

---

<sup>10</sup><http://retit.de/>



**Figure 2.2:** *Example of a simple queuing network. (Adapted from Balsamo (2007); Koch-Kemper (2015))*

load. The accuracy of these predictions relies on the quality of the performance model and its parameters, mainly its resource demands and resource capabilities. Solver predict these metrics usually faster, but are limited in terms of the model complexity. Simulation engines are in general slower, but allow to simulate much more complex and realistic models.

Tools or workbenches to create, maintain, change, and simulate or solve such models exist. Several such models are based on already established model notations such as (Layered) Queuing Networks or Petri Nets (Cortellessa/Di Marco/Inverardi, 2011). This allows to reuse already established solver and simulation techniques (Koch-Kemper, 2015). However, newly developed architecture-level performance models such as Palladio Component Model (PCM) and Decartes Modeling Language (DML) exist, which do not rely on such classical meta-model techniques (Becker/Koziolek/Reussner, 2009; Reussner et al., 2016; Kounev/Brosig/Huber, 2014). They are still compatible to some extent to layered queuing networks (LQNs). In our work we use and extend LQNs and PCM. Therefore, we only present these model techniques in the following.

### 2.3.1 Queuing Networks

A certain amount of resource consumption is necessary every time an operation of an application is invoked (e.g., HTTP request triggered by a button click in an EA). These resources are limited, leading to a certain wait time if the resource (e.g., CPU or HDD) is not available yet. *Queuing Networks* model and represent this as so-called *Queued Center* (Cortellessa/Di Marco/Inverardi, 2011).

Figure 2.2 shows such *Queued Center* that consists of a resource server (typically modeled as a circle) and a waiting queue in front of it. The server represents the resource itself (e.g., the CPU), which is utilized by *Tasks* or *Jobs*. Waiting queues develop and are

filled as usually multiple *Tasks* access the same resource at the same time. As soon as the resource completes a *Tasks* the scheduling strategy (e.g, “First-Come First-Served“ or “Last-Come First-Served“) select the next *Job* from the waiting queue which again blocks the resources. In that manner *Jobs* travel through a directed graph from one center to another. Annotated branches contain the probability of selecting one of the branches. This allows to model control-flows and a certain non determinism. Furthermore, *Delay Nodes* allows to model waiting times, which can represent network latency, to delay the execution of a *Job* in the model.

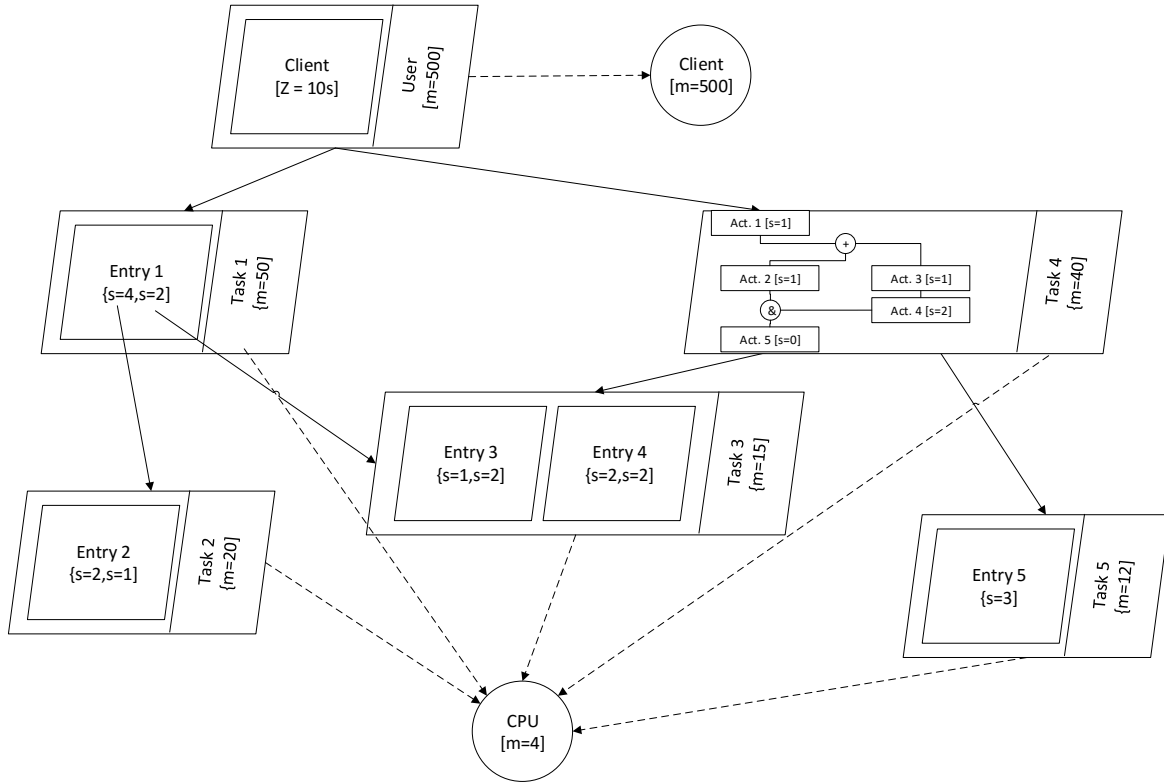
Two types of *Queuing Network* exist regarding the job creation: Open and closed *Queuing Network*. An open *Queuing network* contains source and sink nodes marking start end end points of a transaction. Figure 2.2 shows an open *Queuing Network*. New *Jobs* arrive at the source node and are processed until they leave the network using the sink node. Closed *Queuing Networks* contain so-called *Terminal Nodes*. When a *Job* arrives at a *Terminal Node*, it is stored until a certain time period passes. This represents the think time of a user. After this time passed, a new *Job* with the same properties is created and begins to travel through the network starting at the *Terminal Node* (Koch-Kemper, 2015).

The presented techniques allow to model a basic application, its resources, and simple control-flows. We need parameters in order to predict performance metrics using *Queuing Network*. Multiple approaches exist to realize this (Cortellessa/Di Marco/Inverardi, 2011). In standard *Queuing Networks* the *Job* size is always the same. To represent different resource consumption each *Job* can contain a length representing the required resource service time to process. Furthermore, *Delay Nodes* need a waiting queue to represent the bandwidth of a network and not just the latency.

### 2.3.2 Layered Queuing Networks

Classical *Queuing Networks* represent EAs as a number of connected *Queued Center*, multiple job classes, and workload specifications. This model language is often to simple to represent modern complex component-based software (Cortellessa/Di Marco/Inverardi, 2011). Therefore *LQNs* were introduced as an extension to *Queuing Networks*. *LQNs* represent component-based software in form of layers that allow a flexible mapping to the consumed resources (Rolia/Sevcik, 1995; Franks et al., 1995).

*LQNs* consists of *Processors* typically represented as circles and *Tasks* that are represented as parallelograms. These parallelograms contain one or multiple entries. *Processors* represent a hardware resources, while *Tasks* represent the logic of a software or database system. The arrangement is hierarchical, so that elements that are depicted above other elements can access only lower elements. Therefore, the lowest element is always the hardware resource (Cortellessa/Di Marco/Inverardi, 2011; Rolia/Sevcik, 1995). In the example depicted in Figure 2.3 we chose a CPU as hardware resource of which access is modeled in the *LQN*. In *LQNs* *Processors* and *Tasks* have their own waiting queue with a specified length and scheduling strategy.

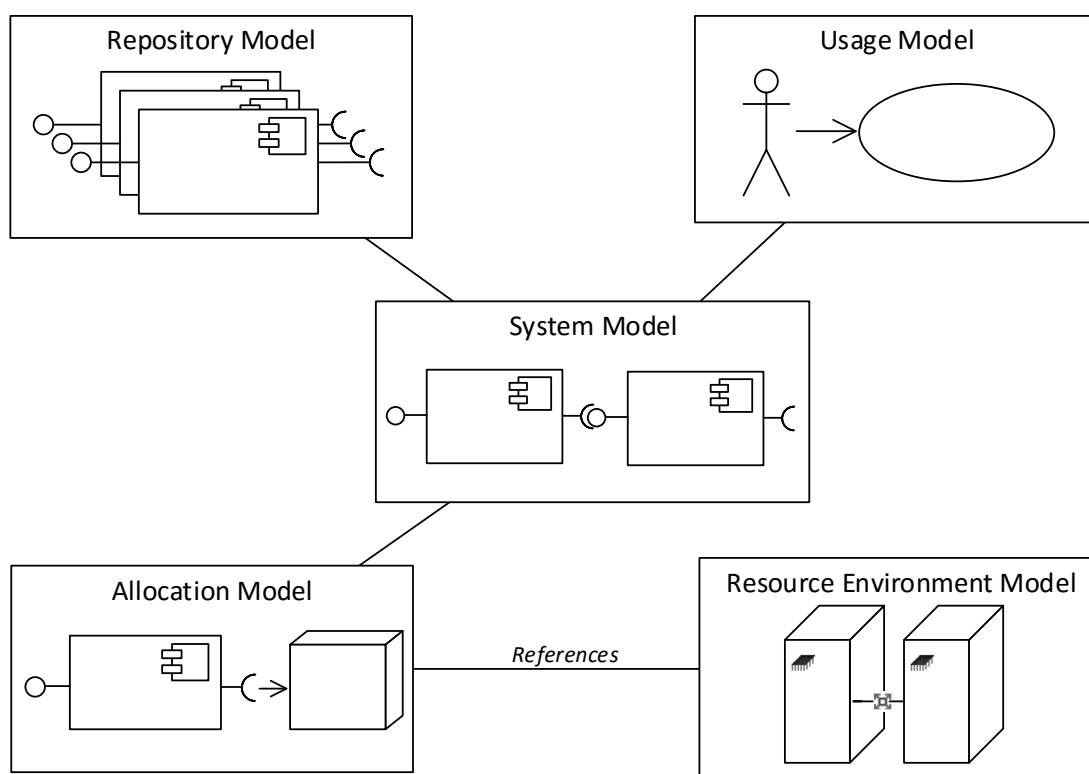


**Figure 2.3:** Example of a layered queuing network. (Adapted from Cortellessa/Di Marco/Inverardi (2011))

The number of parallel available processing slots is defined by the multiplicity  $m$ . For a *Processor* this could mean the number of CPU cores while for *Tasks* these slots can represent a thread pool. The operations are modeled as entries which can consist of multiple phases. These phases are depicted as comma-separated lists of *Service Times*  $s$  below the entry name in Figure 2.3. In our example Entry 1 consists of two phases. The first phase requires 4 and the second 2 service times of the *Processors* CPU. T

*Activity Graphs* are another model element that allow to represent complex control-flow including parallel executions (Cortellessa/Di Marco/Inverardi, 2011). In our example Task 4 uses this *Activity Graphs*. The “+” and “&” elements represent forks and joins. This allows to model parallel execution and synchronization. In our example Ac, 2 and 3/4 are executed in parallel until they are joined again before ending with the last *Activity* in this Task.

As a final element of this modeling language *Reference Tasks* are the top level *Task*. They represent access from users as they cannot be accessed from other *Tasks*. They contain a *Think Time*  $Z$  instead of *Service Time*  $S$ . The only resource used by these types of *Tasks* are *Client-Pools*. They represent the number of users accessing the system. Therefore, the workload here is defined by the number of users and the think time on the top level of the LQN (Koch-Kemper, 2015).



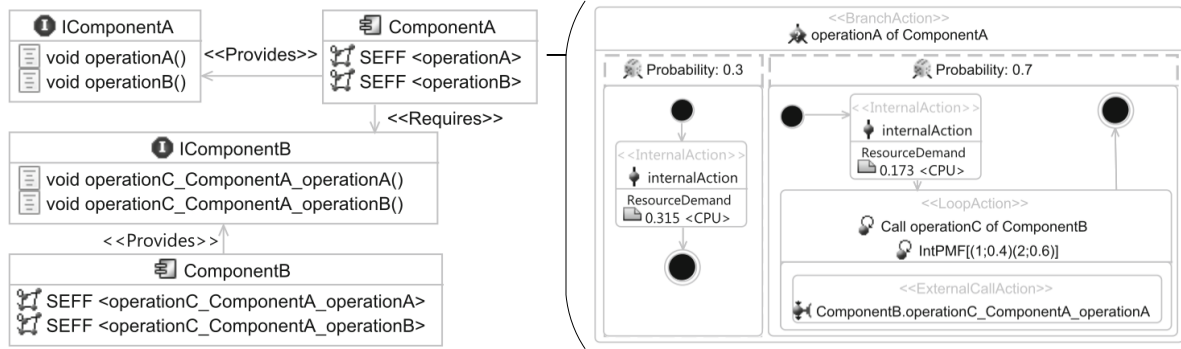
**Figure 2.4:** *Components of PCM. (Adapted from Becker/Koziolok/Reussner (2009))*

### 2.3.3 Palladio Component Model

Even though LQNs allow for much more complex component-based systems, the multiple layers, distributed systems, and an abstraction of the hardware is hard to accomplish. One major problem of the above mentioned approaches is the level of abstraction. Instead of modeling and describing the system as it is, queuing networks require a certain abstraction and mapping to the actual system. Another approach that does not require this level of abstraction but allows for modeling the systems as it is are architecture-level performance models. Amongst these model types PCM is one of the most prominent representatives (Becker/Koziolok/Reussner, 2009; Reussner et al., 2016).

PCM was created to represent component-based software systems in an expressive model that can be used to represent complex distributed applications. The meta-model defines five sub-models as shown in Figure 2.4 that are referenced by each other reflecting several aspects of a software system. Furthermore, Becker/Koziolok/Reussner (2009) defines several roles each responsible for different sub-models:

- (i) *Component Developer* specify and implement the software components and thus create the *Repository Model* of the PCM.
- (ii) *Software Architects* create and assemble several software components to a consistent complete system. In PCM this is represented in *System Model*.



**Figure 2.5:** *Repository model graph and example RDSEFF.* (Adapted from Brunner/Vögele/Krcmar (2013))

- (iii) *System Deployer* account for the allocation and operation of the resources required by the software system. This is defined in the *Resource Environment Model* and the *Allocation Model*.
- (iv) *Business Domain Experts* know the users and their behavior regarding the systems. This workload definition is in PCM configured in the *Usage Model*.

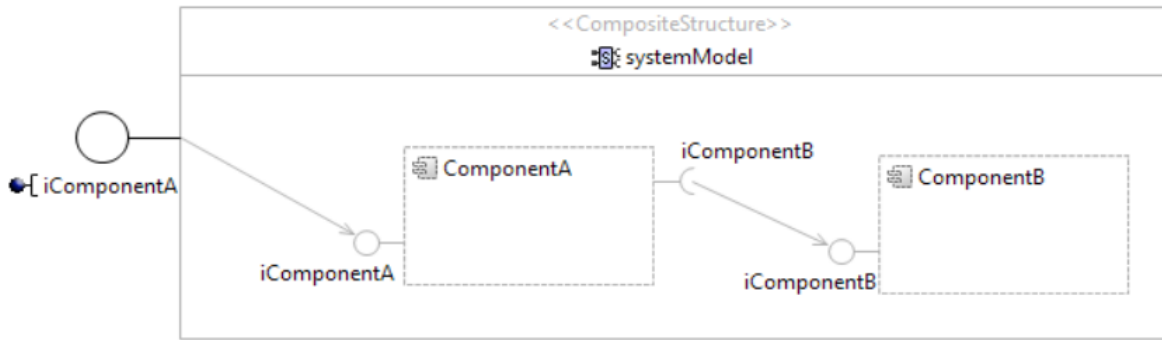
We will describe the five sub-models in more detail in the following paragraphs:

**Repository Model** The core sub-model of PCM is the *Repository Model*. All components of the application and their dependencies are defined in this sub-model. Each component can provide interfaces that are accessible from other components. Figure 2.5 shows two example components on the left side. *ComponentA* provide an interface called *IComponentA* and uses the interface *IComponentB* which is provided by *ComponentB*. The relationships are defined by the graph and its annotations *Provides* and *Requires*.

Each interface defines a set of operations, which are provided by all components that implement this interface. This implementations are in PCM defined in so-called RDSEFF. These RDSEFFs allow to specify the behavior of an operation in similar way as activity models in Unified Modeling Language (UML) (Rumbaugh/Jacobson/Booch, 2004). RDSEFFs contain annotations about performance and reliability in order to simulate the performance and availability behavior of the application (Becker/Koziolek/Reussner, 2009; Reussner/Becker et al., 2009). An example of such an RDSEFF is depicted on the right side of Figure 2.5.

Control-flows can be modeled using loops, forks, joins, and branches. In our example we use one *BranchAction* with two branches. Figure 2.5 shows these two branches annotated with different probabilities. In 30% of all cases the left branch is executed and with a likelihood of 70% the right branch. The operation executes different actions depending on the branch with different resource demands.

The right branch also contains a *LoopAction*. The number of iterations of this action is defined by a distribution probabilistic function. In our example we use *IntPMF* which defines an integer distribution functions. Resource demands are specified within *InternalActions*. Resource demands can be specified using constants or again using probabilistic functions.



**Figure 2.6:** *System model graph (Adapted from Koch-Kemper (2015))*

In PCM we use *ExternalCallActions* in order to call external components, which can be distributed and deployed on external systems. Where a component is deployed depends on their packaging, which is defined in the *System Model*.

**System Model** The system model defines the deployment units of an application. Components are packaged in *ComponentStructures* marking the system boundaries. Internal and external interfaces are defined and linked. In a Java EE environment one such structure could be a *WAR*, *JAR*, or *EAR*. Our example in Figure 2.6 is very simple as it only consists of one structure containing both components. *ComponentA* provides one external interface. *ComponentB* is only used internally and cannot be accessed from components outside of the *ComponentStructure*. Each structure can be deployed on a separated server instances. Multiple structures per server are also possible.

**Resource Environment Model** The hardware resources are specified using the *Resource EnvironmentModel*. This model consist of *Resource Containers* representing (virtualized) server instances and *Linking Resources* that represent network links between different container. Each container is built with a set of resources like CPU and HDD as depicted in Figure 2.7. These resources have a *Scheduling Strategy* defining how multiple parallel tasks should be selected. Furthermore, the *Number of Replicas* defines the number of CPU cores while the *Processing Rate* defines the number of instructions per time unit (usually seconds but this can be configured) during simulation. In Figure 2.7 this rate is set to 1000, meaning that a resource demand of 315 as in Figure 2.5 is processed within 315 milliseconds (assuming that the time unit during simulation is seconds). This calculation assumes that no scheduling occurs and enough resources are available, otherwise the processing takes longer depending on the length of the queue waiting to access the resource.

Network is simulated depending on bandwidth and latency as specified in Figure 2.7. The latency marks the delay between sending and arrival of a packet between to resource containers. The bandwidth is only important when the number of bytes transfered through the link exceeds the available capacity (bandwidth) and thus queuing occurs. This model allows for simple network simulations. If two component structures are deployed on two separated servers, which are linked using a *Linking Resource*, network simulation is automatically conducted when components of these structures exchange data in form of *ExternalCallActions*.

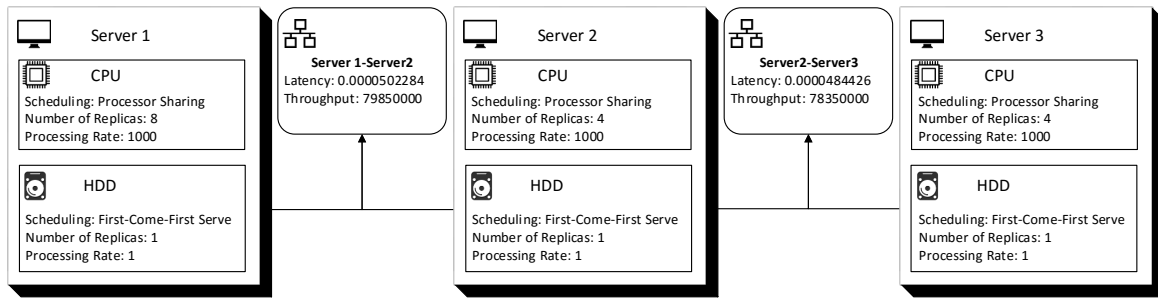


Figure 2.7: *Resource environment model example*

**Allocation Model** The deployment topology in the sense that the structures of the *System Model* are deployed on instances of the *Resource Environment Model* is defined using the *Allocation Model*. Each structure is linked to one or many server instances. If many instances are selected round-robin occurs when calling operations that are available on many server instances.

**Usage Model** The last sub-model of PCM is the *Usage Model*. User interaction with the system is specified creating instances of this sub-model. The meta-model provides two types of workloads: *Open Workloads* and *Closed Workloads*. *Closed Workloads* consist of a population (number of concurrent users) and a think time per user between two interactions. Each virtual user waits the length of time specified as think time between two interactions with the system. The workload is called closed as no new users access the system but the same virtual users access the system over and over again until the simulation ends. Similar specifications are used for load tests so that a workload from a load test can be easily translated into a workload specification in PCM.

*Open Workloads* on the other hand are characterized by an arrival rate. This probabilistic variables specified the mean time between two new users accessing the system. Here an infinite number of new users arrives until the simulation ends in contrast to *Closed Workloads* where the same users access the system again after completing a transaction. The total population is thus not fixed but depends on the capabilities of the application responding to the user interactions. The user is discarded and removed after he finished its transaction.

Our example in Figure 2.8 uses *Closed Workload*. The actual interaction with the application is defined similar to RDSEFF definition in a activity diagram like modeling style (Rumbaugh/Jacobson/Booch, 2004). Interactions with the system are defined using *SystemCallActions*, that can access all public available interfaces of the application. *Branch* and *Loop* elements allow for control-flow modeling similar to the elements in RDSEFFs.

Figure 2.8 presents a simple scenario. We use a *Closed Workload* with a population of 1500 users. The users are either accessing *operationA* or *operationB* of our *ComponentA*, which are both public. A *Branch* divides the users on a probabilistic basis. *OperationA* is called in 60% of all cases, while *OperationB* is used in the rest of the cases. After the system interaction each user waits for 9.7 seconds until a new interaction is started and either *operationA* or *operationB* is accessed by the user.



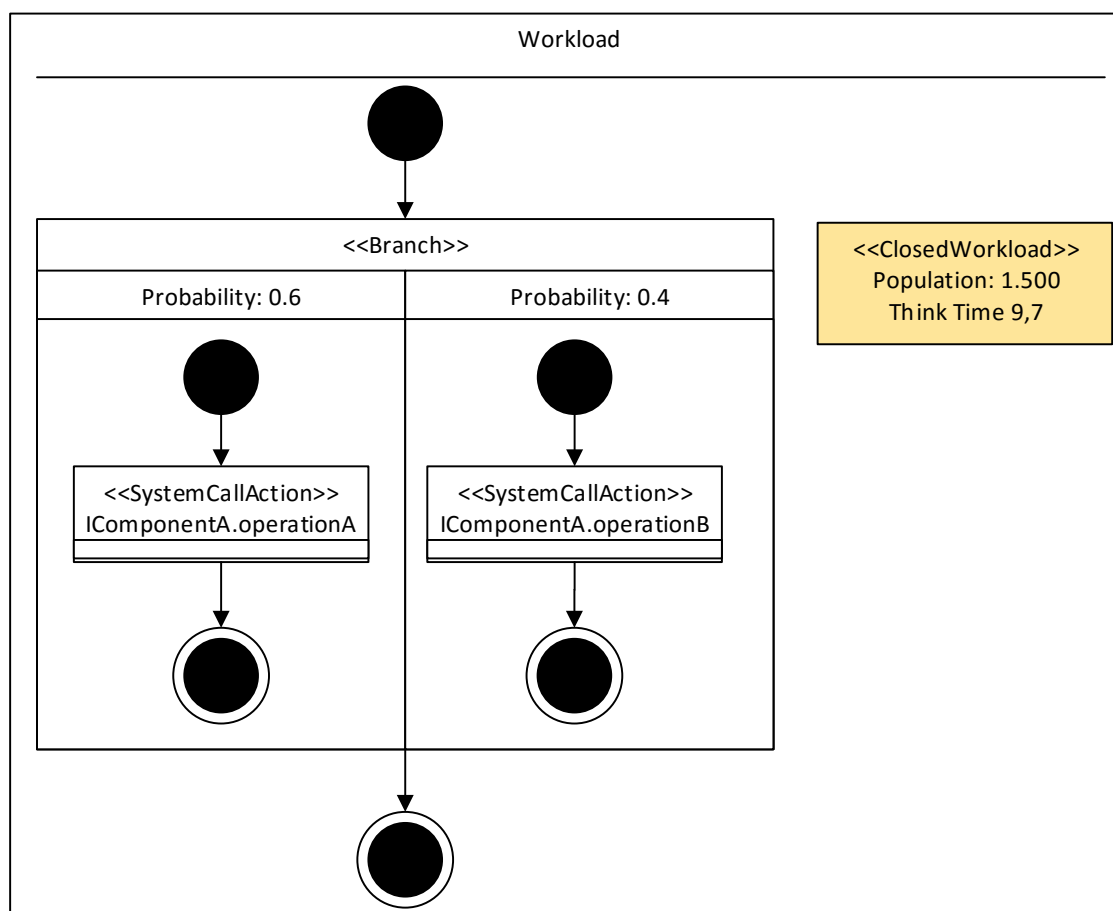


Figure 2.8: Usage model example

PCM plays an important role in our research. However, we usually do not create such models manually but use performance model generators. Our generators represent the EAs we use in the way they are composed of. This means, a component in PCM is mapped to a component in Java EE such as an Enterprise JavaBean (EJB). And an operation is mapped to a method in Java EE. This leads to rather complex models, with complex and interlaced control-flows. Manual comparison with the actual system is usually not possible and even graphical representations are hard to understand. Therefore, we use these models and simulate different workloads, changes to the system, or changes to the resource environment and compare the simulated performance metrics with actual load tests of the system in a real environment. The Palladio-bench, the tool set around PCM, offers several simulation engines and solvers<sup>11</sup>. Amongst these we use two simulation engines *SimuCom* and *EventSim* (Becker/Kozirolek/Reussner, 2009; Merkle/Henss, 2011).

**SimuCom** transforms PCM model instances into Java code that is executed. The code that is here created is the simulation code itself and has dependencies to other parts of the Palladio-bench like analysis frameworks (Becker/Kozirolek/Reussner, 2009). Even though, *SimuCom* is still the standard simulation engine, it has certain disadvantages. The model transformation is an unnecessary intermediate step that needs to be considered each time

<sup>11</sup>[http://www.palladio-simulator.com/science/palladio\\_component\\_model/model\\_solvers/](http://www.palladio-simulator.com/science/palladio_component_model/model_solvers/)

the meta-model is extended. Furthermore, the class method size limit of 64 kilobyte (kB) and the maximum number of methods limited to 65536 in the Oracle JVM prevent very complex simulations.

**EventSim** follows an alternative approach and works directly using PCM model instances. Instead of simulating processing it only simulates the time between two events (Merkle/Henss, 2011). This significantly speeds up the simulation. The direct interaction with the model reduces the complexity for introducing new meta-model elements. Early parts of our research are based on *SimuCom*. Our later work uses *EventSim* due to the above mentioned advantages.

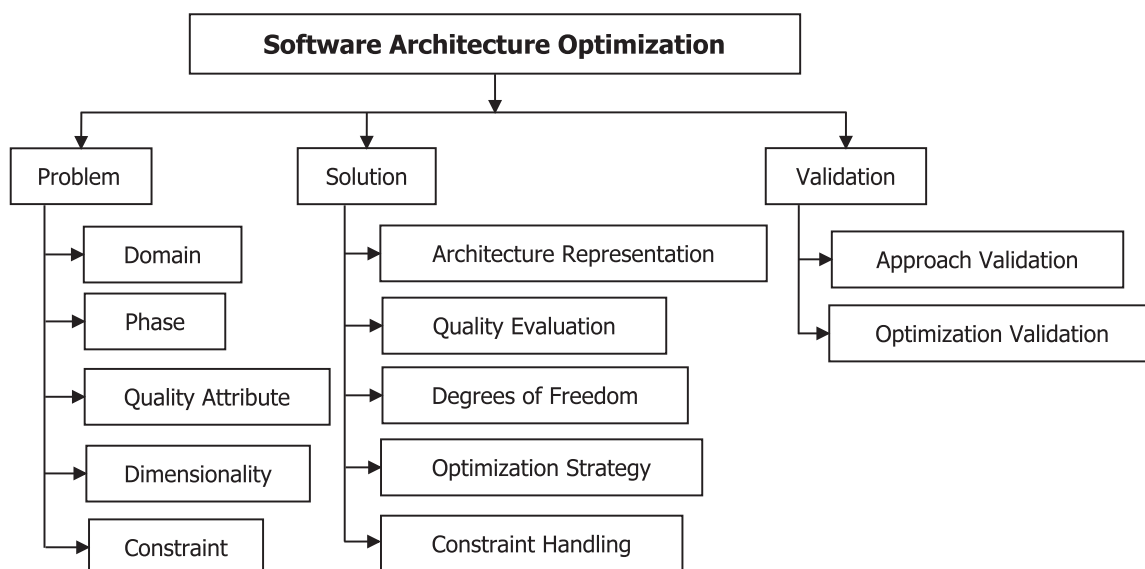
## 2.4 Automatic Architecture Optimization

Software Architecture Optimization has been the topic of many research artifacts. A comprehensive literature review as provided by Aleti et al. (2013). It structures almost 200 peer-reviewed papers that have been published between 1992 and 2011 in a simple and sound taxonomy depicted in Figure 2.9 (Aleti et al., 2013).

Software Architecture Optimization can be sorted into three main categories: Problem to be solved or optimized, the proposed solution, and the validation strategy. Aleti et al. (2013) found that most works focus on one or a few problems such as optimizing quality attributes or focus on a certain problem domain. Furthermore, the proposed solutions can be sorted into five major categories such as solutions that search for a adequate representation of the architecture or change it within certain degrees of freedom. Most validation approaches aim on validating the optimization, some only validate the approach Aleti et al. (2013). Using this taxonomy we sort our work in the problem category *Quality Attribute*, as we want to improve performance, costs, and resource utilization. Our solution is an *Optimization Strategy* within the *Degrees of Freedom* allocation and hardware/instance replication. We evaluate our approach by applying the selected *Optimization Strategy* to real systems.

Most works focus on optimizing or analyzing the architecture itself. Often within certain degrees of freedom, such as selection of components, instance replication, or service granularity (Aleti et al., 2013). However, most approaches either provide an approach to generate or discover a representation of the examined system or optimize an already created model (Koziolek/Koziolek/Reussner, 2011). Aleti et al. (2013) discovered the need for a holistic tool support that creates a model for optimization and validates the optimization on the changed system (Aleti et al., 2013). One of the major goals of our work is to connect performance model generation and performance model optimization to improve deployment topologies for distributed EAs. Thus, we try to close this research gap by providing such a holistic tool support.

A work that stands out, as it uses an earlier version of PCM and optimizes performance metrics and cost is PerOpteryx (Koziolek/Koziolek/Reussner, 2011). This approach was designed to search within certain degrees of freedom for a set of optimal solutions. It is,



**Figure 2.9:** *Software Architecture Optimization Approaches Taxonomy*  
Aleti et al. (2013)

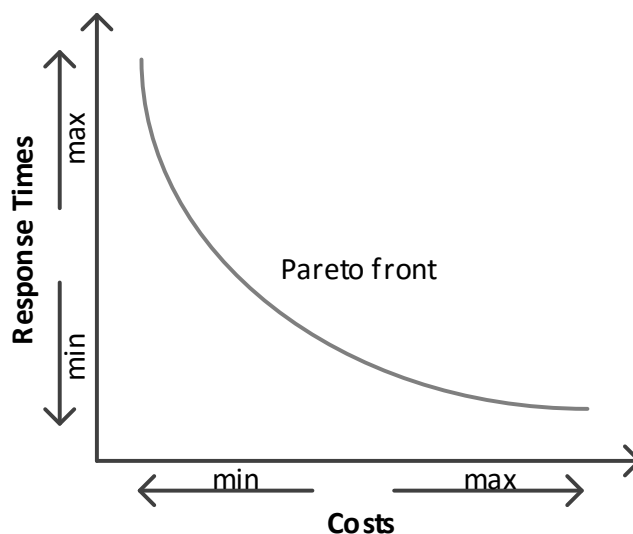
even model based, not possible to simulate or solve all possible solutions as their numbers grows exponentially with the number of decisions. Koziolok/Koziolok/Reussner (2011) use a solver that provides an analytical solution for PCM models for a single user and an approach based on multi-objective optimization. This optimization searches for Pareto-optimal solutions within the search space of possible solutions (Coello/Lamont/Van Veldhuisen, 2007). These solutions can be local optima, but even than the selected solutions are usually superior to other not optimized architectures.

### 2.4.1 Multi-objective Optimization Problems

Evolutionary algorithms allows for single and multi-objective optimizations. In general such an algorithm tries to maximize or minimize an evaluation function  $f(p)$  (Brunnert et al., 2015). In our case  $p$  is the result of a simulation process for a selected performance model. Many such evaluation functions are used for multi-objective optimizations, such as a function predicting the costs, and another predicting the response times. Both functions can be the result of a single simulation run.

Coello/Lamont/Van Veldhuisen (2007) describes multi-objective optimization as “*a vector of decision variables which satisfies constraints and optimizes a vector function whose elements represent the objective functions. These functions form a mathematical description of performance criteria which are usually in conflict with each other. Hence, the term “optimize” means finding such a solution which would give the values of all the objective functions acceptable to the decision maker.*”

Therefore, the goal of a multi-objective optimization is to optimize  $K$  objective functions in parallel. Each function can be either minimized or maximized allowing for combinations depending on the objective-goal. Mathematically speaking multi-objective optimization



**Figure 2.10:** *Exemplary multi-objective optimization problem with two objective functions that should be minimized (Adapted from Coello/Lamont/Van Veldhuisen (2007))*

tries to maximize a function  $F(x)$  consisting of several objective functions as depicted in Equation 2.2 (Coello/Lamont/Van Veldhuisen, 2007).

$$F(x) = [f_1(c), f_2(c), \dots, f_k(c)] \quad (2.2)$$

As the resulting solutions are in conflict, not only one solution exists (Coello/Lamont/Van Veldhuisen, 2007). The theory of Pareto-optimal solutions is applied to derive this set of solutions (Ehrgott, 2006). This theory defines a solution a Pareto-optimal, if no other solution exists that is better or equal considering all objective functions. The example in Figure 2.10 shows a Pareto-front along two objective functions, both with a minimization goal. The objective function of the y-axis is response times, and the objective function on the x-axis is costs. The Pareto-front shows the trade-off surface along which a decisions must be made to favor one optimization goal over another (e.g., minimize response times but accepting higher costs) or to choose a solution in the middle of the front.

## 2.4.2 Basic Concepts of Evolutionary Optimization

In order to derive this Pareto-front several optimization techniques exists amongst which we use evolutionary optimization. Evolutionary optimization can evaluate multiple solutions simultaneously, making it a good match for searching in a large solution space with many possible solutions.

The idea behind evolutionary optimization is a biological concept. Each solution candidate corresponds to a biological *genotype*. This would be a set of performance design options such as multiple allocation options or hardware replications. *Genotypes* are decoded in *phenotypes*, here a performance model instance conform to PCM. The design space options are represented as *chromosomes* of the *genotype*. In our case this allows us to define the allowed options of the deployment topology. Concrete options of these *chromosomes* are called *genes* and instance of the *genotypes* with different sets of *genes* are the *population*.

The process of altering, evaluating, and spawning new solution candidates is called a *generation*. In evolutionary optimization one such iteration consists of: reproduction, competition, and selection. Within such a *generation* parents (current generation) and children (follow-up generation) exist. Each generation spawns new solution candidates (children) out of the earlier evaluated candidates (parents).

**Reproduction Phase** In a first step new solutions are searched that can fulfill the objective. New candidates can be spawned either by *Recombination* or by *Mutation*:

1. **Recombination** mixes the genes of selected candidates and creates new ones out of it. This crossover function derives new solutions and should prevent running into local optima. This is conducted by mixing the genes of two parent solutions and creating children for the next generation.
2. **Mutation** randomly changes certain genes of one parent and creates new children out of it. This technique is usually used after recombination. If only mutations are used local optima can be reached instead of searching the whole space of possible solutions. We use a combination of both techniques in our work.

Afterwards, each candidate is evaluated in the **Competition Phase**. This means that the objective functions are executed and the resulting values are compared. In our work we simulate the performance model and derive performance metrics from this simulation in this phase.

The final phase of each *generation* is called the *Selection Phase*. The genes with the best fitness values are selected for recombination and mutation in the next phase. This ensures that the following generations produce equal or better results compared to the current generation.

This approach enables us to iterate through the design space of possible deployment topologies. The most time consuming part is the simulation in the *Selection Phase*. Therefore, we simulate a whole generation in parallel using a distributed simulation service. One generation is then evaluated as soon as the slowest simulation process of a generation has been conducted instead of after a series of simulations. This parallel evaluation allows us to use simulations instead of solvers, which enables us to simulate complex models accessed by multiple concurrent users.

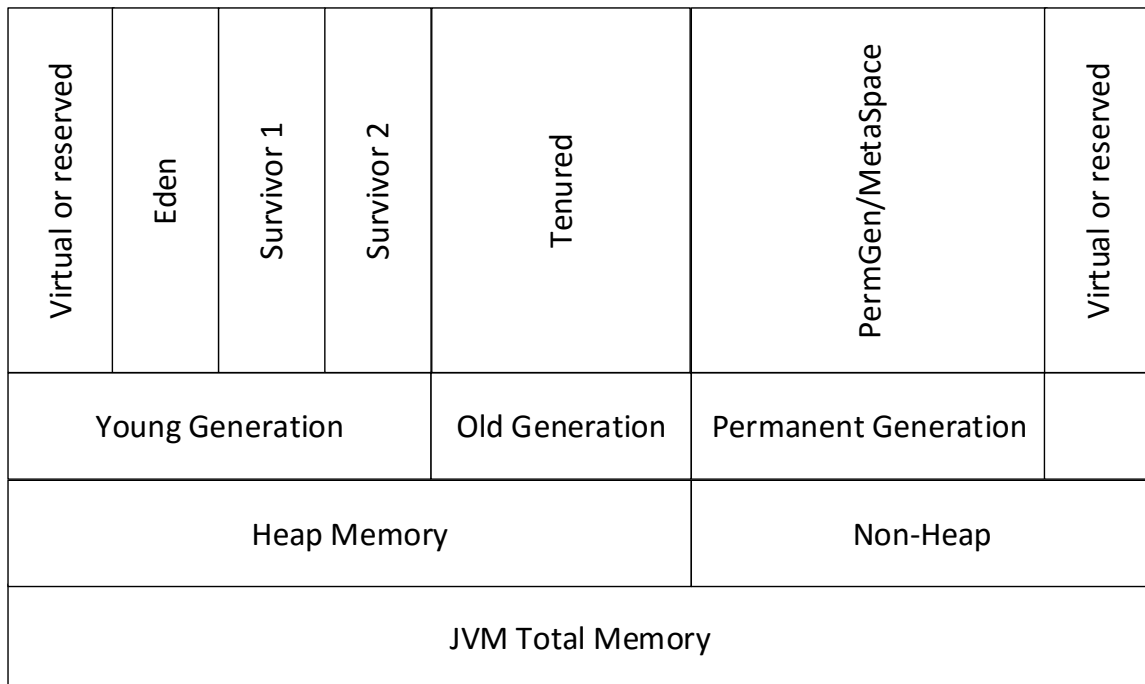
## 2.5 Memory Management

Memory of an application is usually divided into several major sections. One section is pretty static and contains the (compiled) program code (Forouzan, 2013). This memory section is relatively small and can be well estimated as the size of the required memory space is known right after the build process (Forouzan, 2013). Sometimes libraries are loaded dynamically during runtime, which leads to an increased demand in this section, but still this memory section stays easy manageable and can be estimated by the programmer or architect. In Java this part of the memory is divided into “Permanent Generation“ ( $\leq$  Java 7) or “MetaSpace“ ( $\geq$  Java 8), containing classes and methods loaded in the current JVM, and the “Code Cache“, that contains native code usually compiled out of directives that have been executed frequently (Schildt, 2014; Oransa, 2014). However, both parts contain program code just in different spaces (Schildt, 2014).

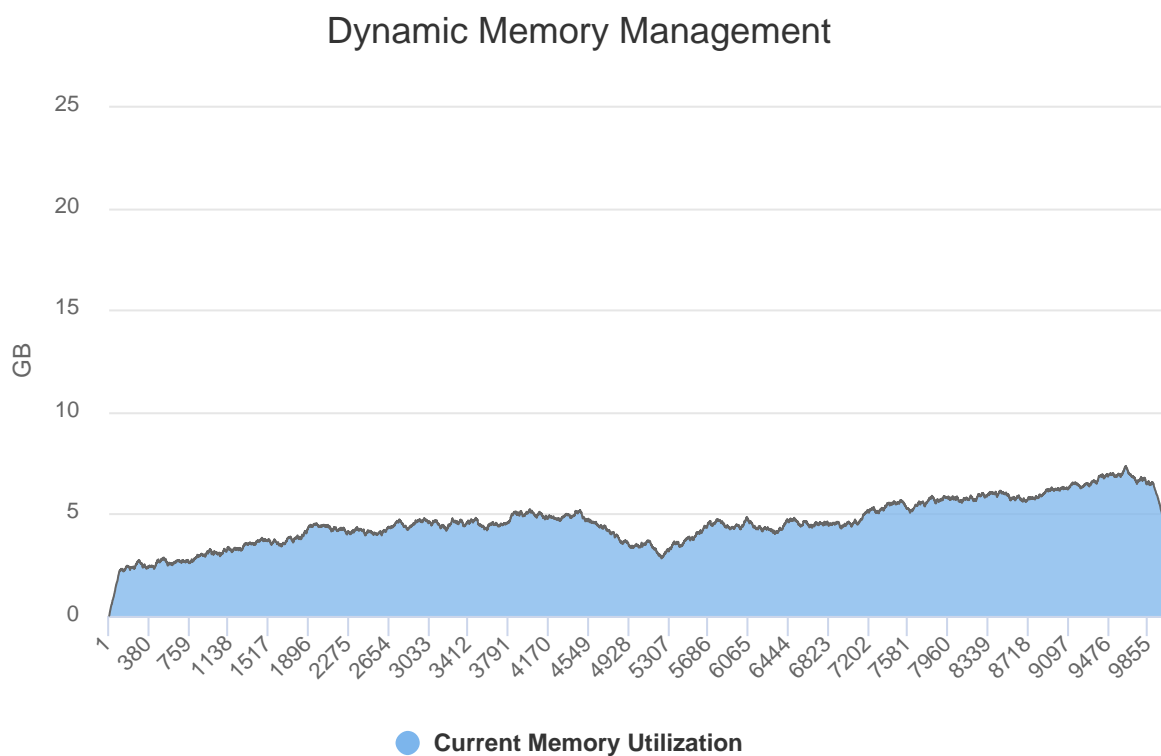
The second large section of memory of an application contains the data of the program (Forouzan, 2013). This section can be huge, depending on the amount of data an application uses, the interaction of the user with the system, and caching mechanisms that for example pre-load certain data from disk (Forouzan, 2013). The capacity of this section is hard to estimate and manage because of the dependencies on the software code itself, the workload, the runtime configuration, and the resource environment. This becomes even more complex in distributed systems as data is than shared, replicated, and synced amongst several instances. Our research focuses on this dynamic section of memory. In Java, this section is called Heap and again divided into smaller sub-sections: Eden, Survivor 1 & 2, and Tenured as depicted in Figure 2.11 (Schildt, 2014). The different sections correlate to the age of an object. The longer an object lives in Heap, the more likely it travels to the upper sections (Schildt, 2014).

Depending on the runtime environment, either dynamic memory management or automatic memory management is used. Dynamic memory management requires to explicitly load and unload objects meaning allocating and releasing memory based on the program code. Runtime environments for C++ or older versions of the iOS operating system (OS) use this memory model to manage memory. Mistakes in the program code or unexpected behavior of the user can lead to memory leaks due to missing object releases. This model is, good programming assumed, pretty efficient as memory is immediately released after an object is not needed anymore. Profiling an application using this memory model shows a lot of small spikes and a waveform as depicted in Figure 2.12.

Automatic memory management in contrast organizes the memory in the runtime. Creating new objects automatically allocates memory. Frequently checks on the references of these objects are conducted and the object and the memory it requires are released when no reference to the object remains. The algorithm class that checks and frees the memory is called Garbage Collection (Schildt, 2014). Figure 2.13 shows a typical memory profile of an application using automatic memory management. A typical pattern is the triangular form. The flank of each spike marks the execution of GC. The real demand is lower, but as the GC runs only after certain thresholds exceed, we see a delay in freeing memory compared to dynamic memory management (Schildt, 2014). In Java each GC

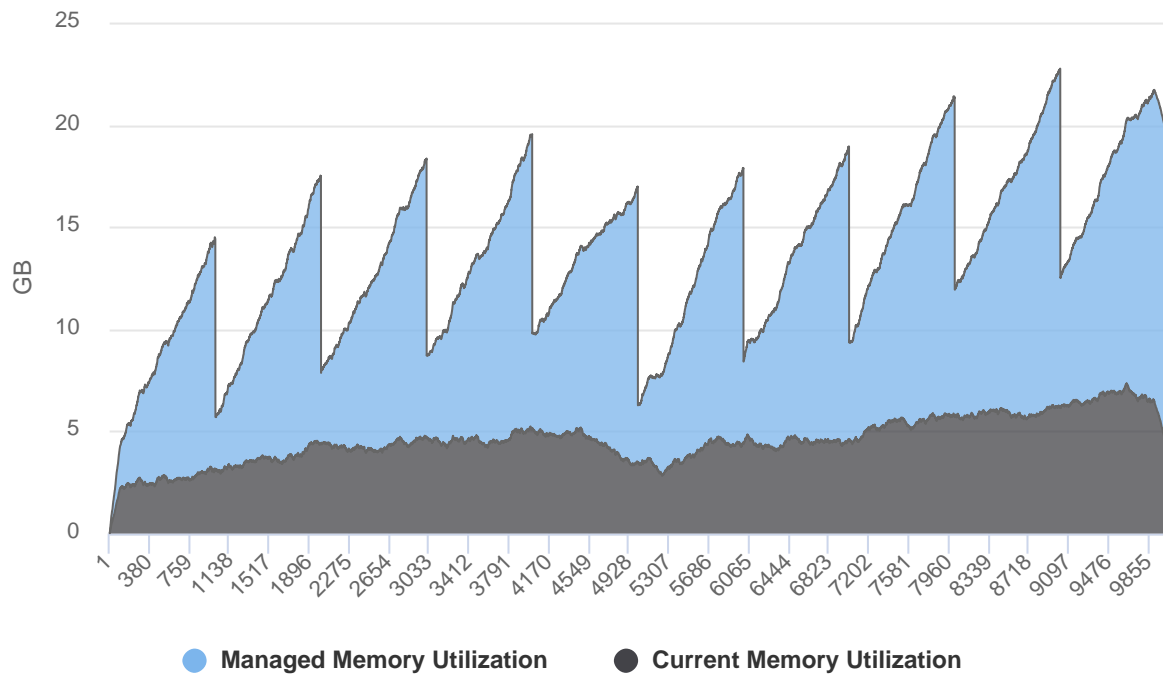


**Figure 2.11:** *Memory space organization in Java adapted from Honk (2014)*



**Figure 2.12:** *Memory trace of a system using Dynamic Memory Management*

## Automatic Memory Management



**Figure 2.13:** *Memory trace of a system using Automatic Memory Management*

run can promote objects to higher memory spaces (e.g., Survivor 1 to Survivor 2) (Schildt, 2014).

The more GC runs an object survives, meaning references to this object exists, the higher the memory space of this object (Schildt, 2014). Different GC check different spaces and based on different thresholds. Besides freeing memory, a GC run also consumes CPU when computing which objects are releasable. The number of spaces, the type of GCs, the thresholds, and the CPU consumption depend on the runtime (e.g, Java vs. .NET), the runtime version (e.g., Java 7 vs. Java 8), the configuration, and the application itself, more specific its object structure. We propose a model for simulating automatic memory management based on an approximation of these parameters. The same model, without GC behaviors is eligible for simulating dynamic memory management.



# Chapter 3

## Research Methodology

The following chapter presents the research methodology applied to conduct the research presented in this work. We present the design approach followed by the methods. Afterwards, we present the publications that build the core of this thesis, as well as related publications that were not part of the main research questions.

### 3.1 Research Design

The principal research design concept applied for this thesis is the design-science approach (Simon, 1996; Hevner et al., 2004). Applied on information systems design-science solves research problems by creating new and innovative artifacts (Denning, 1997; Hevner et al., 2004). We use the design-science approach applied on information systems introduced by Hevner et al. (2004) and Peffers et al. (2007) consisting of the following six steps:

1. **Identify problem and motivate:** The problem is identified, described, and a solution proposed. This includes reviewing existing research and identifying and address a research gap. Section 1.1 shows the result of this motivation process in our work.
2. **Define objectives of the solution:** The solution to the identified problem is cut down into goal(s). We derived three research question addressing individual goals in order to create a holistic solution address deployment topology optimization. These questions and corresponding goals are defined in Section 1.2.
3. **Design and development:** The goal(s) are addressed by designing and implementing artifact(s). These artifact(s) re-use, complement, enhance, or replace existing theories and knowledge.
4. **Demonstration:** In order to present the artifact(s) experiments, simulations, case studies, or similar approaches are applied. In this phase, the applicability of the research artifact(s) is demonstrated.

5. **Evaluation:** Validation of the research objectives is conducted in the evaluation phase. This requires to observe the artifact(s) and evaluate if they solve the identified problem. The process from design on till evaluation can be repeated until the research goal(s) is/are met.
6. **Communication:** After successful evaluation the artifact(s) and the results of the evaluation should be presented and made publicly. We present our artifacts and results in multiple publications outlined in Section 3.3.

## 3.2 Research Methods

We use the research methods *Simulation, Controlled Experiments, and Optimization* developed by Hevner et al. (2004). Furthermore, we added *Literature Reviews* to address the first two phases of our research design (Simon, 1996; Webster/Watson, 2002). Finally, we use *Prototyping* to design and implement our artifacts (Hevner et al., 2004).

**Literature Review** A *Literature Review* enables us to find and review existing research and begin the design-science process (Peppers et al., 2007; Webster/Watson, 2002). We conducted multiple of theses reviews systematically by the approach proposed by Webster/Watson (2002). Using this method, we identified relevant research publications and outlets as well as authors working in the same or related fields. Our research questions and methodology were designed based on the derived knowledge of these reviews. These reviews were conducted multiple times during the creation of our work and publications. Thus, the identified publications were updated and newest research artifacts were found on a regular basis. A final review was conducted while writing this thesis.

The found publications of our literature reviews are part of the related work sections in the publications of Part B and chapter 2. We used keywords that matched the topic of the individual work and research goals and used scholarly databases to find related publications. The following keywords were used for this search *architecture optimization, deployment optimization, cloud optimization model-based memory management, garbage collection simulation performance prediction, performance models, evolutionary algorithms, multi-objective optimization* and various combinations these keywords and modifications (e.g., singular/plural). We added forward and backward searches to the process to investigate work that was used by related work of our approaches and work based on that (Levy/Ellis, 2006). Backward search was conducted by analyzing the references of the initially found publications. Second and third level backward search was conducted when major publications related to our work were identified. Forward search was conducted by searching for publications that cited the analyzed publications. Forward and backward searches were combined in some cases and identified major publications that were not found just by keyword search. We rated all identified publications based on their titles, keywords, and abstracts.

The most important online scholarly databases that were considered during this thesis are:

1. ACM<sup>1</sup>
2. IEEE<sup>2</sup>
3. Springer<sup>3</sup>
4. Google Scholar<sup>4</sup>

The main focus of the literature review are publications in the field that are published in the following workshops, conference, and journals:

1. International Conference on Software Engineering (ICSE)
2. Journal of Systems and Software (JSS)
3. ACM Transactions on Internet Technology (ACM TOIT)
4. IEEE Transactions on Software Engineering (IEEE TSE)
5. International Journal on Software and Systems Modeling (SoSyM)
6. International Conference on Performance Engineering (ICPE)
7. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)
8. IEEE International Conference on Software Architecture (ICSA)
9. International Conference on the Quality of Software Architecture (QoSA)
10. International Conference on Autonomic Computing (ICAC)
11. European Performance Engineering Workshop (EPEW)
12. International Conference on Performance Evaluation Methodologies and Tools (ValueTools)
13. Business & Information Systems Engineering (BISE)
14. Performance Evaluation Journal
15. International Symposium on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS)
16. Symposium on Software Performance (SSP)
17. International Conference on Simulation Tools and Techniques (SIMUTOOLS)

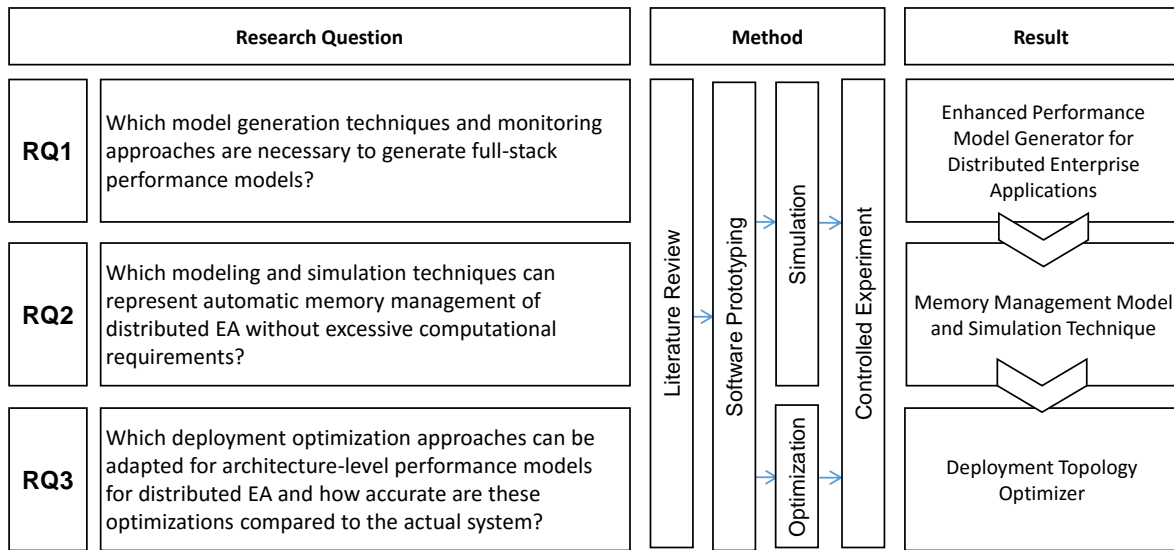
---

<sup>1</sup><http://dl.acm.org/>

<sup>2</sup><http://ieeexplore.ieee.org/Xplore/>

<sup>3</sup><http://www.springer.com/de/>

<sup>4</sup><https://scholar.google.de/>



**Figure 3.1:** *Memory trace of a system using Dynamic Memory Management*

**Prototyping** We developed multiple prototypes during this dissertation and our major artifacts are the result of applying this popular method (Alavi, 1984). This method allows to rapidly develop, adjust, and refine software artifacts and to experiment with and learn from these prototypes (Alavi, 1984). We created prototypes iteratively, evaluated intermediate results, refined, and adjusted them until our research goals were fulfilled.

**Simulation** Simulation is an experimental evaluation method that we use to validate our models (Hevner et al., 2004). The models are result of our prototypes and are validated by comparing the simulation results with observations from the real system. Furthermore, we test our models on robustness and scalability by adapting the model and conducting further simulations again validated by real system observations.

**Controlled Experiments** The developed prototypes were mainly evaluated using controlled experiments. They are “randomized experiments or quasi-experiments in which individuals or teams (the experimental units) conduct one or more software engineering tasks for the sake of comparing different populations, processes, methods, techniques, languages, or tools (the treatments)” (Sjoeberg et al., 2005). Our experiments were conducted in a reproducible setup using industry standard benchmarks in a lab and public cloud environment.

**Optimization** Optimization is another evaluation method designed to “demonstrate optimal properties of an artifact or to provide optimally bounds on artifact behavior” (Hevner et al., 2004). We optimize our models and demonstrate their validity using controlled experiments and simulations.

The research questions, applied methods, and the resulting artifacts of our research are depicted in Figure 3.1. The research results are based on each other and work as input for the following research questions. We structured the remainder of this thesis in the same order in Part B of this thesis.

No.	Authors	Title	Outlet
P1	<b>Willnecker</b> , Brunnert, Gottesheim, Krmar	Using Dynatrace Monitoring Data for Generating Performance Models of Java EE Applications	6th ACM/SPEC International Conference on Performance Engineering (ICPE) 2015
P2	<b>Willnecker</b> , Dlugi, Brunnert, Spinner, Kounev, Gottesheim, Krmar	Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques	European Workshop on Performance Engineering (EPEW) 2015
P3	<b>Willnecker</b> , Brunnert, Koch-Kemper, Krmar	Full-Stack Performance Model Evaluation using Probabilistic Garbage Collection Simulation	Symposium on Software Performance (SSP) 2015
P4	<b>Willnecker</b> , Krmar	Model-based Prediction of Automatic Memory Management and Garbage Collection Behavior	Simulation Modelling Practice and Theory (submitted) <sup>5</sup>
P5	<b>Willnecker</b> , Krmar	Optimization of Deployment Topologies for Distributed Enterprise Applications	12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA) 2016
P6	<b>Willnecker</b> , Vögele, Krmar	SiaaS: Simulation as a Service	Symposium on Software Performance (SSP) 2016
P7	<b>Willnecker</b> , Krmar	Multi-Objective Optimization of Deployment Topologies for Distributed Applications	ACM Transactions on Internet Technology (TOIT) 18.2 <sup>6</sup>

**Table 3.1:** *Publications embedded in this thesis*

### 3.3 Publications

Part B, the core of this thesis is composed of seven publications of the author (Table 3.1). Further publications that have been (co-)authored during the research are given in Table 3.2. Both tables include a publication number, the authors, the title and the outlet of each publication. Publications listed in Table 3.1 (**P1-P7**) are included in Part B with permission of the corresponding publishers.

In the following a brief summary for each embedded publication is given:

**P1** introduces an abstraction to a performance model generation approach that allows for embedding industry standard APM solutions. This work provides a solid interface for attaching further monitoring solutions. This work addresses core parts of the first research questions, which targets on enabling a performance model generation approach for distributed EAs. The example of Dynatrace was used in this work, as it provides monitoring solutions for a broad set of technologies and thus EAs.

<sup>5</sup>Impact factor, 2 years: 1.954, 5 years: 2.063

<sup>6</sup>Impact factor, 2 years: 0,705, 5 years: 1.118

**P2** compares resource demand measurements and resource demand estimations. Even though, a broad technological spectrum was enabled to produce models for distributed EAs by applying the results of **P1**, certain technologies do not provide a sound monitoring interface. Among those technologies, databases are the ones with the most vital part in EA architectures. We combined two approaches (measurements and estimations) to enable performance model generations even if no detailed monitoring interface is available. The results of these first two applications allowed us to answer research question 1 and to generate performance models for distributed EAs.

**P3** and **P4** address a gap in current performance evaluation research: automatic memory management. Performance model concentrated on CPU, HDD, and network resources. However, memory can have severe effects on (enterprise) applications if not managed and sized correctly. This can lead to application faults, high CPU utilization, long response times, and even terminations. The results of these two publications allows a PMG to provide holistic performance models that address all four major resource types. The accuracy of such models and especially of the memory management model and simulation are demonstrated in this work. These two publications answer the second research question.

**P5** provides a first deployment topology optimizer. This optimizer can search for good deployment topologies to either optimize response times or to increase resource utilization, which usually implies less costs in a cloud provisioning scenario. The optimizer considers all four major resources and demonstrates the optimization of an industry standard benchmark using the microservice paradigm and including its databases.

**P6** introduces a simulation service for the Palladio Component Model. This service can simulate multiple models in parallel and scale-out to many instances. This allows this service to speed up the optimization approach presented in **P5** and provides room for extension in order to use other simulation engines or performance meta-models.

**P7** shows an enhancement of **P5** and provides a multi-objective deployment topology optimizer. This optimizer searches for the Pareto-front along the optimization goals minimum response times, minimize cost, and optimal resource utilization. Furthermore, this work introduces a flexible cost model that covers on-premise, cloud, and hybrid deployment scenarios. The publication finalizes the answer to research question 3 and provides an approach to select and optimize the deployment topology of distributed EAs by using full-stack performance models.

Additionally to the embedded publications, the author contributes as (co-)author to several further publications (**P8-P17**) related to the topic of this dissertation. A complete list is provided in Table 3.2.

**P8** presents a distributed performance evaluation and comparison framework. The results of this work allow to easily collect, aggregate, compare, and present measurements and performance simulation data. The framework consist of collector interfaces, a con-

No.	Authors	Title	Outlet
P8	Kroß, <b>Willnecker</b> , Zwickl, Krcmar	PET: continuous performance evaluation tool	2nd International Workshop on Quality-Aware DevOps (QU-DOS) 2016
P9	<b>Willnecker</b>	Optimization of Component Allocations in Middleware Platforms using Performance Models	Software Engineering & Management 2015
P10	Brunnert, van Hoorn, <b>Willnecker</b> , Danciu, Hasselbring, Heger, Herbst, Jamshidi, Jung, von Kistowski, Koziolk, Kroß, Spinner, Vögele, Walter, Wert	Performance-oriented DevOps: A Research Agenda	Technical Report, SPEC Research Group – DevOps Performance Working Group 2015
P11	<b>Willnecker</b> , Krcmar	Towards Predicting Performance of GPU-dependent Applications on the Example of Machine Learning in Enterprise Applications	Symposium on Software Performance (SSP) 2017
P12	Düllmann, Heinrich, van Hoorn, Pitakra, Walter, <b>Willnecker</b>	CASPA: A Platform for Comparability of Architecture-based Software Performance Engineering Approaches	2017 IEEE International Conference on Software Architecture (ICSA 2017)
P13	<b>Willnecker</b> , Brunnert, Krcmar	Model-based Energy Consumption Prediction for Mobile Applications	EnviroInfo 2014
P14	<b>Willnecker</b> , Brunnert, Krcmar	Predicting Energy Consumption by Extending the Palladio Component Model	Symposium on Software Performance (SOSP) 2014
P15	Kindelsberger, <b>Willnecker</b> , Krcmar	Long-Term Power Demand Recording of Running Mobile Applications	IEEE 10th International Conference on Global Software Engineering Workshops (ICGSEW) 2015
P16	Danciu, Kroß, Brunnert, <b>Willnecker</b> , Vögele, Kapadia, Krcmar	Landscaping Performance Research at the ICPE and its Predecessors: A Systematic Literature Review	International Conference on Performance Engineering (ICPE) 2015
P17	Ardagna, Casale, van Hoorn, <b>Willnecker</b>	Proceedings of the 2nd International Workshop on Quality-Aware DevOps	2nd International Workshop on Quality-Aware DevOps (QU-DOS) 2016
P18	Di Nitto, Leitner, Ardagna, Casale, van Hoorn, <b>Willnecker</b>	Proceedings of the 3rd International Workshop on Quality-Aware DevOps	3rd International Workshop on Quality-Aware DevOps (QU-DOS) 2017

**Table 3.2:** *Further publications during the work on this dissertation*

trol interface for measurement collectors, and provides and integration for load testing frameworks such as Faban<sup>7</sup>.

**P9** introduced the general idea of this thesis. The work is an early proposal for this dissertation and was discussed with researchers at the Software Engineering Conference 2015.

**P10** outlines existing performance management activities and challenges to apply it in DevOps scenarios. The work presents current research in this area as well as future opportunities. The work concentrates on activities that require or enable DevOps, by means of integrated development (Dev) and operations (Ops) teams.

**P11** introduces an approach to integrate graphics processing for EAs into performance research, specifically into performance models and corresponding simulation engines.

**P12** presents an inter-operable platform for performance research. This platform allows researcher to use and integrate several APM and Software Performance Engineering (SPE) tools. The communication is based on open standard like OPEN.xtrace and uses container and container orchestrator to run experiments (Okanović et al., 2016). The platform aims on creating an performance-aware DevOps tool chain for researchers and practitioners in a second step.

In **P13** and **P14** introduce an approach to use model-based performance evaluation to predict the power consumption of mobile-phones. Established performance simulation techniques were adapted and enhanced to specify energy demands. **P16** enhances this model to provide long-term predictions and evaluates this approach using multiple sport tracker applications.

**P16** is a systematic literature review of papers published in the proceedings of the ICPE and its predecessors (Danciu et al., 2015). The work focuses on one of the main conference in the area of performance evaluation. This work analyzes topics, evaluation methods, and types of systems that are presented at this conference over time and presents relationships to geographical and organizational information.

**P17** and **P18** represents the proceedings of the QUDOS Workshop 2016 and 2017 co-organized by the author that addresses the challenge of integrating quality assurance techniques, like performance, into DevOps scenarios<sup>8</sup>.

---

<sup>7</sup><http://faban.org/>

<sup>8</sup><http://qudos2017.fortiss.org>



## Part B

# Chapter 4

## Using Dynatrace Monitoring Data for Generating Performance Models of Java EE Applications

Authors	Willnecker, Felix <sup>1</sup> (willnecker@fortiss.org) Brunnert, Andreas <sup>1</sup> (brunnert@fortiss.org) Gottesheim, Wolfgang <sup>2</sup> (wolfgang.gottesheim@dynatrace.com) Krcmar, Helmut <sup>3</sup> (krcmar@in.tum.de)  <sup>1</sup> fortiss GmbH, Guerickestraße 25, 80805 München, Germany <sup>2</sup> Compuware Austria GmbH, Freistädter Str. 313, 4040 Linz, Austria <sup>3</sup> Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE) 2015
Status	Accepted
Keywords	Load Testing; Performance Evaluation; Application Performance Management
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

**Table 4.1:** *Bibliographic details for P1*

**Abstract** Performance models assist capacity management and planning for large-scale enterprise applications by predicting their performance for different workloads and hardware environments. Manually creating these models often outweighs their benefits. Automatic performance model generators have been introduced to facilitate the model creation. These generators often use custom monitoring solutions to generate the required input data for the model creation. In contrast, standardized application performance management (APM) solutions are used in industry to control performance metrics for productive systems. This work presents the integration of industry standard APM solutions with a performance model generation framework. We apply the integration concepts using the

APM solution Dynatrace and a performance model generation framework for Palladio Component Models (PCM).

## 4.1 Introduction

Performance of large-scale enterprise applications (EA) is a critical quality requirement (Brunnert/Vögele et al, 2014). Application providers and data center operators tend to over-provision capacity to ensure that performance goals are met (Pawlish/Varde/Robila, 2012). This is due to a lack of tool support for predicting the required capacity of a software system for expected workloads (Brunnert/Wischer/Krcmar, 2014). Performance models and corresponding model solvers or simulation engines can enhance current capacity estimations and therefore increase the utilization of hardware and reduce costs for application operations (Brosig/Kounev/Krogmann, 2009; Grinshpan, 2012).

The effort of manually creating such performance models often outweighs their benefits (Kounev, 2005). Automatic model generators have been introduced to reduce this effort (Brunnert/Vögele/Krcmar, 2013; Brosig/Kounev/Krogmann, 2009). These approaches rely on monitoring data from running systems to extract the performance models. Such generated models can be used as input for a simulation engine or an analytical solver to predict the resource utilization, throughput and response time for different workloads and hardware environments.

Monitoring data for the generation of performance models is gathered by either custom solutions or tools from the scientific community (Brunnert/Vögele/Krcmar, 2013; van Hoorn/Waller/Hasselbring, 2012). On the other hand, monitoring of large-scale EAs are state of the art technology in practice (Kowall/Cappelli, 2012). Companies use the gathered monitoring data to detect and resolve performance problems in productive environments (Koziolek, 2010). This work presents an extension of our existing performance model generation framework to work with industry standard Application Performance Management (APM) solutions. We extend the Performance Management Work Tools (PMWT<sup>1</sup>) model generator to create Palladio Component Models (PCM) based on data collected by the Dynatrace<sup>2</sup> APM solution (Becker/Koziolek/Reussner, 2009; Reussner/Becker et al., 2009; Greifeneder, 2011; Brunnert/Vögele/Krcmar, 2013).

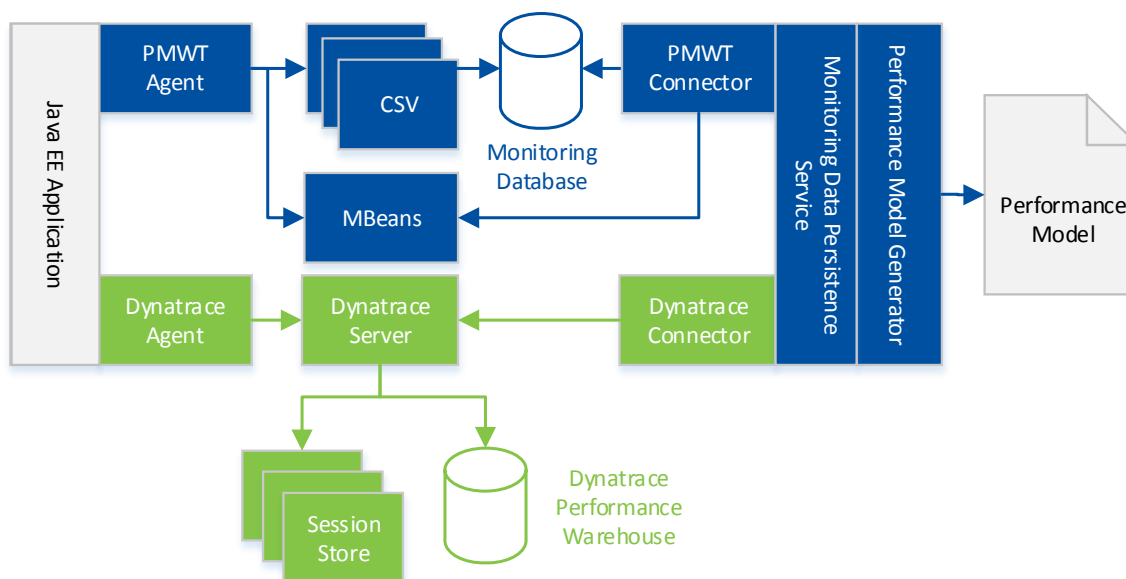
## 4.2 Automatic Performance Model Generation Framework

In order to use the Dynatrace APM solution we extend the model generation framework presented in (Brunnert/Vögele/Krcmar, 2013) and shown in figure 4.1. This framework uses a custom agent that collects the monitoring data from a running Java EE application.

---

<sup>1</sup><http://pmw.fortiss.org/>

<sup>2</sup><http://www.dynatrace.com/>



**Figure 4.1:** *PMWT Performance Model Generation Framework*

The monitoring data is then processed and aggregated either as comma-separated value (CSV) files and imported into a database or as Managed Beans (MBeans). The aggregated data is input for the model generation. The result is a performance model compliant with the PCM meta-model. The extension proposed in this work allows to use data extracted by standard monitoring frameworks exemplified by a Dynatrace agent for the purpose of performance model generation. This agent is attached using runtime options without changes to the instrumented application system's source code. The agent forwards collected data to the Dynatrace server, where detailed traces about method calls and error states are stored in session files for further analysis. Performance metrics derived from these traces are stored in a performance warehouse, and these metrics are typically used by operation engineers as data provider for monitoring dashboards. We extract data from both sources using an extension to our model generation framework called Dynatrace connector.

The Dynatrace connector leverages the representational state transfer (REST) interface of the Dynatrace server to extract detailed monitoring data. This REST interfaces provides, among others, call traces for instrumented operations including their resource demands. The Dynatrace connector is an extension of the monitoring data persistence service that is used by the model generator to access data from different sources. The model generator creates a performance model conforming to the PCM meta-model based on the traces and their average resource demands. The resulting models can then be used for the existing simulation engines and analytical solvers that exist for PCM models (Reussner/Becker et al., 2009).

## 4.3 Conclusion & Future Work

This work proposed an integration of an industry APM solution with a performance model generation framework. Different input formats and levels of granularity can be processed. The extension shows that the generator and its interface are generally applicable and other APM solutions as generator input are possible. As the Dynatrace solution is in widespread use, the monitoring technology is tested more intensive than custom solutions and in varied operation environments. The generated model can be used to simulate different workloads and therefore enhance the Dynatrace solution with capacity planning capabilities.

As a next step we will further extend an existing prototype for the integration and evaluate it in a case study comparing the results using our PMWT agent and the Dynatrace agent. For the evaluation, we will extract models from a running SPECjEnterprise2010 instance using the two existing data collection approaches. Afterwards, the resulting models are used to predict the utilization, throughput and response time for an increased number of users. The prediction results are compared with measurement for similar workloads on the SPECjEnterprise2010 instance.

# Chapter 5

## Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques

Authors	Willnecker, Felix <sup>1</sup> (willnecker@fortiss.org) Dlugi, Markus <sup>1</sup> (dlugi@fortiss.org) Brunnert, Andreas <sup>1</sup> (brunnert@fortiss.org) Spinner, Simon <sup>2</sup> (simon.spinner@uni-wuerzburg.de) Kounev, Samuel <sup>2</sup> (samuel.kounev@uni-wuerzburg.de) Gottesheim, Wolfgang <sup>3</sup> (wolfgang.gottesheim@dynatrace.com) Krcmar, Helmut <sup>4</sup> (krcmar@in.tum.de)  <sup>1</sup> fortiss GmbH, Guerickestraße 25, 80805 München, Germany <sup>2</sup> Universität Würzburg, Am Hubland, 97074 Würzburg, Germany <sup>3</sup> Dynatrace Austria GmbH, Freistädter Str. 13, 4040 Linz, Austria <sup>4</sup> Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Proceedings of the European Workshop on Performance Engineering (EPEW) 2015
Status	Accepted
Keywords	Performance Model Generation, Resource Demand Measurements, Resource Demand Estimations
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, paper writing, paper editing

**Table 5.1:** *Bibliographic details for P2*

**Abstract** Resource demands are a core aspect of performance models. They describe how an operation utilizes a resource and therefore influence the systems performance metrics: response time, resource utilization and throughput. Such demands can be determined by two extraction classes: direct measurement or demand estimation. Selecting the best suited technique depends on available tools, acceptable measurement overhead and the level of granularity necessary for the performance model. This work compares two direct measurement techniques and an adaptive estimation technique based on multiple statis-

tical approaches to evaluate strengths and weaknesses of each technique. We conduct a series of experiments using the SPECjEnterprise2010 industry benchmark and an automatic performance model generator for architecture-level performance models based on the Palladio Component Model. To compare the techniques we conduct two experiments with different levels of granularity on a standalone system, followed by one experiment using a distributed SPECjEnterprise2010 deployment combining both extraction classes for generating a full-stack performance model.

## 5.1 Introduction

Performance models can be used to predict the performance of application systems. Resource demands are an important parameter of such performance models. They describe how an operation utilizes the available resources. A busy resource increases the time an operation needs to execute, therefore increasing the response time of the operation and ultimately the time for the user accessing the system. When performance models are applied for capacity management, such information is essential as the available hardware must be sized according to the demand of the operations for a certain workload. Demands can be extracted from different sources. Expert guesses are used, especially when no running application artifact is available, to forecast the application's performance behavior. If running artifacts are available (e.g., in a test environment), measurement and estimation techniques can be applied. This work compares two direct measurement techniques and an adaptive estimation technique based on multiple statistical approaches and compares strengths and weaknesses of each technique.

Manually creating performance models often outweighs their benefits (Brunnert/Vögele/Krcmar, 2013). Therefore, automatic performance model generator (PMG) frameworks for running applications have been introduced in the scientific community (Brosig/Kounev/Krogmann, 2009; Brunnert/Vögele/Krcmar, 2013). Such PMGs create performance models, which include the software architecture, control flow and the resource demand of the application. These PMGs use either direct measurements by instrumenting the operations that are executed or resource demand estimations calculated from coarse-grained measurement data like total resource utilization and response time per transaction invocation.

Applying direct measurements requires to alter the installation of the system that is instrumented by applying an agent that intercepts invocations. This allows for extracting the software architecture and control flow, but causes overhead on the system running for every instrumented operation that is invoked (Brunnert/Neubig/Krcmar, 2014). Furthermore, such measurements require that for each instrumented technology and resource type, a dedicated measurement approach must be available. A number of industry solutions for direct measurements are already available and have been integrated into such a PMG previously (Willnecker/Brunnert et al., 2015a).

As an alternative to direct measurements, resource demand estimation techniques can approximate the demand of a resource from coarse-grained monitoring data like Central

Processing Unit (CPU) utilization of a system and response time of a transaction. Such data can be collected for a wide range of systems and technologies and requires no in-depth measurement of the application's technology stack. This coarse-grained monitoring data causes less overhead, produces less data to collect, and to process. However extracting the control flow of an application is not possible with such an approach.

The Library for Resource Demand Estimation (LibReDE)<sup>1</sup> provides different resource demand estimation approaches (Spinner/Casale et al., 2014). In order to do the estimations, LibReDE requires information about the resource utilization as well as about the response times of an operation or transaction during the same time frame. This work integrates LibReDE with the PMG introduced by Brunnert et al. (Brunnert/Vögele/Krcmar, 2013) in order to be able to generate models based on direct resource demand measurements or estimations. This integration allows to compare the direct measurement and estimation approaches and to determine strengths and weaknesses for extracting resource demands using the SPECjEnterprise2010<sup>2</sup> industry benchmark as representative enterprise application for the evaluation.

We compare these two extraction classes for resource demands in a series of experiments evaluating the accuracy of automatically generated performance models in terms of CPU utilization and response times. Therefore, the main contributions of this work are as follows:

- (i) An integration of resource demand estimation in a PMG.
- (ii) A comparison of the accuracy of two direct measurement techniques with the most common resource demand estimation approaches used in practice.
- (iii) An evaluation of an integrated PMG, utilizing the benefits of direct measurement and estimation techniques.

This work begins with an introduction to the performance model generation workflow followed by introducing measurement technologies. We continue with an introduction to LibReDE and the approaches used to estimate resource demands including the selection of the most accurate estimation approach for meaningful resource demands. The experiment for comparing all three approaches is described and evaluated, followed by a hybrid setup where a combination of direct measurements and resource demand estimations is used. The work closes with related work, followed by the conclusion and future work section.

## 5.2 Extracting Resource Demands

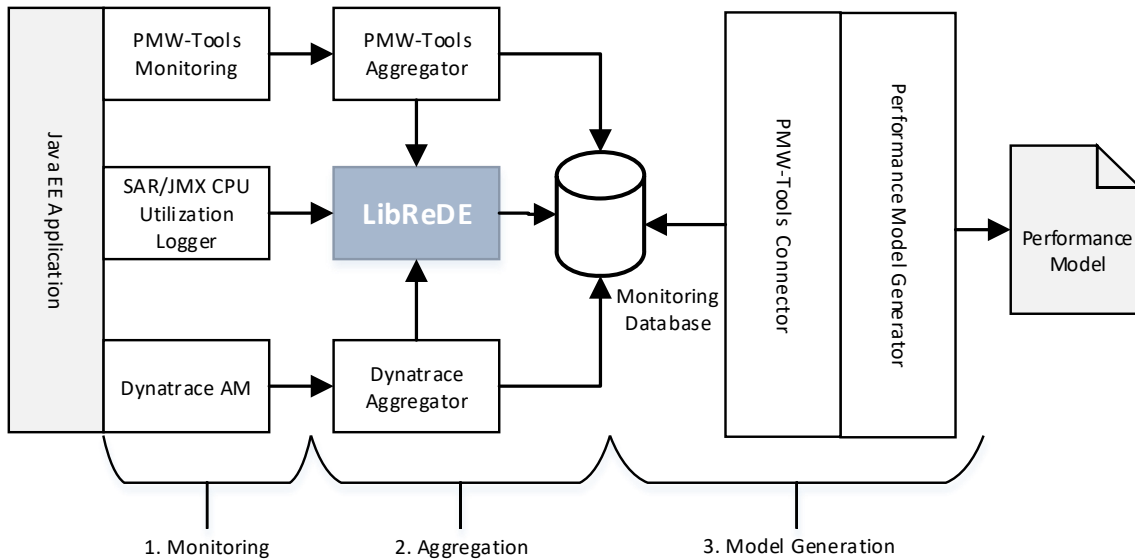
In order to support resource demand measurement and estimation approaches, we extend the previously introduced Performance Management Work (PMW)-Tools' automatic

---

<sup>1</sup><http://se.informatik.uni-wuerzburg.de/tools/librede/>

<sup>2</sup>SPECjEnterprise is a trademark of the SStandard 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/osg/Enterprise2010>.





**Figure 5.1:** *Performance model generator framework (adapted from Brunnert/Neubig/Krcmar (2014); Willnecker/Brunnert et al. (2015a))*

PMG with LibReDE (Brunnert/Vögele/Krcmar, 2013), (Spinner/Casale et al., 2014). Generating a performance model is divided into three separate steps depicted in Figure 5.1. First monitoring data is gathered. This monitoring data is, in a second step, aggregated per operation and stored in a monitoring database (DB). The last step is the actual model generation, which uses the aggregated data and generates an architecture-level performance model based on the Palladio Component Model (PCM) (Becker/Koziolek/Reussner, 2009).

The PMG supports data from different data sources:

- (i) PMW-Tools monitoring, a monitoring solution for Java Enterprise Edition (EE) applications to measure CPU, memory, and network demands and response times of Java EE components and its operations (Brunnert/Krcmar, 2017; Brunnert/Vögele/Krcmar, 2013).
- (ii) Dynatrace<sup>3</sup> Application Monitoring (AM), an industry monitoring solution for Java, .NET, PHP and other technologies (Willnecker/Brunnert et al., 2015a).
- (iii) System Activity Reporter (SAR), an Unix/Linux based tool to display various system loads like CPU utilization.
- (iv) Java Management Extensions (JMX) Logger, a command line tool that reads CPU utilization values from Java Virtual Machines (JVMs) using the JMX interface.

The first two data sources are able to collect direct measurement data, but also response times for estimation techniques. The demand estimation is realized using LibReDE (Spinner/Casale et al., 2014). This library uses response times of an operation or transaction and utilization of a resource, collected by one of the last two data sources, to estimate the resource demands of an operation (Spinner/Casale et al., 2014).

<sup>3</sup><http://www.dynatrace.com>

### 5.2.1 Performance Management Work - Tools Monitoring

PMW-Tools monitoring provides a Servlet Filter, an Enterprise JavaBean (EJB) Interceptor, a SOAP-Handler and a Java Database Connectivity (JDBC)-Wrapper for Java EE applications (Brunnert/Krcmar, 2017; Brunnert/Vögele/Krcmar, 2013). The aforementioned technologies allow to collect CPU time, heap allocation and network demand on the level of single operation invocations (Brunnert/Krcmar, 2017; Brunnert/Neubig/Krcmar, 2014; Brunnert/Vögele/Krcmar, 2013). Furthermore, the PMW-Tools monitoring allows to collect information about the transaction control flow and about an application architecture on the level of components and their operations. All public operations within the instrumented system are extracted and combined to one transaction. The PMW-Tools monitoring agent is able to measure the response time of an operation. The start and end time of each operation invocation is measured. Subinvocations are removed from this time interval, so the actual response time of one operation invocation is calculated.

### 5.2.2 Dynatrace Application Monitoring

The Dynatrace AM solution allows for measurements on different levels of granularity. This ranges from measuring the response time on the system entry point level, through fine-grained measurements per operation invocation. Dynatrace AM uses, depending on the host system, various timers that measure the CPU utilization in different time intervals (Dynatrace, 2015). It furthermore traces a transaction throughout the instrumented system and can therefore determine the control flow as the PMW-Tools monitoring does (Willnecker/Brunnert et al., 2015a). The Representational State Transfer (REST) interface of this solution provides, among other metrics, the ability to access CPU time and response times of the instrumented operations. Thus, this approach, as well as the PMW-Tools monitoring approach can be used for direct measurements and estimation techniques.

### 5.2.3 Library for Resource Demand Estimation

#### 5.2.3.1 Demand estimation approaches

While the monitoring tools described in Section 5.2.1 and Section 5.2.2 are able to directly measure the CPU time per operation invocation, their usage is infeasible in certain situations, e.g., when using third-party or legacy applications that cannot provide the required instrumentation. For other scenarios, the costs for fine-grained instrumentation can be considered too high. Therefore, different statistical approaches have been proposed in the literature to estimate resource demands for individual operations based on aggregated measurements such as average response time or CPU utilization. These aggregated measurements are often collected by default in applications (e.g., in access log files) and in the operating system (OS). Therefore, resource demand estimation techniques can be applied in many situations where the usage of direct measurements is prohibitive.

LibReDE is a Java library providing different ready-to-use implementations of statistical approaches for resource demand estimation (Spinner/Casale et al., 2014). The library currently comes with implementations of six commonly used approaches: response time approximation (Brosig/Kounev/Krogmann, 2009), service demand law (Brosig/Kounev/Krogmann, 2009), linear regression (Rolia/Vetland, 1995), two variants of a Kalman filter (Wang et al., 2012; Zheng/Woodside/Litoiu, 2008) and an optimization-based approach (Menascé, 2008). Previous work (Spinner, 2011) showed that the accuracy of the individual techniques strongly depends on the characteristics of the observations and the modeled system resulting in significant differences in the estimates. In order to evaluate the accuracy of the estimated resource demands, LibReDE supports the evaluation of the results using  $k$ -fold cross-validation: the input data is randomly partitioned into  $k$  equally large subsets and the estimation is repeated  $k$  times, each time using a different one of the  $k$  subsets as validation set and the others as training set. As the actual values of the resource demands are unknown, the estimation error is evaluated using the observed utilization  $U_{act}$  and the observed response times  $R_{act,r}$  of operation  $r$ . The observed values are compared to the calculated ones,  $U_{calc}$  and  $R_{calc,r}$ , which are obtained using equations from operational analysis of queuing networks. Using the estimated resource demands,  $U_{calc}$  is determined based on the Utilization Law (Harchol-Balter, 2013, Chap. 6):

$$U_{calc}(\lambda) = \frac{1}{p} \sum_{r=1}^n \lambda^r D^r \quad (5.1) \quad R_{calc}^r(\lambda) = D^r \left(1 + \frac{P_Q}{1 - U_{calc}(\lambda)}\right). \quad (5.2)$$

Assuming a M/M/k/PS queue for Equation 5.2 (Harchol-Balter, 2013, Chap. 14):  $n$  is the number of operations,  $D_r$  is the estimated resource demand of operation  $r$ ,  $\lambda = (\lambda_1, \dots, \lambda_n)$  is a vector of arrival rates,  $p$  is the number of processor cores and  $P_Q$  is the probability that an arrival finds all servers busy (calculated using the Erlang-C formula (Harchol-Balter, 2013, Chap. 14)).

The mean relative errors  $E_{util}$  for the utilization and  $E_{rt,r}$  are then determined on the validation set  $V = \{(\lambda_1^{(i)}, \dots, \lambda_n^{(i)}, R_{act,1}^{(i)}, \dots, R_{act,n}^{(i)}, U_{act}^{(i)}) : i = 1 \dots m\}$ :

$$e_{util} = \frac{1}{m} \sum_{i=1}^m \frac{|U_{act}^{(i)} - U_{calc}(\lambda^{(i)})|}{U_{act}^{(i)}} \quad (5.3) \quad e_{rt}^r = \frac{1}{m} \sum_{i=1}^m \frac{|R_{act}^{(r,i)} - R_{calc}^r(\lambda^{(i)})|}{R_{act}^{(r,i)}} \quad (5.4)$$

The relative errors are calculated for each of the  $k$  validation sets and the result of the cross-validation is the mean relative error over all validation sets. Based on the relative errors, the PMG dynamically chooses an approach as described in the next section.

### 5.2.3.2 Estimation approach selection

Selecting the right estimation approach for LibReDE makes a huge difference (in our experiments we observed differences in the range of 6% to 6000% relative response time error). Each approach has strengths and weaknesses depending on the application in place (Spinner, 2011; Spinner/Casale et al., 2014).

$$\begin{bmatrix} e_{util}^{(1)} \\ \vdots \\ e_{util}^{(i)} \\ \vdots \\ e_{util}^{(m)} \end{bmatrix} + \left( \begin{bmatrix} e_{rt}^{(1,1)} & \dots & e_{rt}^{(1,j)} & \dots & e_{rt}^{(1,n)} \\ \vdots & & \vdots & & \vdots \\ e_{rt}^{(i,1)} & \dots & e_{rt}^{(i,j)} & \dots & e_{rt}^{(i,n)} \\ \vdots & & \vdots & & \vdots \\ e_{rt}^{(m,1)} & \dots & e_{rt}^{(m,j)} & \dots & e_{rt}^{(m,n)} \end{bmatrix} \times \begin{bmatrix} \lambda^{(1)} & \dots & 0 & \dots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \dots & \lambda^{(i)} & \dots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \dots & 0 & \dots & \lambda^{(n)} \end{bmatrix} \right) \times \begin{bmatrix} 1 \\ \vdots \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (5.5)$$

We are looking for the approach that calculates the most accurate resource demands, therefore we use both validators and select the one with the lowest relative error when combining both validation results provided by LibReDE. The utilization law validator provides a vector  $E_{util}$ , as we only use one resource, with the length of  $m$ , where  $m$  is the number of estimation approaches used. Each row in this vector contains the relative utilization error of one approach. The response time validator provides a  $m \times n$  matrix  $E_{rt}$ , where  $m$  is the number of estimation approaches used and  $n$  the number of operations to estimate resource demands for. Each row  $i$  contains all relative response time errors of one approach and each column  $j$  contains the relative response time error of one operation. Therefore, the value at index  $i,j$  is the relative response time error of operation  $j$  using approach  $i$ .

Some operations might get a small amount of calls, misleading the approach selection when just selecting the approach with the smallest relative error. We weight the relative error of each operation according to the arrival rates of the input data as the number of values used for the estimation varies due to different workload on each operation. We therefore multiply the arrival rates matrix  $\lambda$  with the relative response time error matrix  $E_{rt}$ . The result is a weighted matrix that considers the operation call probability. To select the best suited approach we need to reduce this matrix to a vector, where each value contains a meaningful relative error for one approach considering all operations. We calculate the sum over each row of the matrix resulting in a relative response time error vector. Both vectors, containing either the response times or the CPU utilization error, are added up as shown in Equation 5.5.

We finally select the approach with the minimum total error in the resulting vector. The resource demands  $D_r$  of this approach are stored in the monitoring DB of the PMG. The model generation then uses these resource demands for building an architecture-level performance model.

## 5.3 Evaluation

In order to evaluate the accuracy of resource demand measurement and estimation approaches, we used two environments. The first evaluation compares the three presented approaches (PMW-Tools monitoring, Dynatrace AM and LibReDE) with each other on two levels of granularity in a virtualized environment. In the second evaluation, we use a distributed bare-metal installation and combine direct measurement and estimation approaches.

For both evaluations, we use the orders domain application of the SPECjEnterprise2010 (Version 1.03) industry standard benchmark as exemplary enterprise application. Since the benchmark defines a workload and a dataset for the test execution, the results are reproducible for others. The orders domain application is a Java EE web application comprised of servlet, JavaServer Pages (JSPs) and EJB components. The application represents a platform for automobile dealers to sell and order cars; the dealers (henceforth called users) interact with the platform using the Hypertext Transfer Protocol (HTTP). There are three basic business transactions which describe how users interact with the system: Browse, Manage and Purchase.

### 5.3.1 Standalone evaluation

For the standalone evaluation, we installed the SPECjEnterprise2010 benchmark and its corresponding load test driver on two Virtual Machines (VMs), each deployed on separate hosts (IBM System X3755M3) to avoid interferences between the two systems. The system under test (SUT) VM contains the application server, hosting the orders domain application. The other VM executes load tests on the SUT using the Faban<sup>4</sup> harness driver of the benchmark. Both virtual machines run openSUSE 12.3 64-bit as OS and have access to 40 gigabytes of Random Access Memory (RAM). The application server VM uses six CPU cores while the driver VM has access to four CPU cores.

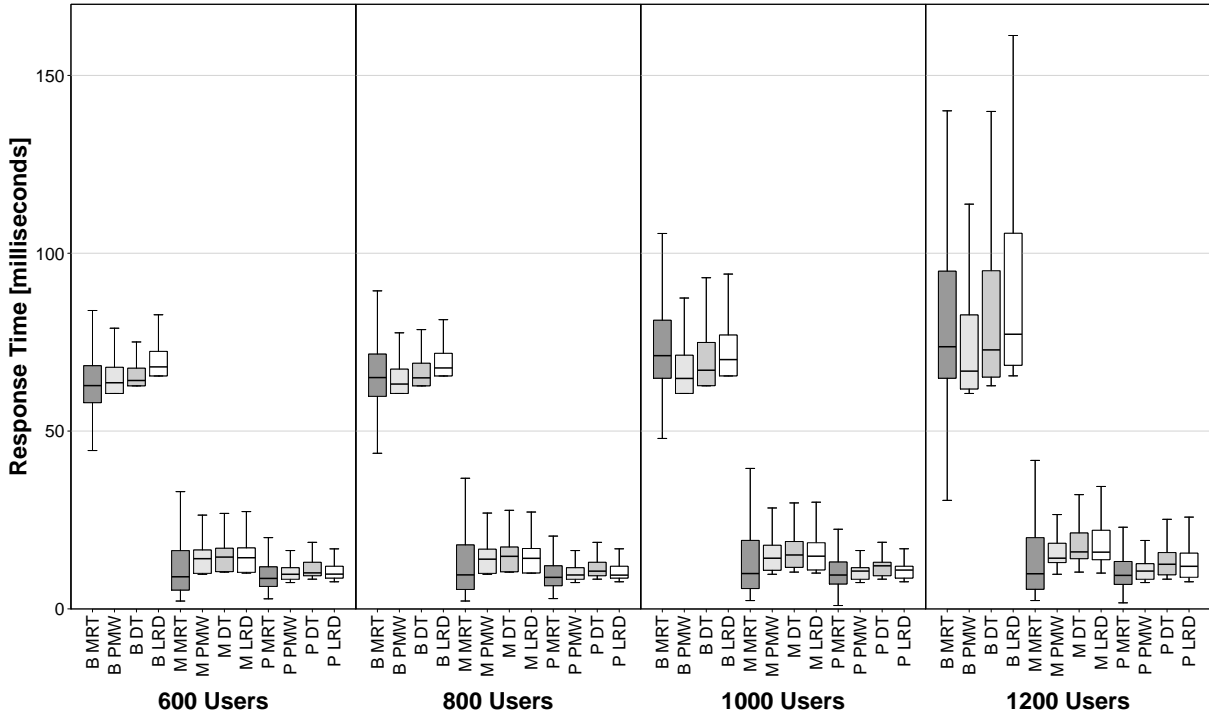
The benchmark is deployed on a JBoss Application Server (AS) 7.1 in the Java EE 7.0 full profile. The DB on the test system VM is an Apache Derby DB in version 10.9.1.0. The JBoss AS and the Apache Derby DB are both executed in the same 64-bit Java OpenJDK VM (JVM version 1.7.0\_17).

The first step of the evaluation is to obtain the relevant performance metrics (response time, utilization and throughput) of the SUT under different workloads by performing measurement runs. As the network overhead between the Faban harness and the SUT is not considered in the first step, the response time measurements are conducted by measuring the system entry point response times with the PMW-Tools monitoring. For this purpose, a workload of 600, 800, 1000 and 1200 concurrent users is put on the SUT, resulting in a mean CPU utilization of 39%, 56%, 69% and 79% on the server. Each measurement run lasts for sixteen minutes while data is only collected between a five minute ramp-up and a one minute ramp-down phase.

The standalone evaluation is conducted on two levels of granularity. We compare system entry point level, where only the boundaries of the system are monitored, with a component operation level monitoring, where each public operation of each used component is instrumented. This results in different performance models as resource demands are only measured or estimated for either servlet invocations (system entry point) or servlet calls and EJB operation invocations. For both cases we execute a load test with 600 concurrent users and collect monitoring data. Depending on the approach selected, this

---

<sup>4</sup><https://java.net/projects/faban/>



**Figure 5.2:** *Measured and simulated response times on system entry point level*

monitoring data contains either fine-grained measurements of CPU demanded time per operation invocation or only response times and total CPU utilization of the VM.

The performance models generated with this monitoring data are used for simulating the same and higher amounts of concurrent users (800 - 1200). We compare the simulated CPU utilization and the response times with actual measurements on the system. For the utilization we compare the measured mean CPU utilization (MMCPU) with the simulated mean CPU utilization (SMCPU) and calculate the relative CPU utilization prediction error (CPUPE).

When examining the CPU utilization prediction results shown in Table 5.2, it is visible that LibReDEs prediction is very accurate, especially in the replay case with 600 concurrent users and the upscaled case with 1200 concurrent users. The two monitoring solutions only measure the CPU time of the actual request thread while LibReDE also takes the overhead of the application server and CPU time for other processing like Garbage Collection (GC) into account. Dynatrace AM can use different CPU timers optimized for specific environments (i.e., VM, Windows OS, etc.) and the here used POSIX Hi-Res timer produces more accurate results than the PMW-Tools monitoring (Dynatrace, 2015).

**Table 5.2:** *Measured and simulated CPU utilization for system entry point level*

System		PMW-Tools monitoring		Dynatrace AM		LibReDE - estimation	
Users	MMCPU	SMCPU	CUPE	SMCPU	CUPE	SMCPU	CUPE
600	39,33%	36.66%	6.80%	38.73%	1.53%	39.73%	1.01%
800	55,69%	48.68%	12.58%	51.41%	7.68%	52.69%	5.37%
1000	69,28%	60.92%	12.06%	64.02%	7.58%	65.56%	5.36%
1200	79,31%	73.21%	7.69%	77.33%	2.50%	78.66%	0.82%

Figure 5.2 shows the response times for system entry point level granularity using box plots. Each box depicts one measurement/simulation series. The figure is divided into four sections, distinguishing between different user amounts. In each section, three measured response time (MRT) box plots are shown, one for each business transaction: Browse (B), Manage (M), Purchase (P). The sections are completed by nine simulation box plots, one for each of the three business transactions times the three techniques: PMW-Tools monitoring (PMW), Dynatrace AM (DT) and LibReDE (LRD).

We see that LibReDE tends to overestimate the resource demands, leading to a higher median and broader Interquartile range (IQR) for the Browse and Manage transaction, but delivers good results in general. The differences between PMW-Tools monitoring and Dynatrace AM are minimal in most cases. All approaches have in common that they cannot predict the lower quartiles. However, this is most likely caused by the fact, that only mean values for CPU demands are represented in the resource demands of the generated performance models.

The CPU utilization results and errors are similar for component operation level compared to system entry point level. Table 5.3 shows that LibReDE again produces the most accurate resource demands when simulating and comparing the CPU utilization with actual measurements. Dynatrace again is more accurate than PMW-Tools monitoring but the differences are smaller compared to the system entry point level.

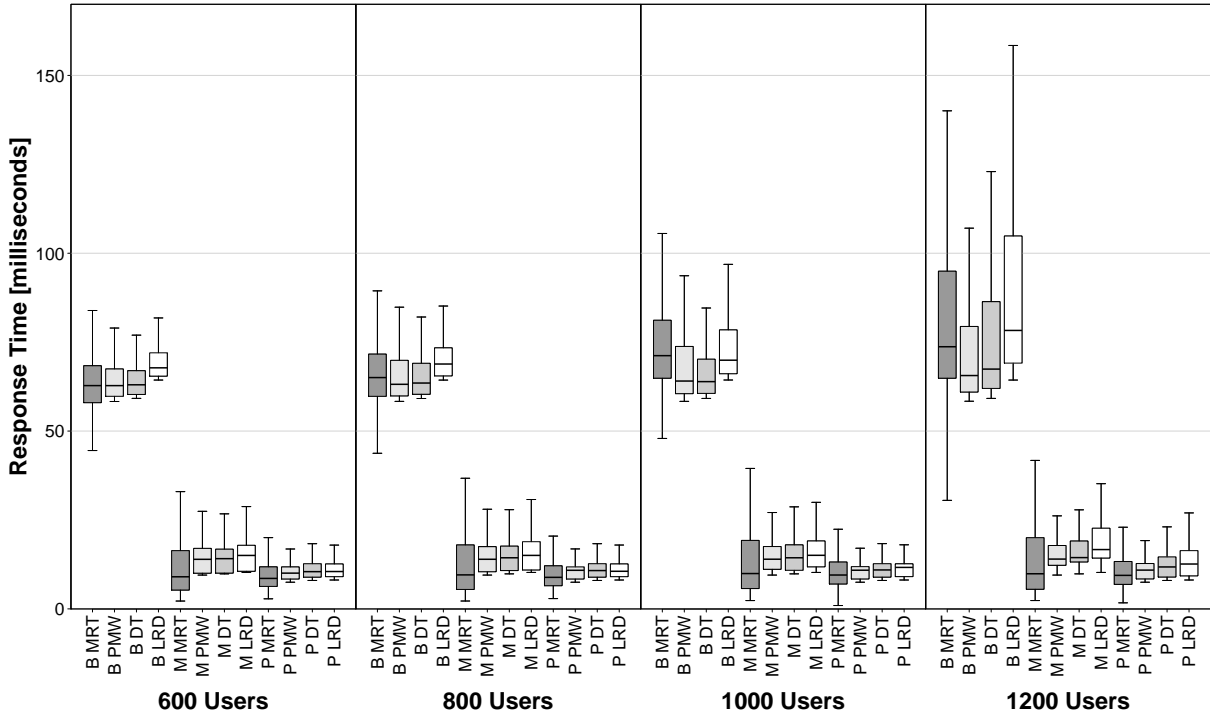
The response time errors presented in Figure 5.3 are best predicted with direct measurements. The differences between the two monitoring approaches are rather small. LibReDE overestimates in most of the cases. The upper quartiles are better predicted using estimation than direct measurements, but the median and IQR are worse with estimation approaches. Again all approaches have in common that they cannot predict the lower quartile.

### 5.3.2 Distributed Setup

The previous evaluation showed that resource estimation techniques provide sufficiently accurate results for most of the evaluated scenarios. However, in order to use these estimations, it is important to be able to measure control flows and response time on the level of granularity that needs to be represented in a model. Furthermore, estimations work only as long as response time and throughput values for all requests are available for a measurement interval. Therefore, there are a lot of cases in which it is desirable to mix direct measurements with resource estimation techniques.

**Table 5.3:** *Measured and simulated CPU utilization for component operation level*

System		PMW-Tools monitoring		Dynatrace AM		LibReDE - estimation	
Users	MMCPU	SMCPU	CPUPE	SMCPU	CPUPE	SMCPU	CPUPE
600	39,33%	36.39%	7.49%	37.21%	5.39%	39.61%	0.69%
800	55,69%	48.42%	13.04%	49.83%	10.51%	52.77%	5.24%
1000	69,28%	60.26%	13.01%	61.89%	10.67%	65.71%	5.15%
1200	79,31%	71.78%	9.49%	74.07%	6.60%	79.32%	0.01%



**Figure 5.3:** *Measured and simulated response times on component operation level*

This evaluation validates a distributed deployment scenario for SPECjEnterprise2010 in which direct measurements and estimations are used in combination. This is necessary to be able to properly account for the resource demands and times spent on different layers of the architecture (e.g., what portion is spent in the DB tier). It is important to note that the following models also account for network resource demands which was not done for the previous evaluations as the standalone setup was deployed on a single server. The models for this evaluation are automatically generated using the PMG by providing input from multiple sources (PMW-Tools monitoring, Dynatrace AM, SAR and LibReDE).

The SPECjEnterprise2010 benchmark is deployed in a multi-tier architecture consisting of a presentation, application and a data tier. As we do not have an in-depth monitoring for the data tier, we use estimation here while the presentation and application tier are instrumented using the PMW-Tools monitoring as well as the Dynatrace AM. The resulting resource demands are used to build a performance model based on PCM. In order to model the data tier, the data collection solution (i.e., PMW-Tools monitoring, Dynatrace AM) gathers the tier's response times, CPU utilization on the DB is gathered using SAR. These values are used as input for a resource demand estimation using LibReDE (Spinner/Casale et al., 2014). The generated performance model is then enriched with the data tier's estimated resource demands. Finally, the model is used to perform simulations with increasing workloads; the results are then compared to measurements of the real system to gauge the prediction performance of the approach.

To obtain a multi-tier architecture, the standard orders domain application is modified by converting the EJB components to web services. This allows for the application's deployment on two different machines. In addition, the application tier is connected to a PostgreSQL DB located on a third machine.



The different tiers of the application are deployed on three different machines which in the following will be called User Interface (UI) server, Web Service (WS) server and DB server. Additionally, a benchmark driver is deployed on one VM to generate load on the whole system by accessing the UI server using the three business transactions. To achieve a moderate load on each system, the CPU core count of each system has been modified by disabling some cores. All of the systems' technical specifications are listed in Table 5.4.

The distributed evaluation also begins with performing similar measurement runs using minimal instrumentation. Executing the same workload (600 - 1200 users), as in the previous evaluation results in a maximum CPU utilization of 77%, 59% and 68% on the UI, WS and DB server, respectively. The benchmark driver has been modified to collect the response time of the three business transactions for each invocation, instead of measuring them directly on the SUT as in the previous evaluation.

Afterwards, the UI and WS server are instrumented and another benchmark run with a workload of 600 concurrent users is performed. The collected data is used to generate a performance model using the PMG. Simultaneously, the response times per invocation and aggregated utilization of the DB server are collected. These are automatically used by the PMG as input for the LibReDE resource demand estimation. The model is further enhanced by adding latency and throughput values of the network connecting the individual servers as shown in (Brunnert/Krcmar, 2017). These values are gathered using the lmbench<sup>5</sup> benchmark suite. Finally, the finished model is used to simulate the SUT with a workload of 600, 800, 1000 and 1200 concurrent users; the duration and steady state times correspond to the ones used for the measurements.

When examining the CPU utilization values in Table 5.5 and Table 5.6, we see that the SMCPU of the DB server is predicted with very high accuracy using Dynatrace AM, with the highest error being 1.21% at 1000 concurrent users. The PMW monitoring does not intercept all JDBC calls, leading to an overestimation of CPU demands on the calls that are intercepted. Furthermore, the accounting of this calls is also missing in the WS server, leading to an underestimation of the CPU demands in the business tier. The CPU utilization of the WS server is predicted very well using Dynatrace AM, while the UI server's utilization is predicted too low. Dynatrace distributes the processing time

<sup>5</sup><http://lmbench.sourceforge.net/>

**Table 5.4:** *Software and hardware configuration of the SUT*

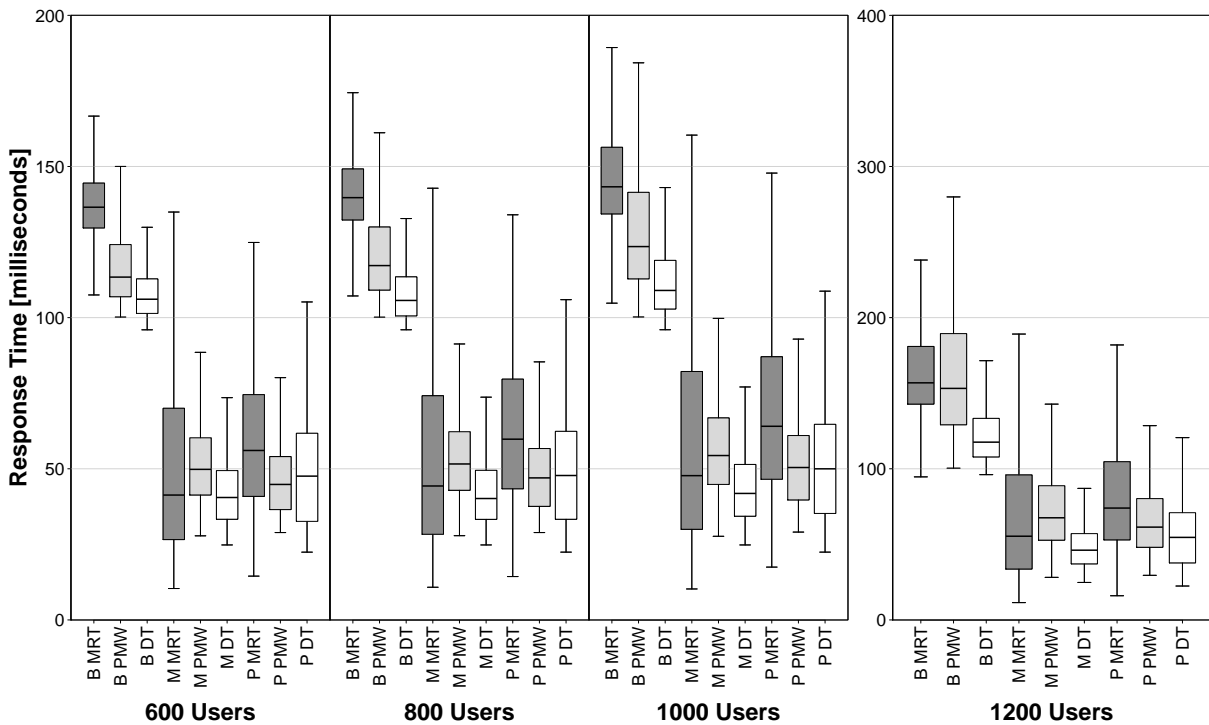
Server	UI Server	WS Server	DB Server
<b>Application</b>	SPECjEnterprise2010 (version 1.03) orders domain		
<b>AS/DB</b>	GlassFish 4.0 (build 89)	JBoss AS 7.1.1	PostgreSQL 9.2.7
<b>JVM</b>	64-bit Java HotSpot JVM version 1.7.0_71	64-bit Java OpenJDK JVM version 1.7.0_40	-
<b>OS</b>	openSUSE 12.2		openSUSE 12.3
<b>CPU Cores</b>	2 x 2.1 GHz	6 x 2.1 GHz	4 x 2.4 GHz
<b>CPU Sockets</b>	4 x AMD Opteron 6172		2 x Intel Xeon E5645
<b>RAM</b>	256 GB		96 GB
<b>Hardware System</b>	IBM System X3755M3		IBM System X3550M3
<b>Network</b>	1 Gigabit-per-second (GBit/s)		

**Table 5.5:** *Measured and simulated CPU utilization using PMW-Tools monitoring*

Users	UI server			WS server			DB server		
	MMCPU	SMCPU	CPUPE	MMCPU	SMCPU	CPUPE	MMCPU	SMCPU	CPUPE
600	39.97%	40.36%	0.96%	30.96%	26.93%	14.96%	34.51%	40.77%	15.35%
800	53.11%	54.05%	1.74%	41.86%	36.11%	15.94%	45.89%	54.54%	15.86%
1000	65.27%	67.37%	3.11%	48.39%	44.99%	7.57%	56.51%	68.02%	16.93%
1200	77.01%	80.52%	4.36%	59.71%	53.81%	10.96%	68.38%	81.42%	16.01%

**Table 5.6:** *Measured and simulated CPU utilization using Dynatrace AM*

Users	UI server			WS server			DB server		
	MMCPU	SMCPU	CPUPE	MMCPU	SMCPU	CPUPE	MMCPU	SMCPU	CPUPE
600	39.97%	33.29%	20.06%	30.96%	30.54%	1.36%	34.51%	34.25%	0.77%
800	53.11%	44.47%	19.43%	41.86%	40.82%	2.55%	45.89%	45.80%	0.20%
1000	65.27%	55.55%	17.49%	48.39%	51.03%	5.17%	56.51%	57.20%	1.21%
1200	77.01%	66.82%	15.25%	59.71%	61.34%	2.66%	68.38%	68.92%	0.79%

**Figure 5.4:** *Measured and simulated response times*

to all active operations. We have more running operations on the WS server, leading to better results for this tier compared to the UI server. The PMW monitoring instruments the CPU demands of the UI server better, because its servlet interceptor measures each operation individually. Overall, the results show that the approach is well suited for predicting the performance of a multi-tier application.

The response time values are illustrated in the box plots in Figure 5.4. The figure is divided into four sections, one section for each user amount. Each section again contains three MRT series (Browse, Manage, Purchase) and six simulation box plots. Three plots for the combination PMW-Tools monitoring and LibReDE (PMW) and three plots for the combination Dynatrace AM and LibReDE (DT). Note that the last section uses another scale as the first three sections, as the response times are significantly higher with 1200 concurrent users.

The comparison shows that the combination of resource demand measurement and estimation techniques leads to a good representation of the real system. The median of the simulated response time is close to the actual measurements. The prediction error for the median response time values is at most 25.02% for the browse transaction at 1200 concurrent users. The IQR prediction using PMW is usually a bit closer to the real system measurements than DT.

## 5.4 Related Work

This section presents related work that is concerned with measurement accuracy in different environments or the overhead caused by such measurements.

CPU accounting on VMs can be error prone due to sharing the same physical resource over multiple machines. Hofer et al. (Hofer/Hörschläger/Mössenböck, 2015) discovered that malicious accounting, so called steal time, can be detected and calculated in a VM. If not corrected, CPU utilization measurements produce wrong resource demands. Wrong CPU utilization accounting decreases the quality of performance models created either using direct measurement or estimation methods. We avoid this by isolating the SUT VM on a single host. However, virtualized environments need to correct this steal time in order to calculate accurate resource demands.

Estimating the overhead of virtualized environments has been described by Brosig et al. (Brosig et al., 2013) and Huber et al. (Huber/Quast et al., 2011). These approaches estimate, among others, virtualization overhead based on monitoring data using a queuing network. Such calculations can increase the accuracy of resource demands of such environments.

Kuperberg compared different timers and measurement approaches for a number of systems (Kuperberg, 2010). While the Dynatrace AM already offers different timers to select the most suitable one, the other two approaches rely on either the ThreadMXBean, JMX monitoring or SAR. The accuracy of these approaches can vary depending on the underlying system monitored and therefore the calculated resource demands accuracy may vary.

Measurement approaches cause overhead on the SUT. Brunnert et al. (Brunnert/Neubig/Krcmar, 2014) measured and discussed this effect for the PMW-Tools monitoring solution in previous work. This overhead effect turns out to be at around 0.003 ms for each measurement when only CPU no other resource demands are collected. This overhead can effect the system at its capacity limits, while an estimation approach can use coarse-grained monitoring data with less overhead.

## 5.5 Conclusion and Future Work

This work compared three different techniques for deriving resource demands for performance models. We compared a monitoring approach from academia, an industry monitoring solution and a library combining six different estimation approaches. These techniques have been integrated into a single automatic PMG. The evaluation compared all techniques in a standalone and a distributed setup, as well as in a virtualized and a bare-metal environment for two levels of granularity: system entry point level and component operation level.

All techniques deliver good results for both granularity levels and in all environments. Estimation techniques deliver better results for the system entry point level, but fall short behind direct measurements for the component operation level. Furthermore, direct measurements can extract resource demands on any level of detail, while estimation techniques must calculate demands for the complete system to distribute the measured utilization among the components. Estimation techniques can be applied to a broad variety of technologies as the requirements for data collection are lower. We demonstrated accurate results using a hybrid setup, where measurement approaches are used to extract resource demands for the UI and WS combined with estimations for the DB.

The evaluation uses a Java EE application. Industry monitoring like Dynatrace AM are capable of observing other technologies. Demonstrating the applicability of the framework for other technology stacks as well as extending the monitored resources are interesting challenges for further research.

# Chapter 6

## Full-Stack Performance Model Evaluation using Probabilistic Garbage Collection Simulation

Authors	Willnecker, Felix <sup>1</sup> (willnecker@fortiss.org) Brunnert, Andreas <sup>1</sup> (brunnert@fortiss.org) Koch-Kemper, Bernhard <sup>2</sup> (kochkemp@in.tum.de) Krcmar, Helmut <sup>2</sup> (krcmar@in.tum.de)  <sup>1</sup> fortiss GmbH, Guerickestraße 25, 80805 München, Germany <sup>2</sup> Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Proceedings of the Symposium on Software Performance (SSP) 2015
Status	Accepted
Keywords	Performance Evaluation, Palladio Component Model, Java, Enterprise Applications, I/O Performance Simulation, Garbage Collection Simulation
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

**Table 6.1:** *Bibliographic details for P3*

**Abstract** Performance models can represent the performance relevant aspects of an enterprise application. Corresponding simulation engines use such models for simulating performance metrics (e.g., response times, resource utilization, throughput) and allow for performance evaluations without load testing the actual system. Creating such models manually often outweighs their benefits. Therefore, recent research created performance model generators, which can generate such models out of Application Performance Management software. However, a full-stack evaluation containing all relevant resources of an enterprise application (Central Processing Unit, memory, network and Hard Disk Drive) has not been conducted to the best of our knowledge. This work closes this gap using a pre-release version of the next generation industry benchmark SPECjEnterpriseNEXT of the Standard Performance Evaluation Corporation as example enterprise application, the

Palladio Component Model as performance model and the performance model generator of the RETIT Capacity Manager. Furthermore, this work extends the generated model with a probabilistic garbage collection model to simulate memory allocation and releases more accurately.

## 6.1 Introduction

Evaluating the performance of an enterprise application (EA) requires either load testing this application or creating a performance model to simulate the performance metrics (e.g., response times, resource utilization, throughput) of a system. Simulations are often less cost intensive as they do not require a full scale deployment environment (Kounev, 2005). Performance models and corresponding simulation engines have been introduced to the scientific community (Becker/Koziolk/Reussner, 2009). Resource profiles based on the Palladio Component Model (PCM) have demonstrated to accurately represent the Central Processing Unit (CPU) and network demand of EAs (Becker/Koziolk/Reussner, 2009; Brunnert/Krcmar, 2017). Resource profiles are generated using a performance model generator, as a manual creation of such profiles often outweighs their benefits (Kounev, 2005). These profiles already introduce heap, as the most relevant aspect of memory in EAs, and contain a Hard Disk Drive (HDD) representation (Brunnert/Krcmar, 2017). However, these concepts have not been evaluated. This work demonstrates the heap and HDD modeling and simulation capability of resource profiles using the industry benchmark SPECjEnterpriseNEXT<sup>1</sup> as EA and the performance model generator of the RETIT<sup>2</sup> Capacity Manager .

An accurate model needs to take all relevant resources (CPU, network, memory, and HDD) into account. The generation of such a model requires to monitor a running artifact using Application Performance Management (APM) software (Willnecker/Brunnert et al., 2015a). This data can be collected during small scale test runs and scaled to the expected workload (Brunnert/Krcmar, 2017). In this work we use the RETIT Java EE Monitoring and extend it with HDD demand measurements for Linux based systems. Furthermore, the heap model of resource profiles is extended using a young and old generation garbage collection (GC) model and corresponding simulation to increase the correctness of the heap simulation.

## 6.2 Garbage Collection Model

The RETIT performance model generator already contains a heap representation. This representation is based on a simple memory model where allocated heap is immediately

---

<sup>1</sup>SPECjEnterpriseNEXT is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterpriseNEXT 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.

<sup>2</sup><http://www.retitt.de/>

freed when the contained objects are released (Brosig et al., 2013). The PCM entity *Passive Resource* is used for this purpose (Becker/Koziolok/Reussner, 2009). Each model contains one component called *Heap* that contains one *Passive Resource* called *HeapSpace* with the available memory in the Java Virtual Machine (JVM) heap space. Before and after each operation invocation an allocation and free operation is called with the number of bytes allocated or released. The values are derived from monitoring and represent the mean bytes used in this operation invocation. This implementation has certain disadvantages: (i) each JVM manages its own heap space, therefore each JVM instance in the performance model needs a separate *Heap* component (ii) the memory in a JVM is only cleaned when GC occurs. Only after such a GC run a certain amount of heap is freed.

We change the model in order to address the aforementioned disadvantages. Each JVM representation uses its own *Heap* component representing its own heap space. We measure the mean time between different GC runs and the average number of bytes released. We distinguish between two types of GC: young and old generation GC (Libič et al., 2015). Even though GC implementations may vary throughout different JVM versions and types, these two GC types occur in most of the GC implementations (Libič et al., 2015). For each GC type we add an *Open Workload Scenario* that calls the GC operation in the *Heap* component. The inter-arrival time between these scenarios is the mean time between two GC runs of the same type. The operation call has one parameter: the average number of bytes released per GC run. The operation releases the provided number of bytes in the *HeapSpace*. This probabilistic approach converges the generated performance model to the real memory management in the Java Virtual Machine (JVM).

## 6.3 Evaluation

The SPECjEnterpriseNEXT industry benchmark is the successor of the SPECjEnterprise2010 benchmark. Both are Java Enterprise Edition (EE) applications typically used to benchmark the performance of different Java EE application servers. We use a pre-release version<sup>3</sup> of the SPECjEnterpriseNEXT as example EA. This application represents an insurance policy holder that manages car insurances. The application consists of three different components (Insurance Domain, Vehicle Service, Insurance Agent) as depicted in Figure 6.1. Each component is deployed as one deployment unit in one application server running in a virtual machine (VM). Each VM has 4 CPU cores, runs the Java EE application server Wildfly 8.1.0, the embedded database Derby 10.11.1.1, and has 8 GB of heap for the JVM. The operation system is OpenSuse 13.2 (x86\_64) and a 1 GBit/s network connection is used for communication between the different VMs.

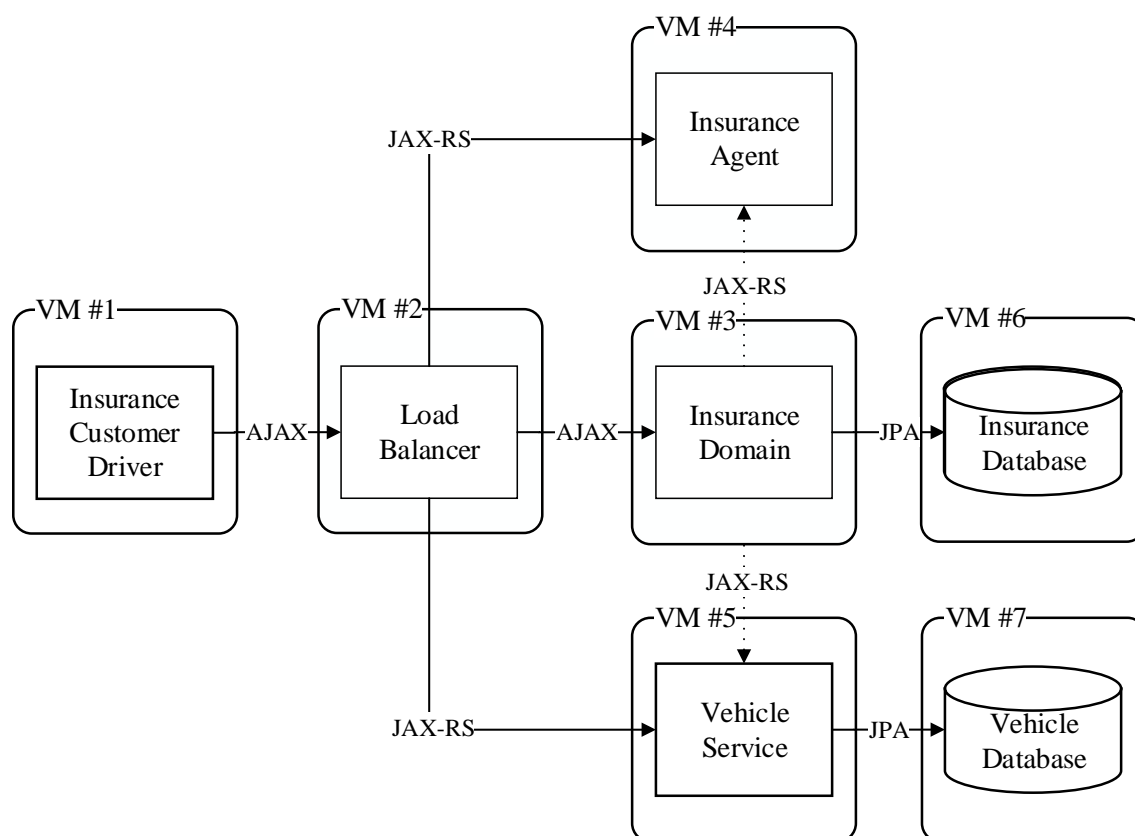
The Insurance Customer Driver is based on Faban<sup>4</sup> and executes five different business transactions on the Insurance Domain server, which triggers JAX-RS<sup>5</sup> REST calls on the other two servers (Fielding, 2000). We executed a measurement run without the RETIT Java EE monitoring to minimize instrumentation overhead and use the results to validate

---

<sup>3</sup>version from 29.06.2015

<sup>4</sup><http://faban.org/>

<sup>5</sup><https://jax-rs-spec.java.net/>



**Figure 6.1:** *SPECjEnterpriseNEXT* deployment

our simulation. The response times of the business transactions were measured on the driver. CPU and heap utilization were measured using Java Management Extensions (JMX) and the HDD demand using IOTop<sup>6</sup>. These measurements have been executed for 100, 120, 140 and 160 users in a 17 min interval with 5 min ramp up and 2 min ramp down phase. The model generation is based on a run with 100 users and activated RETIT Java EE monitoring on system-entry-point level. The monitoring is extended to derive the resource demand of the HDD. Therefore, the procfcs<sup>7</sup> system is used, a pseudo file system containing the read and write bytes per thread.

To calculate the resource capacity of the HDD we use the bonnie++ 1.97 benchmark<sup>8</sup>. For calculating the network latency and bandwidth we use LMBench 3.0 (McVoy et al., 1996). The number of CPU cores is stored in the resource environment replicas setting for each *Resource Container* and the available heap is stored in the *Passive Resource* of the corresponding heap component for each JVM.

The usage model is based on an *Open Workload Scenario*. The inter-arrival time per business transaction  $IAT_{BT}$  is calculated based on the *SteadyStateTime* of 600s, the total number of calls per business transaction  $TotalCalls_{BT}$  and the 100 users of the

<sup>6</sup><http://guichaz.free.fr/iotop/>

<sup>7</sup><https://www.kernel.org/doc/Documentation/filesystems/proc.txt>

<sup>8</sup><http://www.coker.com.au/bonnie++/>



**Table 6.2:** *Measurement and simulation results*

Resource	User	Metric	Insurance Domain	Vehicle Service	Insurance Agent
CPU	100	Measured utilization	43.80%	51.57%	46.13%
		Simulated utilization	40.48%	48.29%	43.04%
		Relative error	7.58%	6.38%	6.70%
	160	Measured utilization	70.19%	77.91%	71.73%
		Simulated utilization	64.75%	77.25%	68.88%
		Relative error	7.75%	0.85%	3.97%
Heap	100	Measured demand	1458.58 MB	1435.41 MB	-
		Simulated demand	1299.22 MB	1319.41 MB	-
		Relative error	10.93%	8.08%	-
	120	Measured demand	1360.19 MB	1304.46 MB	-
		Simulated demand	1296.29 MB	1317.88 MB	-
		Relative error	4.70%	1.03%	-
HDD	100	Measured demand	0.23%	0%	0%
		Simulated demand	0.21%	0%	0%
		Relative error	10.72%	0%	0%
	160	Measured demand	0.34%	0%	0%
		Simulated demand	0.35%	0%	0%
		Relative error	0.65%	0%	0%

generation run (*UsersGeneration*) as depicted in Equation 6.1. The number of users in the simulation *UsersSimulation* is up-scaled from 100 to 160 users in steps of 20.

$$IAT_{BT} = \frac{SteadyStateTime}{\frac{TotalCalls_{BT}}{UsersGeneration} * UsersSimulation} \quad (6.1)$$

The inter-arrival time for the garbage collection  $IAT_{GC}$  is calculated in a similar way. Instead of the total number of business transactions we use the number of GC events  $TotalEvents_{GC}$  intercepted during the generation run as shown in Equation 6.2. The calculation is conducted for each GC type (young and old generation GC). Again, the number of users *UsersSimulation* in the simulation is up-scaled from 100 to 160 users in steps of 20.

$$IAT_{GC} = \frac{SteadyStateTime}{\frac{TotalEvents_{GC}}{UsersGeneration} * UsersSimulation} \quad (6.2)$$

Table 6.2 shows the results for 100 and 160 users. The complete results are available online<sup>9</sup>. The heap simulation has only been conducted for the Insurance Domain and the Vehicle Service as the heap demand for the Insurance Agent server is almost 0. The relative error of the heap simulation is between 10.93% and 1.03%. HDD demands only occur on the Insurance Domain server. Even though, the utilization for the HDD is relative low the simulation delivers accurate results for this resource. The relative error here is between 10.72% and 0.65%. The relative CPU utilization error is below 10%. The utilization in the GC models improves the CPU utilization simulation compared to previous research (Willnecker/Dlugi et al., 2015c). The response time error based on the median of simulation and measurements is between 0.16% and 51.65% depending on the business transaction and user scale.

<sup>9</sup>[http://download.fortiss.org/public/pmw/SSP2015/FullStack\\_EvaluationResults.\(xlsx/pdf\)](http://download.fortiss.org/public/pmw/SSP2015/FullStack_EvaluationResults.(xlsx/pdf))

## 6.4 Conclusions

The improvements and extensions to the model generated by the RETIT performance model generator prove to be accurate for all resources. Even though the resource utilization simulation is accurate, the response time error leaves room for improvement. The parameter of a request as well as the number of available database and server threads have impact on the response time simulation. In total the response times of the deployment are quite high, as the embedded database is not very scalable. A setup with a dedicated database server appears to be a better solution but requires additional APM measurement tools or resource demand estimations (Willnecker/Dlugi et al., 2015c). The probabilistic GC model delivers promising results, however the average number of bytes could be eliminated as an input parameter for the GC operations. This requires, that the actual number of bytes ready to be freed is stored in a variable of the model. Furthermore, the current evaluation is based on system-entry-point level. The performance model generator of the RETIT Capacity Manager provides component-level model generation which could improve the simulation results presented in this work.

# Chapter 7

## Model-based Prediction of Automatic Memory Management and Garbage Collection Behavior

Authors	Willnecker, Felix <sup>1</sup> (willnecker@fortiss.org) Krcmar, Helmut <sup>2</sup> (krcmar@in.tum.de)  <sup>1</sup> fortiss GmbH, Guerickestraße 25, 80805 München, Germany <sup>2</sup> Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Simulation Modelling Practice and Theory
Status	Submitted
Keywords	performance evaluation, capacity planning, Microservices, architecture design decisions, architecture quality
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

**Table 7.1:** *Bibliographic details for P4*

**Abstract** Performance models focus on resource consumption and the effects of CPU, network, or hard-disk utilization. These resources usually have the largest effect on the response times and throughput of an application. However, deficient memory management can have severe effects on an application and its runtime, such as overlong response times or even crashes. As memory management has been disregarded in performance simulations, we address this gap with an approach based on memory measurements and derived metrics to predict the behavior of memory management. Although numerous works exist that analyze memory management and especially garbage collections, accurate prediction models are rare. We demonstrate the automatic extraction of memory behavior using an extended performance model generator. Furthermore, the approach is evaluated using the SPECJEnterprise2010 and the SPECjEnterpriseNEXT industry benchmark, using different resource environments, garbage collection algorithms, and workloads. This work demonstrates that a certain set of probabilities allows one to create a memory profile for an architecture and predict the behavior of the memory management. The results of such predictions can be used for better capacity planning (on-premise), cost-prediction

---

(cloud), architecture evaluation and optimization, or memory profiling. This approach allows for a continuous evaluation of an enterprise architecture regarding its memory footprint.

Automatic memory management is part of modern architectures, virtual runtime environments (e.g., Java Virtual Machine (JVM), .NET, etc.), and scripting languages (e.g., Python, PHP, etc.). Allocating and releasing objects during runtime is the main objective of automatic memory management. An important feature of automatic memory management is Garbage Collection (GC) (Jones/Lins, 1996; Jones/Hosking/Moss, 2016). GC searches for objects that are no longer referenced and are thus no longer used in order to release the memory space these objects occupy (Libič et al., 2015; Jones/Lins, 1996; Jones/Hosking/Moss, 2016). Most runtime environments contain at least one, often multiple GC implementations, which again follow a number of strategies depending on their configuration (Libič et al., 2015; Blaschek/Lengauer, 2015). GC execution runs are usually triggered when memory reaches a certain level (e.g., 90% full) and require a certain amount of Central Processing Unit (CPU) resources and execution time. Each of these GC runs frees up a certain amount of memory, depending on the current state of the object space. Some GC algorithms can even suspend the application or its runtime for short period of time to free up memory spaces. Therefore, GC strategies can have a large impact on the application runtime, its resource utilization, throughput, and response times.

The impact of different automatic memory management or GC strategies is unveiled only under some load, as memory spaces have to be filled with a vast amount of objects until certain thresholds are reached. Small, functional tests usually do not reach these thresholds. Therefore, predicting the impact of different memory strategies requires a fully operational test environment including a load test and test drivers. The number of potential memory strategies and GC implementations is high, so testing and measuring the impact is very time consuming and costly (Blaschek/Lengauer, 2015). Performance model simulations can reduce the effort by simulating the effects of software, hardware, runtime configuration, or workload changes on the application's performance (Brunnert/Vögele et al, 2014). This requires a comprehensive performance model, a performance model generator to reduce the effort for creating such models, and a corresponding simulation environment (Kounev, 2005; Willnecker et al., 2015b).

Even though automatic memory management and GC can have a huge impact on the performance of applications, memory resources are usually not considered by modern performance models (Willnecker/Krcmar, 2016). Most performance models and corresponding solvers or simulation engines focus on resources like the CPU, Hard Disk Drive (HDD) or network bandwidth and latency in order to predict performance metrics (e.g., resource utilization, throughput, response times) (Brunnert/Krcmar, 2017; Brosig/Huber/Kounev, 2014). The effects of false or adverse memory management are usually not considered (Libič et al., 2015). This work closes this gap by introducing a memory model and corresponding extension for an existing simulation engine that can predict the effects of memory management including garbage collection.

This model-based approach, integrated into a continuous delivery pipeline, allows architects and developers to continuously monitor and evaluate the memory footprint of their

applications' architecture. Changes to the architecture caused by new features or refactoring become visible. This allows architects to evaluate their design decisions based on all major resources and to preserve and increase the quality of their applications.

The introduced approach is evaluated by an extension to the Palladio Component Model (PCM) (Becker/Koziolek/Reussner, 2009; Reussner et al., 2016). This extension consists of a meta-model for memory resources including different memory spaces, GC behaviors, and memory growth and shrinking. This allows one to model automatic memory management as well as dynamic memory management, and all types of different GC implementations and configurations. We complement this with a memory and GC monitoring solution for Java Enterprise Edition (EE) applications and an extension for a performance model generator (PMG) (Willnecker/Krcmar, 2016). The generator creates performance models based on monitoring data considering the four major resource types CPU, HDD, and network in addition to our extension memory.

We provide an evaluation on the accuracy and robustness of this approach using the SPECjEnterprise2010<sup>1</sup> industry benchmark and the SPECjEnterpriseNEXT industry benchmark. Both enterprise applications are evaluated using different GC implementations and configurations in an on-premise and in an industry cloud environment. The evaluation demonstrates the feasibility of our approach in a monolithic (SPECjEnterprise2010) and in a distributed microservice architecture (SPECjEnterpriseNEXT).

This paper builds on our previous work (Willnecker/Krcmar, 2016; Willnecker et al., 2015b) on memory-aware deployment topology optimization and contains the following major improvements and extensions:

- (i) In this work, we introduce memory spaces to simulate object longevity and their relation to GC runs. This allows to promote and move objects to other memory spaces.
- (ii) We extended our memory resource with a flexible GC implementation that allows one to separately manage each space and, thus, work for typical managed memory scenarios (e.g., Java, .NET) (Oransa, 2014; Oracle Cooperation, 2015). Previous versions of this concept used a fixed implementation. Each GC behavior manages at least one and up to several memory spaces.
- (iii) We added the ability to grow and shrink memory according to the current state of committed memory. Both mechanisms are triggered by thresholds that are parameters of the runtime.
- (iv) We evaluate the concept using multiple applications focusing on the accuracy of the memory predictions. The variables of our experiments are application, workload, runtime configuration and resource environment.

We present our research in the following way. After this section we present related approaches that consider performance analysis or simulation of GC behavior (Section 7.1). We introduce use cases for research and industry applying our approach in Section 7.2,

---

<sup>1</sup>SPECjEnterprise2010 and SPECjEnterpriseNEXT are a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterprise2010 and SPECjEnterpriseNEXT 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.

followed by our methods and a description of our approach in Section 7.3. We present the setup for our evaluation in Section 7.4. Afterwards, we elaborate on our results, followed by a discussion. We close with our conclusion and future work in Section 7.5.

## 7.1 Related Work

This section presents related approaches regarding GC monitoring and/or modeling to observe or predict the performance of applications and related model generation approaches.

### 7.1.1 Memory management

GC implementations have been the topic of many research and development projects (Dijkstra et al., 1978; Boehm/Weiser, 1988; Jones/Lins, 1996; Appel, 1989). Most of these projects focus on implementing new GC approaches. Predicting the effects, especially on the performance of applications, was often disregarded. Load tests and profiling, trial and error approaches, have been the state of the art to optimize GC and memory management (Blaschek/Lengauer, 2015). However, some approaches already focused on performance simulation and the prediction of GC and memory management:

The work of Libiř et al. (2014) demonstrated an accurate GC simulation (Libiř et al., 2014). Their approach simulates fine-grained object movements in the JVM process and demonstrated impressive accuracy considering the behavior of the GC (Libiř et al., 2014). Unfortunately, the computation power necessary to achieve this was quite high (Libiř et al., 2014). The simulation costs outweighed the computational costs for executing the same workload in a real JVM and just observing the behavior (Libiř et al., 2014). Thus, we propose a simpler approach based on probabilities to reduce the computational costs for predicting memory behavior. Libiř et al. (2015) further presented an application of their approach in order to estimate the effects of code additions regarding the GC (Libiř et al., 2015).

Blaschek et al. (2015) presented a large set of benchmarks executed with numerous GC options using the JVM (Blaschek/Lengauer, 2015). They estimated the most relevant GC options as a decision guidance for finding the best suited options for the current approach (Blaschek/Lengauer, 2015). A good set of default options for the GC has been identified as part of their research (Blaschek/Lengauer, 2015). Their research is based on observing the GC behavior and searching for patterns in the results of these observations (Blaschek/Lengauer, 2015). The conducted experiments for collecting a significant amount of data took months of computation. However, applying these options to other applications has not been researched yet (Blaschek/Lengauer, 2015).

Instead of optimizing the overall GC overhead, Saraswati et al. (2016) try to optimize the timing of GC runs in a highly distributed systems (Saraswati/Chatterjee/Ramachandra, 2016). They search for optimal moments to trigger GC on one node of a distributed clus-

ter (Saraswati/Chatterjee/Ramachandra, 2016). During that GC, run the affected node slows down, but the rest of the system balances the load of the busy node (Saraswati/Chatterjee/Ramachandra, 2016). This prohibits a situation in which multiple nodes simultaneously execute a GC run leading to a heavy load on the remaining system and potential crashes (Saraswati/Chatterjee/Ramachandra, 2016).

## 7.1.2 Performance Model Generation

Our approach is based on performance model generation approaches. Modeling and simulating memory profiles, regarding automatic and dynamic memory management, requires an existing performance model containing at least CPU resource demands. In addition, an applicable performance model generator needs to derive transaction workflows of an application out of monitoring data. Several approaches for generating such models have been introduced to academia and industry:

We rely on the performance model generation approach described by Brunnert et al. (2015), which is nowadays available as the RETIT<sup>2</sup> Capacity Manager (Brunnert/Krcmar, 2017). This PMG considers CPU, HDD, and network demands based on fine-grained resource demand measurements (Willnecker/Dlugi et al., 2015c). Furthermore, it already contains a rudimentary memory model that can simulate dynamic memory management. We extend this generator as it seems most suited for our purpose. An earlier version of this model generator has been developed within our research group. Therefore, we are most familiar with this generator and selected it for our extension.

Similar approaches using Queuing-networks instead of Architecture-level performance models have been described by Rolia et al. (1995) (Rolia/Vetland, 1995). Although, Queuing-networks allow to adopt to different resource environment or workload, the adoption is rather complicated due to the missing separation between software, hardware, and workload in to separated models. This makes it hard to predict the effects of software changes to other environments or other workloads. Architecture-level performance models have been introduced and use different sub-models to ease model changes (Becker/Kozielek/Reussner, 2009).

Brosig et al. (2014) described a performance model generation approach based on resource demand estimations (Brosig/Huber/Kounev, 2014). This generator approach also uses Architecture-level performance models as we do, but only considers CPU and network demands (Brosig/Huber/Kounev, 2014). The approach presented by Brunnert et al. (2015) fulfills all requirements for integrating our memory model approach (Brunnert/Krcmar, 2017). Therefore, this approach was extended in this work.

---

<sup>2</sup><https://www.retitt.de/>

### 7.1.3 Performance Management

PMG combine Application Performance Management (APM) and Software Performance Engineering (SPE) research (Brunnert/Vögele et al, 2014). One of the main problems with pure APM solutions is the amount of data collected while observing an application. Millions of data points are collected easily, but the analysis of so much data is challenging (Brunnert et al., 2015). Silo departments for software engineers and software operations further increase this problem (Brunnert et al., 2015). The DevOps movement creates holistic teams of engineerings and operators and brings the responsibility of operating applications to the team that created this application. This means, regarding memory management, that the parameters influencing the memory management are now managed by DevOps teams. Using an APM/SPE combination approach as suggested here allows these teams to easily integrate memory analysis into their delivery pipeline and adjust the parameters to the actual workload, changes in the software, or changes in the runtime environment. Such holistic tools were identified as a major challenge for software engineering in terms of performance by Brunnert et al. (2014) (Brunnert/Vögele et al, 2014).

## 7.2 Use Cases

We identified four major use cases that require performance models enhanced with accurate dynamic or automatic memory management: Capacity planning for enterprise applications, deployment optimization for distributed applications, memory footprints of complex application architectures, and the prediction of resource requirements for self-adaptive systems.

Performance models work well for planning the capacity of Enterprise Applications (EAs) (Brunnert/Krcmar, 2017). Accurate capacity planning is conducted using a performance model containing the relevant resource calls and capabilities (e.g., CPU, HDD, network, and memory). In order to size the necessary capacity, we can simulate the expected workload and evaluate the necessary resource capacity (e.g., number of CPU cores, storage capacity and speed, or network speed) (Brunnert/Krcmar, 2017). Performance model generators produce such models. Simulating such generated models provides accurate results that predict all aspects of an application compared to simple approximations based on total CPU utilization and response times (Brunnert/Krcmar, 2017). However, this planning can be misleading if they do not contain all major resources and their behavior. Our memory and GC behavior model allows us to accurately represent real memory resources and thus enhance capacity planning based on (generated) performance models. Our approach contributes to a holistic model and increases the quality on which capacity planning decisions are based.

Such accurate and holistic models are helpful to conduct architecture optimization (Willnecker/Krcmar, 2016). Instead of just planning and sizing necessary server capacity, optimization also searches for an optimal deployment (e.g., cheapest, fastest, ...) (Willnecker/Krcmar, 2016). This is especially useful in distributed applications following the mi-



crosservice pattern. This allows one to sort CPU-intensive applications to HDD-intensive services. While one service is waiting on the HDD, the other service can utilize the CPU for calculations. These optimizations require a holistic model that covers the main resource types, otherwise invalid deployment topologies result (Willnecker/Krcmar, 2016). A topology becomes invalid if, for instance, two services complement each other utilizing different resources but require more memory than available. The representation of memory and GC behavior enhance the optimization process by covering all major resource types. Rouseel and Branson (2017) showed that the potential savings of an optimized deployment easily exceeds 1 million \$ annually in public Infrastructure as a Service (IaaS) environments (Roussel/Branson, 2017).

Evaluating the impact of code changes and architecture re-factoring to the overall memory footprint of an application can be hard to achieve. The observation over a longer period that includes multiple garbage collections for all used memory spaces is necessary. Our approach assists in and simplifies measuring and evaluating the memory footprint of an application. We first observe the memory consumption of all operations multiple times and calculate statistical values like quartiles, mean and median. An increased median denotes an increase in total memory consumption for at least this operation. Furthermore, we measure GC runs, the spaces and amount of bytes they clean, and the memory consumption of such a run. We use this to calculate thresholds for GC excesses and their effect on memory and CPU resources. Statistics on these metrics are represented in our performance model and can be simulated. We can then simulate the effects of small code changes or changes in the architecture on the overall performance of the system even though the deployment topology in a productive environment diverges from the one in which we conduct our measurements (Willnecker/Krcmar, 2016). Brunnert/Krcmar (2017) show that such simulations integrated into a continuous delivery pipeline can automatically detect performance bugs introduced into the code base.

Automatically adapting the runtime of an application is an important feature of autonomous systems. Adaption requires one to predict the necessary resources in the near future in order to react to these predictions. This can result in spawning new instances, changing instance types/sizes, or dismissing unused instances to save costs. Herbst et al. (2014) showed how workload predictions using performance models can help self-adaptive systems and predict the workload in the near future. Using these forecasts in combination with the simulation of holistic performance models allows one to predict the required resources. This work contributes to self-adaptive algorithms by providing an accurate memory model, therefore allowing one to consider memory resource requirements when adapting the runtime of an application automatically.

## 7.3 Research Method

Our research is based on multiple controlled experiments following the design science approach as depicted in Figure 7.1 (Hevner et al., 2004; Wieringa, 2014). We iterated multiple evaluation cycles altering our key artifacts: Memory Meta-Model and our Memory Model Generator. During our experiments we compared a real load test and the

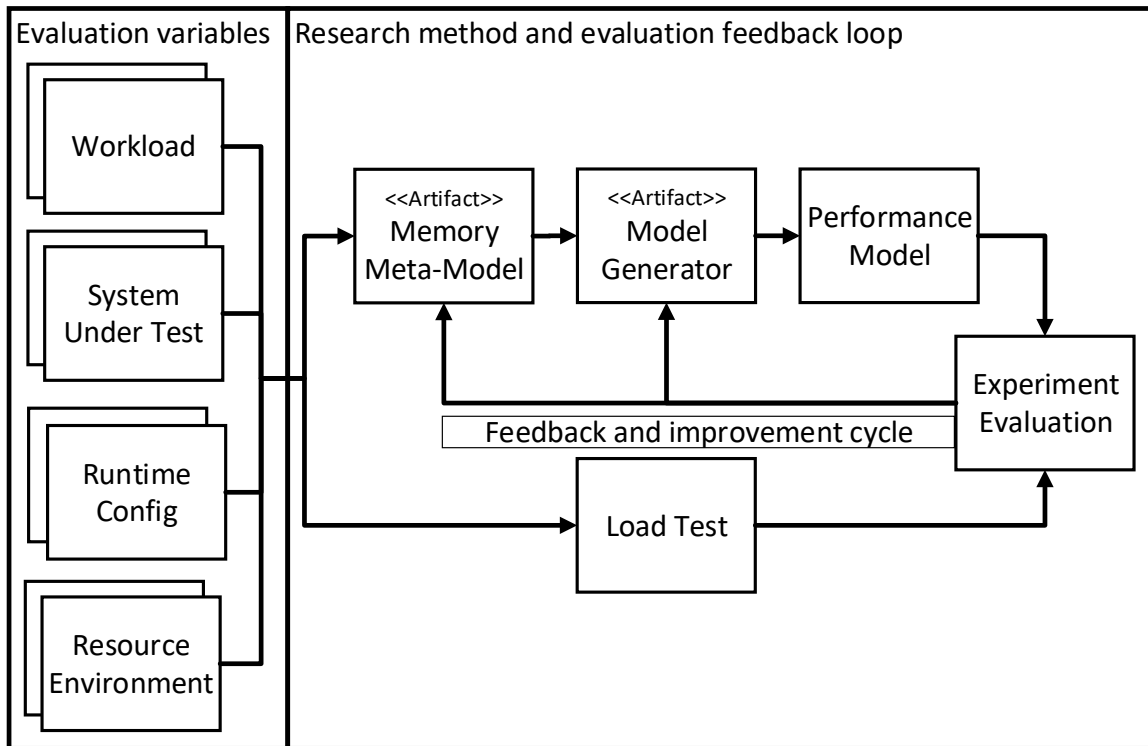


Figure 7.1: *Research method and evaluation process*

simulation of generated performance models containing our automatic memory management and GC behavior model. We extended the PCM meta-model with our memory model in order to simulate memory along with CPU, HDD, and network resources (Becker/Koziolok/Reussner, 2009; Reussner et al., 2016). Furthermore, we extended a previously created PMG to create an instance of the extended PCM meta-model (Becker/Koziolok/Reussner, 2009; Reussner et al., 2016). We used the RETIT Capacity Manager as PMG and extended it to generate memory resource and GC instances (Brunnert/Krcmar, 2017).

Our experiments used four different input variables, which we altered throughout our research: Different workloads, different system under tests (SUTs), different runtime configurations, and different resource environments. We used the Orders Domain of the SPECjEnterprise2010 benchmark as an example of a monolithic EA and the SPECjEnterpriseNEXT benchmark as a distributed application following the microservice approach (Fowler/Lewis, 2014). We executed both benchmarks with multiple workloads in two different environments: In an on-premise setup and a cloud setup using Amazon Web Services (AWS) Elastic Compute Cloud (EC2) infrastructure.

In order to evaluate our approach, we compared the simulation results of our generated model with real executions of the system. Therefore, we executed one load test while detailed monitoring was activated including traces for each operation, its resource consumptions, and GC execution listeners. We generated a performance model representing the SUT out of this data. Afterwards, we conducted load tests on the SUT with coarse-grained monitoring (e.g., total CPU utilization) activated in order to minimize

the monitoring overhead. We simulated the generated model and compared the results with the load test monitoring using the Performance Evaluation Tool (PET) (Kroß et al., 2016). PET is service created by our research group for easy comparison and analysis of performance monitoring results and performance simulation results. It provides a Representational State Transfer (REST) interface and a web interface to support both automatic and manual analysis by other tools and performance analysts.

The first experiment validated the accuracy of the predicted performance metrics, response times, resource utilization, and throughput in a replay scenario. To strengthen the evidence of our approach, we conducted experiments using other workloads and runtime configurations, but used the same performance model. We also conducted the adjustments of the load test in the corresponding PCM sub-models. For instance, we altered the workload in the load test configuration and in the PCM *Usage Model*. Afterwards, we again compared the predicted performance metrics with the monitoring of the load test.

Finally, we deployed the SUTs in a different PCM *Resource Environment*. Our generation runs were conducted in the on-premise scenario, thus we deployed the SUTs on the AWS EC2. We needed to calibrate the PCM *Resource Environment* model, as the EC2 CPU speed and the network bandwidth and latency diverges from our on-premise setup. Therefore, we used the SPEC CPU 2006<sup>3</sup> benchmark in our on-premise setup and in the EC2 infrastructure. The relative CPU speed was calculated in the *Resource Environment* model. To benchmark the network, we used lmbench<sup>4</sup>, which was executed on-premise and in the cloud. The resulting bandwidth and latency were also configured in the PCM *Resource Environment* model. We conducted load tests in both environments and compared them with the results of our simulation.

Our memory meta-model, our memory model generation approach, and the measurements we conducted constantly evolved over the series of experiments. We constantly enhanced our research artifacts and published intermediate results (Willnecker et al., 2015b; Willnecker/Krcmar, 2016). For this work, we conducted a complete evaluation with the latest meta-model, monitoring, and generator. We present the results in Section 7.4.3.

### 7.3.1 Memory Management and Garbage Collection

Memory of an application is usually divided into several sections. One section is pretty static and contains the (compiled) program code (Forouzan, 2013). This memory section is relatively small and can be well estimated as the size of the required memory space is known right after the build process (Forouzan, 2013). Sometimes libraries are loaded dynamically during runtime, which leads to an increased demand in this section, but still this memory section stays easy manageable and can be estimated by the programmer or architect. In Java, this part of the memory is divided into “Permanent Generation“ ( $\leq$  Java 7) or “MetaSpace“ ( $\geq$  Java 8), containing classes and methods loaded in the current JVM, and the “Code Cache“, which contains native code usually compiled out

---

<sup>3</sup><http://www.spec.org/cpu2006/>

<sup>4</sup><http://lmbench.sourceforge.net/>

Virtual or reserved	Eden	Survivor 1	Survivor 2	Tenured	PermGen/MetaSpace	Virtual or reserved
Young Generation				Old Generation	Permanent Generation	
Heap Memory					Non-Heap	
JVM Total Memory						

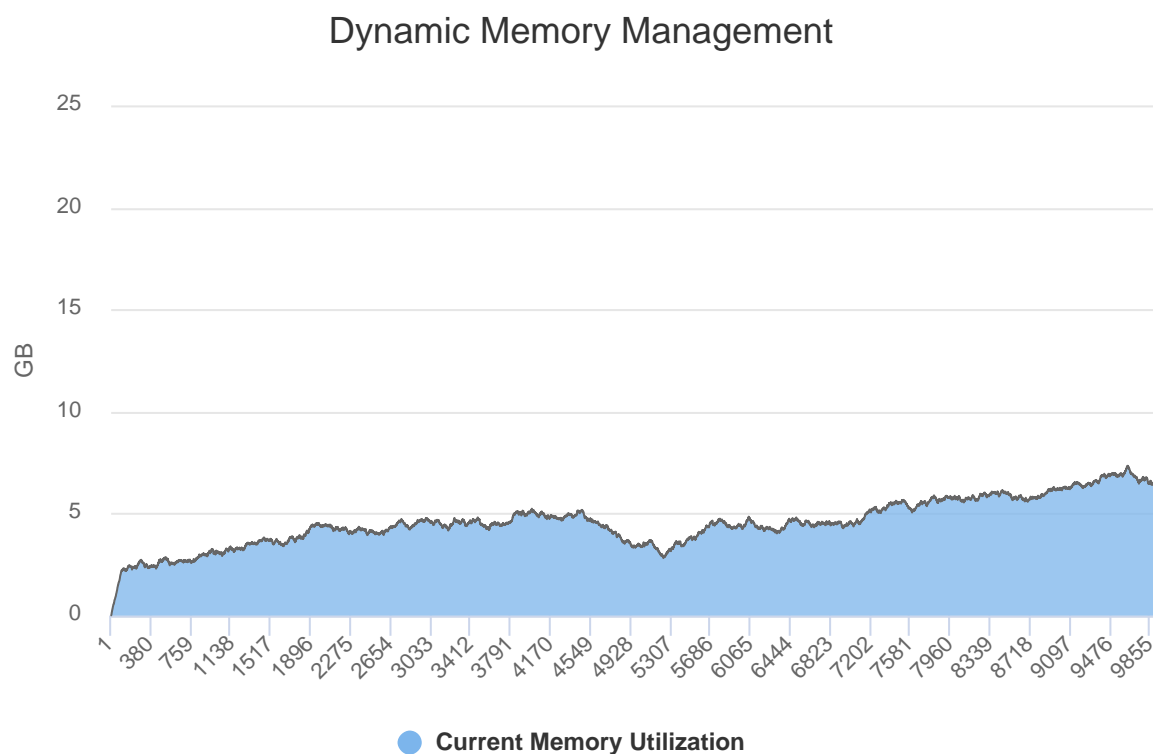
**Figure 7.2:** *Memory space organization in Java adapted from Honk (2014)*

of directives that have been executed frequently (Schildt, 2014; Oransa, 2014). However, both parts contain program code, just in different spaces (Schildt, 2014).

The second section of the memory of an application contains the data of the program (Forouzan, 2013). This section can be huge, depending on the amount of data an application uses, the interaction of the user with the system, and caching mechanisms, that for example, pre-load certain data from a disk (Forouzan, 2013). The capacity of this section is hard to estimate and manage because of the dependencies on the software code itself, the workload, the runtime configuration, and the resource environment. This becomes even more complex in distributed systems, as data is then shared, replicated, and synced amongst several instances. Our research focuses on this dynamic section of memory. In Java, this section is called Heap and again divided into smaller sub-sections: Eden, Survivor 1 & 2, and Tenured as depicted in Figure 7.2 (Schildt, 2014). The different sections correlate to the age of an object. The longer an object lives in Heap, the more likely it travels to the upper sections (Schildt, 2014).

Depending on the runtime environment, either dynamic memory management or automatic memory management is used. Dynamic memory management requires one to explicitly load and unload objects, which means allocating and releasing memory based on the program code. Runtime environments like C++ or older versions of the iOS operating system (OS) use this memory model to manage memory. Furthermore, modern runtimes like the JVM<sup>5</sup> allow to directly allocate memory even though the default mechanism is automatic memory management. Mistakes in the program code or unexpected behavior of the user can lead to memory leaks due to missing object releases. This model is, good

<sup>5</sup><https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html>

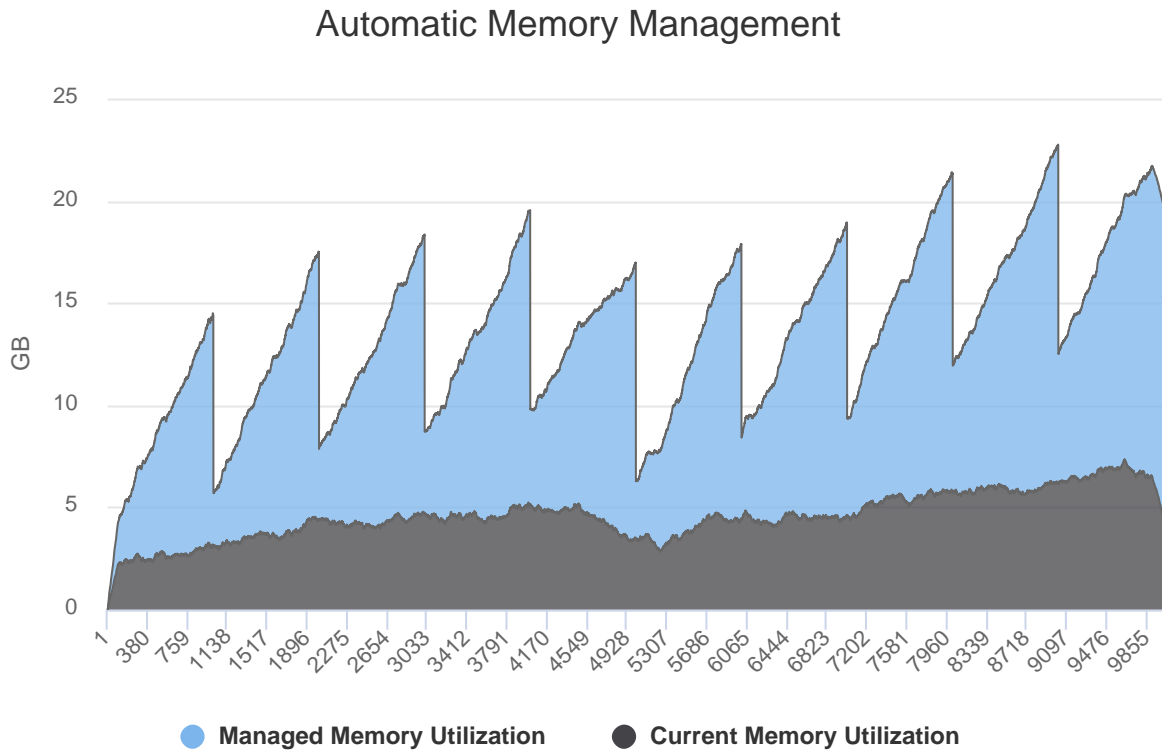


**Figure 7.3:** *Memory trace of a system using Dynamic Memory Management*

programming assumed, pretty efficient regarding the memory footprint, as memory is immediately released after an object is not needed anymore. Profiling an application using this memory model shows a lot of small spikes and a waveform, as depicted in Figure 7.3.

Automatic memory management in contrast organizes the memory in the runtime. Creating new objects automatically allocates memory. Frequently checks on the references of these objects are conducted and the object and the memory it occupies is released when no reference to the object remains. The algorithm class that checks and frees the memory automatically is called Garbage Collection (Schildt, 2014). Figure 7.4 shows a typical memory profile of an application using automatic memory management. A typical pattern is the triangular form. The flank of each spike marks the execution of GC. The real demand is lower, but as the GC runs only after certain thresholds are exceeded, we see a delay in freeing memory compared to dynamic memory management (Schildt, 2014). In Java, each GC run can promote objects to superior memory spaces (e.g., Eden to Survivor 1) (Schildt, 2014).

The more GC runs an object survives, meaning references to this object exist, the rarer the memory space of this object is checked (Schildt, 2014). Different GCs check different spaces and based on different thresholds. Besides freeing memory, a GC run also consumes CPU when computing which objects are releasable. The number of spaces, the type of GCs, the thresholds, and the CPU consumption depend on the runtime (e.g., Java vs. .NET), the runtime version (e.g., Java 7 vs. Java 8), the configuration, and the application itself, more specifically, its object structure. We propose a model for simulating automatic memory management based on an approximation of these parameters. The same model without GC behaviors is eligible for simulating dynamic memory management.



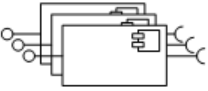

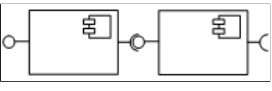

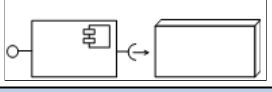
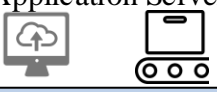
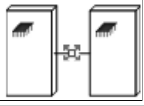



**Figure 7.4:** *Memory trace of a system using Automatic Memory Management*

### 7.3.2 Memory Meta-Model

Memory demands and simulations are more complex due to automatic memory management (Libiř et al., 2015). We chose an approach based on probabilities, as previously introduced techniques were too computationally intensive (Libiř et al., 2015). Furthermore, our approach allows one to discover and approximate automatic and dynamic memory management based on measurements and observations. Technically, our approach extends the PCM, which already supports a number of resources (CPU, HDD, network), and has an expressive workload and software architecture description (Becker/Koziolok/Reussner, 2009; Reussner et al., 2016). We generate and simulate complete applications in order to demonstrate our memory management simulation approach. This means we consider all four major resources (CPU, HDD, network, and memory) and map the different components of real applications to PCM elements. Figure 7.5 shows the different elements of PCM and examples of this representation in Java (EE) applications.

Our approach is based on the state of different memory spaces of the so-called *Heap*. We consider the following variables describing the current state of memory: *used heap*, *committed heap*, *maximum heap size*, *minimum/initial heap size*. The maximum and the minimum/initial heap threshold are configuration parameters of the containing runtime. They are defined during startup of the runtime and may only be changed when restarting the runtime (Blaschek/Lengauer, 2015). The committed heap size is the amount of memory between maximum and minimum that is currently available for object allocation unless the maximum capacity is reached. The value of this size is constantly changing depending on certain thresholds of the runtime but never exceeds maximum and never falls below the minimum size. The used memory is the amount of memory that is actually

	Software Engineering Concept	Palladio Component Model	Java
Workload	Number of Users and User Behavior	Usage Model 	Real Users or Virtual Users 
	Components and Operations	Repository Model 	Classes, EJBs, JSPs 
Resource Profile	Deployment Units	System Model 	JARs/EARs/WARs 
	Deployment Unit to Resource Container Relationship	Allocation Model 	Installation on Application Server 
Deployment Topology	Resource Container, Resources and Network Conn.	Resource Env. 	Container/Virtual Machine/Host 

**Figure 7.5:** *Software Engineering concepts of a distributed application, PCM elements, and equivalent Java components adapted from Willnecker/Krcmar (2016)*

occupied by objects. We observe the state of all *Heap* spaces in order to calculate the currently used memory as depicted in Equation 7.1. The sum of the filling level of each memory space results in the totally used memory.

$$usedMemory = \sum_{i=1} \frac{OccupationMemorySpace_i}{SizeMemorySpace_i} \quad (7.1)$$

We constantly simulate the used memory per space and thus the complete used memory. Our model allows one to detect individual thresholds per space but also reacts to global thresholds defined for the complete heap space, like the grow and shrink thresholds. The simulation reacts to these thresholds based on the currently used memory and increases the committed memory to always provide sufficient headroom for object allocations. All the above-mentioned variables are part of our memory model.

We extend two sub-models of PCM in order to model and simulate memory and GC behavior: The *Resource Environment* and the *Repository Model*. The *Resource Environment* contains all information about the server and network infrastructure of an application. We extend this PCM meta-model in order to add a memory resource representation, as depicted in Figure 7.6. This resource supports dynamic and automatic memory management scenarios including GCs. GCs delay the release of memory, leading to a larger memory utilization during runtime, as depicted in Figure 7.4. Thus, memory is more likely to become a bottleneck. We extended PCM to simulate this effect (Becker/Koziolek/Reussner, 2009; Reussner et al., 2016; Willnecker/Krcmar, 2016). We added two classes to the meta-model to support this behavior:

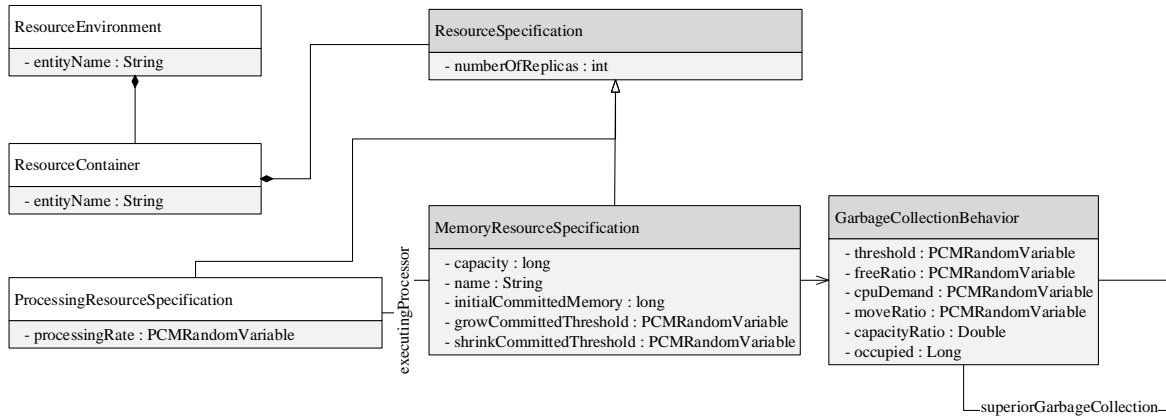
- (i) *MemoryResourceSpecification* to specify the attributes of a memory resource.
- (ii) *GarbageCollectionBehaviour* to define the behavior of automatic memory management. No behavior is specified in dynamic memory management scenarios.

Furthermore, we abstract the *ProcessingResourceSpecification* to an abstract *ResourceSpecification* from which *ProcessingResourceSpecification* and *MemoryResourceSpecification* are derived. This allows the model to simply add more resource types, by just deriving from the abstract base class, which also shares common attributes like the number of replicas of a resource.

In order to place demands on this newly introduced memory resource, we extend PCM *ResourceCalls*. We add two calls, one for allocating and one for releasing memory. These calls contain the number of bytes allocated or released by this call and an optional target name. The latter allows for distinguishing between different processes in the same application, which again have their own memory spaces.

Each runtime processing memory is defined by one *MemoryResourceSpecification*. At least one specification is necessary to specify memory demands in the PCM *Repository Model*. The separation of different memory spaces is defined by creating multiple *GarbageCollectionBehaviour* instances.





**Figure 7.6:** *PCM extension for memory resources*

Each memory occupying process (e.g., a JVM process) is defined by a *MemoryResourceSpecification*. This specification contains a *capacity* defining the total size of this memory resource. The memory specification is named so that it can be addressed individually from the *Repository Model* if multiple processes are simulated in parallel. We also define the *initialCommittedMemory*. This parameter defines the amount of committed memory at the beginning of the simulation. The maximum amount is defined by the *capacity*. The different memory spaces can be connected to promote a certain amount of objects from one memory space to another, which is typical for automatically managed memory runtimes like the JVM. We use the *SuperiorGarbageCollection* relationship between multiple *GarbageCollectionBehavior* objects to model these object movements. The *moveRate* defines which amount of memory is transferred from the lower heap space to the superior one each time a GC run is conducted.

We constantly track the *usedMemory* as defined in Equation 7.1. The *growCommittedThreshold* and *shrinkCommittedThreshold* variables are checked frequently and the committed memory is adjusted accordingly during simulation. These values are expressed as a percentage (default 40% and 70%). The filling level of the total heap is calculated by Equation 7.2. If the resulting value is below the shrink threshold or above the grow threshold, the committed memory is adjusted until the threshold is met again.

$$\text{committedFillingLevel} = \frac{\text{usedMemory}}{\text{committedMemory}} \quad (7.2)$$

A typical GC collects and stores released objects in different memory spaces (Libič et al., 2015). The spaces are cleaned in different intervals. For instance, the JVM executes two types of GCs (minor and major) to clean different spaces or promote objects to another space (Libič et al., 2015). A memory simulation containing GC requires the monitoring of GC events and the generation of instances of the memory resource and the GC behavior in PCM.

We define a *GarbageCollectionBehaviour* using five properties and one variable. The properties are specified before a simulation run and are thus part of the meta-model. The

variable is part of the simulation and changes constantly based on the actions conducted during a simulation run.

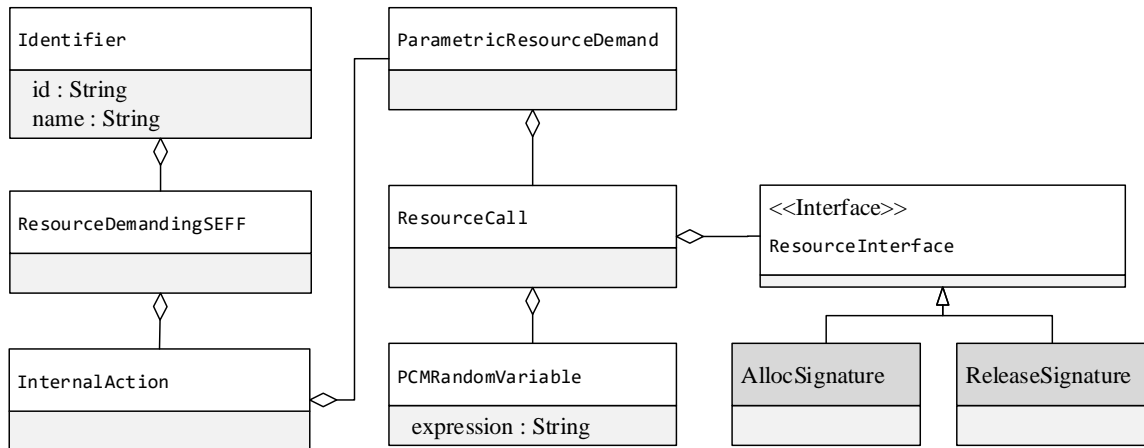
- (i) The *threshold* property defines the filling level of the current memory space that triggers a GC run.
- (ii) The *freeRatio* property marks the percentage of used memory in the current memory space that is released during a GC run.
- (iii) The *cpuDemand* property sets the resource demand per released byte placed on the related CPU resource. The relationship is defined by the parent *MemoryResourceSpecification* of the current *GarbageCollectionBehaviour*.
- (iv) The *moveRatio* property configures the percentage of used memory that is promoted to the superior memory space. Superior GC and thus memory spaces are defined by the *superiorGarbageCollection* relationship.
- (v) The *capacityRatio* property defines the amount of one memory space that is managed by this *GarbageCollectionBehaviour* instance. The total amount of all these properties of a *MemoryResourceSpecification* must add up to 100%.
- (vi) The *occupied* variable is used during simulation to track the currently allocated bytes in the memory space managed by this *GarbageCollectionBehaviour*.

We use the following Equation 7.3 to track the current filling level of a memory space. The filling level is frequently compared to the *threshold* property during simulation. The GC run is executed as soon as the filling level exceeds the defined *threshold* of the *GarbageCollectionBehaviour*.

$$memorySpaceFillingLevel = \frac{occupied}{capacityRatio * parentSpecification.capacity} \quad (7.3)$$

The meta-model instance properties are not constant throughout each runtime instance but depend on the application, its runtime, and its workload. Thus, our approach relies on observing these parameters automatically by conducting load tests. Afterwards, prediction for other workloads, deployment scenarios, and alternative resource environments (e.g., cloud vs. on-premise) are possible by adjusting model parameters. However, changes to the application requires one to observe and adjust certain sub-models (e.g., *Resource Environment*).

In order to access the newly introduced memory resource, we use *ResourceCalls* in PCM, as depicted in Figure 7.7. We added two *ResourceInterfaces* to execute allocation and release calls on the memory resource. In PCM, *ResourceCalls* use these interfaces to distinguish between multiple access methods of a resource (e.g., HDD read and HDD write). Our two new signatures are designed to access the *MemoryResourceSpecification*. The amount of bytes that is either released or allocated is defined by the *PCMRandomVariable* attached to the *ResourceCall* instance. This variable allows one to define constants but also functions like Gaussian distribution or probabilistic distribution functions. These *ResourceCalls* are attached to *InternalActions* defining the steps an operation executes. Each operation in the performance model can now use the *MemoryResourceSpecification* by using the *AllocSignature* to allocate and the *ReleaseSignature* to release memory again.



**Figure 7.7:** *PCM extension accessing the newly introduced memory resource*

In dynamic memory scenarios, this signature is called on each time memory is released. The amount of available memory is immediately increased by the value specified in the related *PCMRandomVariable*. Models using automatic memory management should never use the *ReleaseSignature*, as this is handled by the *GarbageCollectionBehavior*.

### 7.3.3 Observing memory demands and GC

In order to observe memory demands we use agent-based instrumentation on our SUT. The instrumentation measures the memory consumption of an operation or a service, depending on the granularity of the instrumentation. Several tools and techniques exist in academia and practice to extract this data from APM. We successfully demonstrated the use of the industry APM tool Dynatrace, the RETIT Java EE Agent, and the Performance Management Work Tools (PMWT) Java EE agent in previous research (Willnecker/Brunnert et al., 2015a; Willnecker/Krcmar, 2016; Kroß et al., 2016). This previous work showed that this instrumentation on operation level provides accurate data for performance model generation and has low impact on the runtime (Brunnert/Krcmar, 2017). The observer effects, altering the applications behavior by instrumenting, are very limited. Furthermore, the approach of the PMWT workbench measures the memory demand before and after an operation for a certain thread (Brunnert/Vögele/Krcmar, 2013). This allows us to detect the memory demand of a method. Negative demands lead to releasing memory in that case. For our research, we use an enhanced version of the RETIT Java EE agent (Brunnert/Krcmar, 2017), which contains our memory detection algorithm. These measurements result in a large number of observations as we get data for each invocation. Thus, we aggregate the measurements by calculating the mean memory demand of each operation.

Detecting memory demand of an operation is enough to simulate dynamic memory management behavior. With this detection algorithm, we can model memory allocations and releases and thus simulate such behavior. However, in automatic memory management

runtimes, no release of objects can be detected with this method. Releases are only conducted, when a GC run occurs (Libič et al., 2015).

GC runs occur when certain thresholds of the memory spaces are exceeded (Libič et al., 2015; Willnecker/Krcmar, 2016). This happens when too many objects or large objects occupy a certain memory space. The runtime frequently checks whether the conditions for executing a GC run are met and conducts the GC run afterwards. The timing of a GC run varies, depending on the runtime, the applied GC algorithm, the resource environment, and the application, or more specifically, its object structure. Our approach does not cover the specifics of a certain GC implementation, but observes and reasons a set of generic probabilities instead to detect the conditions of a GC run.

In order to conduct an accurate memory simulation, we measure the memory state of a runtime twice, right before and directly after a GC run. We collect the following metrics while the application is running and a load test is executed.

- (i) The type of garbage collection that is executed. For Java (EE) this is either a minor GC or a major GC. Other GC implementations or technologies can have different GC types.
- (ii) Size of total memory available in the JVM.
- (iii) Size of allocated memory for each memory space before and after the GC execution. This is a simplification of the actual mechanism, as we do not simulate object movements in the fine-grained GC spaces. Complex and fine-grained approaches have proven to be too computationally intensive (Libič et al., 2014). Our approach based on probabilities enables automatic memory management simulation with low overhead compared to complex object movement simulations (Libič et al., 2015; Willnecker/Krcmar, 2016).
- (iv) CPU execution time necessary to execute the GC.

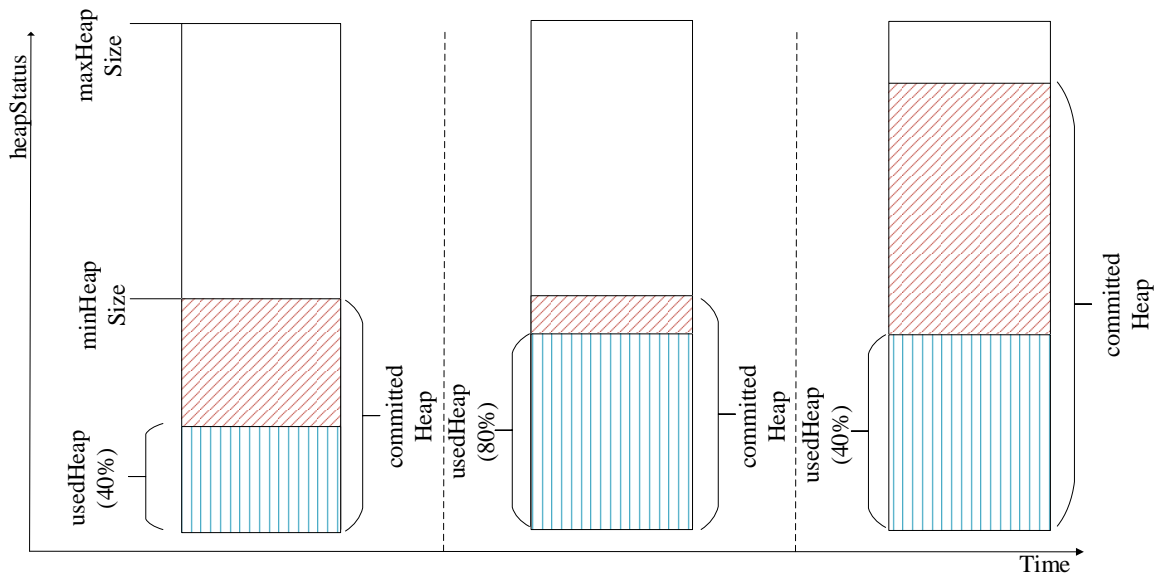
In Java (EE), we monitor the GC events of the running application using the `GarbageCollectorMXBean`<sup>6</sup> via Java Management Extensions (JMX) and the `PerfInstrumentation`<sup>7</sup>. We use the `MXBean` to intercept GC events and the amount of memory affected by such a run and the `PerfInstrumentation` to collect the amount of CPU ticks necessary to do so. Furthermore, as our approach is time based we have to convert the ticks to ms using the `Clock` class of `java.time`. Other runtimes provide similar detection mechanisms. It is also possible to just monitor the complete runtime and detect the timing of a GC run based on memory time series. A GC run must have occurred each time memory is released, as shown in Figure 7.4. However, this method can not distinguish reliably between different GC types unless each memory space can be observed individually.

The result of these measurements is a series of memory states before and after a GC run. We aggregate these measurements and calculate the probability distribution based on this data. We sort the size of each memory space before the GC run and build 5 quartiles with equal probability. This probabilistic approach approximates the actual thresholds of the runtime and also considers differences of the object space for each GC run. The five

---

<sup>6</sup><http://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/GarbageCollectorMXBean.html>

<sup>7</sup><https://github.com/frohoff/jdk8u-jdk>



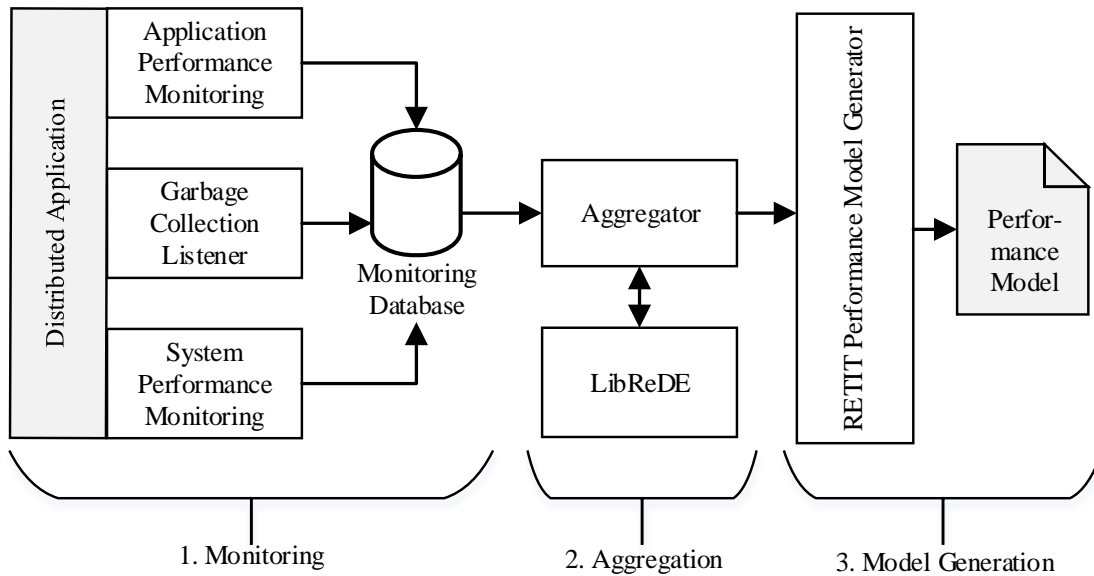
**Figure 7.8:** Example of growing committed memory using automatic memory management

quartiles are translated into a distribution function in PCM as a value for a GC threshold. The same procedure is conducted for each memory space in order to get the individual thresholds for each space and each GC behavior.

Furthermore, certain memory spaces (e.g., *Tenured* as depicted in Figure 7.2) are filled during a GC run. Long living objects move up the memory hierarchy when it is not possible to release the memory they occupy. For the JVM, this means that the memory spaces managed by the major GC are filled during a minor GC run. As we measure the type of a GC run and the state of the spaces before and after this run, we can detect such relationships. If a memory space that is not part of the current GC is filled during a GC run, we reason that the objects are moving from one lower memory space to a superior one. We sort the amount of bytes transferred from one space to another and calculate the probability distribution similar to the GC thresholds. Afterwards, we build a distribution function and store this function in the relationship between the two GC behaviors. This allows us to simulate object movements between multiple memory spaces.

Certain runtimes like the JVM allow one to grow and shrink the committed memory between a minimum and maximum value. The process of the runtime occupies either the maximum value or at least the committed memory. Starting such a runtime requires four parameters: *minHeapSize*, *maxHeapSize*, *shrinkThreshold*, *growThreshold*. The *minHeapSize* configures the heap size, when the runtime starts (e.g., `-Xms1G` sets the heap size to 1 Gigabyte (GB), when a JVM is started). The maximum size to which the heap can grow is set using the *maxHeapSize* parameter (e.g., `-Xmx4g` sets the maximum heap size 4 GB). The runtime cannot grow the committed memory to a larger amount. If more memory is required and the GC cannot clear enough memory, the runtime crashes.

The committed memory is initially set to the *minHeapSize* but might grow until the *maxHeapSize* is reached. The committed memory grows when the *growThreshold* is reached.



**Figure 7.9:** Performance model generation process adapted from Willnecker/Krcmar (2016); Brunnert/Krcmar (2017)

The committed memory grows until this threshold is met again. This threshold is usually a parameter of the runtime (e.g., `-XX:MaxHeapFreeRatio` sets this threshold in the JVM). The committed memory can also shrink when the used memory is below the *shrinkThreshold*. The committed memory is then reduced again until this threshold is met again. Similar to the *growThreshold*, this is a parameter of the runtime (e.g., `-XX:MinHeapFreeRatio` configures this threshold in the JVM). Figure 7.8 depicts an example of growing memory in a scenario with automatic memory management. It is sometimes recommended to set the minimum and maximum committed heap size to equal values, so that no growing or shrinking occurs. Our simulation reacts accordingly and does not conduct any growing or shrinking simulations in that case.

### 7.3.4 Memory model generation

The amount of data points, even when only conducting a short load test of about 10-15 minutes, usually exceeds  $10^8$  records. The amount of data is hard to process manually. We also consider CPU, HDD, and network demands, further increasing the amount of records that must be processed to transfer the observed application behavior into a consistent performance model (Willnecker et al., 2015b). Furthermore, creating performance models manually often outweighs the benefits of its capabilities, like predicting an application’s behavior (Kounev, 2005). Therefore, we reuse and extend a performance model generator called RETIT Capacity Manager, which processes the monitoring data automatically (Brunnert/Krcmar, 2017).

We cannot conduct memory simulations without a complete performance model consisting of a *Resource Environment* model, a workload model, and a model of the application itself.

The PMG used here creates a so-called *Resource Profile*. This intermediate model covers resource demands, focusing on CPU demands but also considering network, memory, and HDD, by aggregating APM measurement data. We inject our approach by adding GC measurements and the corresponding model generation to this PMG.

Model generation consists of three steps: (i) monitoring, (ii), aggregation, and (iii) model generation, as depicted in Figure 7.9.

The monitoring step collects operation invocations of the instrumented EA. We distinguish between resource demand measurement and resource demand estimation (Spinner et al., 2015; Willnecker/Krcmar, 2016). Resource demand measurement uses fine-grained monitoring data per operation invocation to measure the exact demand an operation places on a resource. These measurements can be collected with standard APM software like Dynatrace<sup>8</sup> Application Monitoring (AM), RETIT Java EE, or the PMWT Java EE agent (Willnecker/Krcmar, 2016). Resource demand estimation uses coarse-grained monitoring data like total resource utilization and response time series per operation and distributes the utilization throughout the operations (Spinner et al., 2015). Such coarse-grained resource utilization data can be collected using standard system monitors like System Activity Reporter (SAR), or monitoring and control interfaces of virtual machines like JMX. Load drivers like jMeter<sup>9</sup> or access logs of web servers provide response time series of operations invoked at the system-entry level. For more detailed (e.g., component-level) response time series, custom filters or loggers are necessary. Out of response times and total utilization we can estimate the resource demand using the Library for Resource Demand Estimation (LibReDE) (Spinner et al., 2015).

The PMG supports data from different data sources, as depicted in Figure 7.9:

- (i) Application Performance Monitoring for fine-grained application data. We use the RETIT Java EE Monitoring solution in this work. Previous work demonstrated the applicability of industry standard solutions like Dynatrace AM (Willnecker/Krcmar, 2016).
- (ii) System Performance Monitoring for coarse-grained application data. Standard system tools or custom host agents are possible. We used Apache Webserver<sup>10</sup> access logs, RETIT Host Monitoring and JMX in this and previous works (Willnecker/Brunnert et al., 2015a; Willnecker/Krcmar, 2016).
- (iii) Garbage Collection Listener is an extension created to capture GC runs or continuously monitor process memory to approximate GC runs. We use the JMX interface to capture GC events for Java (EE). Some APM solutions already support GC monitoring (e.g., Dynatrace). No separate listener is necessary in such cases.

The collected data is stored continuously in a monitoring database (DB) based on the Apache Cassandra<sup>11</sup> project including the extensions and base schema of the RETIT APM server. The large amount of data requires a scalable, yet simple DB structure. The main table of this DB covers the operation invocations and its resource demands. Each row in

---

<sup>8</sup><http://www.dynatrace.com/>

<sup>9</sup><http://jmeter.apache.org/>

<sup>10</sup><http://httpd.apache.org/>

<sup>11</sup><http://cassandra.apache.org/>

**Table 7.2:** *GC measurement data collection*

Agentname	Timestamp	RunID	Type	MemorySpace	BeforeRun	AfterRun	CPUTime
AWS_InsuranceProvider	1488133189679	33bc35e1-3...8	Minor	Eden	22347776	2752504	567
AWS_InsuranceProvider	1488133189679	33bc35e1-3...8	Minor	Survivor 1	11605168	3801088	567
AWS_InsuranceProvider	1488133189679	33bc35e1-3...8	Minor	Survivor 2	12314760	1650712	567
AWS_InsuranceProvider	1488132392158	f510b830-4...4	Major	Tenured	42834896	3801088	708

this table corresponds to an operation invocation or a measurement record from system monitoring. We add another table to collect GC monitoring data. Each GC run results in one record containing the state of the memory spaces before and after the GC run as well as the total CPU demand of the GC run. The next phase uses this monitoring DB as a single source of input.

Table 7.2 illustrates two GC runs clearing 4 different memory spaces. Each GC run has an individual *RunID* generated when a GC run is detected and the timestamp of the beginning of this execution. The example in Table 7.2 was collected running a Java EE server using the *Parallel Collector*<sup>12</sup> GC (Cooperation, 2016). This GC implementation executes minor and major GCs, while the minor cleans three memory spaces (Eden, Survivor 1 & 2) and the major only one (Tenured). We can distinguish between the different GC runs by the *Type* column and which space is effected by the *MemorySpace* column. We measure the state of each space before and after the GC run, depending on how much time of the CPU was consumed and the total time the GC run took.

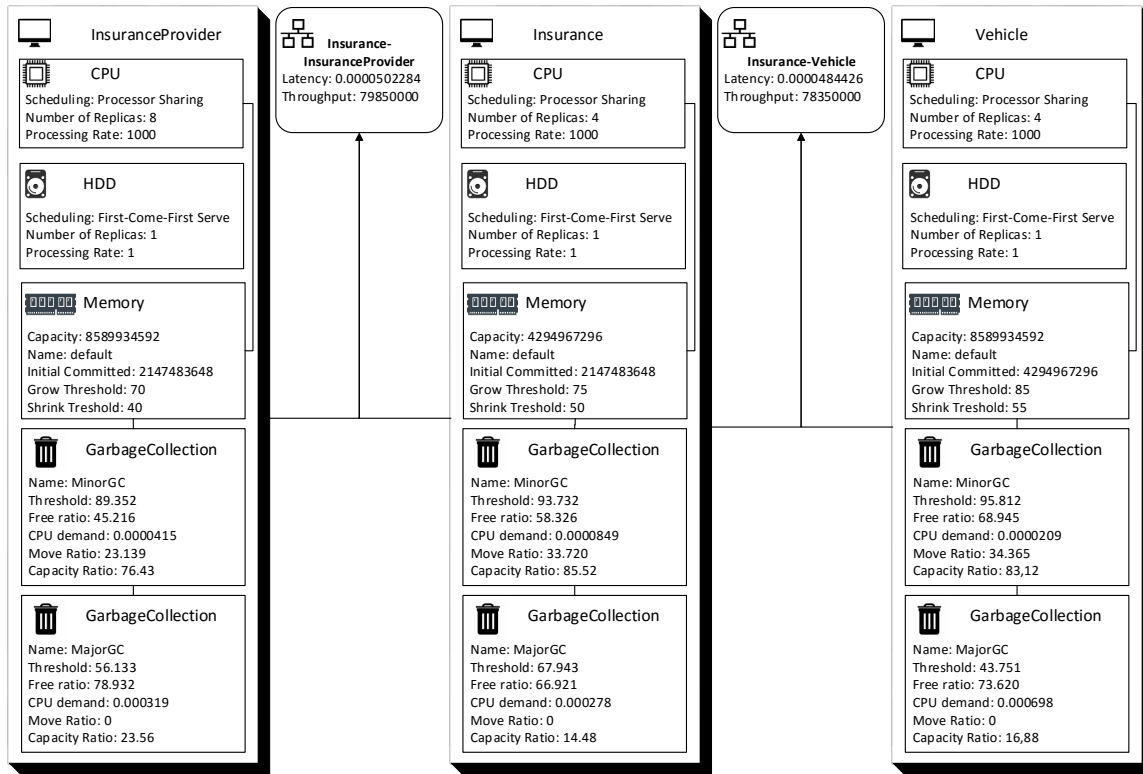
General information about the host and the processes are stored in another table in order to generate a *Resource Environment* model out of this. Information like the number of CPU cores and total memory capacity are monitored on each host and stored in this meta-table.

After the collection of APM data by executing a short load test (10-15 minutes), the collected measurement data is aggregated and processed for performance model generation, as depicted in Figure 7.9. The generator creates three sub-models: (i) the *Resource Environment*, (ii) the workload model, and (iii) the repository model. Furthermore, the PMG creates a system model and an allocation model to connect these three sub-models. The result of the generation process is an PCM instance that is ready for simulation using the Palladio-Bench with our extensions (Becker/Koziolek/Reussner, 2009; Reussner et al., 2016). The PMG uses a sub-model generator for each of the above-mentioned sub-models. We extend the generator for the resource environment and the repository model in order to integrate our memory model.

We extend the *Resource Environment* model generator to create at least one *MemoryResourceSpecification* instance per resource container. Multiple specification instances are possible if multiple processes on one host were monitored. This newly introduced resource sets properties for the initial and the maximum available memory. Furthermore, the grow and shrink threshold are created if the current runtime supports growing and shrinking committed memory.

<sup>12</sup><https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html>





**Figure 7.10:** Example of an automatic generated Resource Environment model for the SPECjEnterpriseNEXT EA

Our generator extension distinguishes between dynamic and automatic memory management. For dynamic memory management, no further generation is conducted and the *Resource Environment* model is finished. For automatic memory management, we extract all GC types and create a *GarbageCollectionBehavior* instance for each GC type. For simplification, we aggregate the memory spaces managed by one GC and consider this a single memory space. This reduced the amount of memory spaces in a JVM to two and in a .NET runtime to three spaces. The generator calculates the total memory space in proportion to the parent *MemoryResourceSpecification* and sets this value as *capacityRatio* for each GC behavior instance. Furthermore, the generator calculates, for each behavior instance, the mean CPU demand per byte that has been released during a GC run, the mean free ratio and the threshold leading to a GC execution. The threshold and free ratios are calculated in proportion to the committed memory. Therefore, the GC properties are automatically scaled to the capacity of the parent *MemoryResourceSpecification*. The generator finally calculates the *moveRatio* representing the amount of memory that is promoted to a superior memory space. This ratio is also calculated in proportion to the committed memory and is only calculated if a superior GC behavior exists, such as major, in contrast to minor, GC in the JVM. This model and generation implementation automatically adapts *GarbageCollectionBehavior* to other *MemoryResourceSpecification* representing larger or smaller runtime instances with less or more available memory.

Figure 7.10 depicts one generated *Resource Environment* model used for the SPECjEnterpriseNEXT EA. We use three application servers each running one JVM process and one deployment unit. The individual resource containers are linked via network. The network

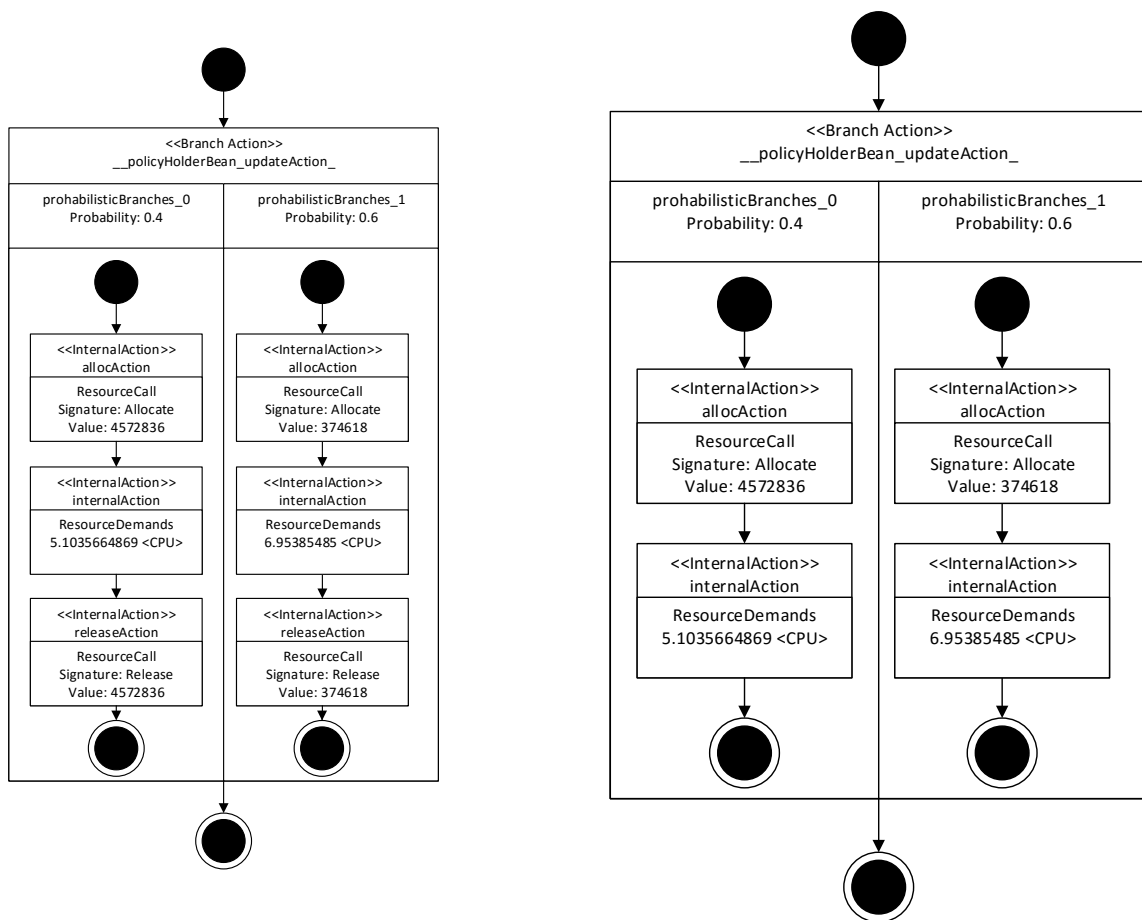
components have been benchmarked using *Imbch* in order to specify throughput and latency. Each resource container has a CPU with a different number of cores, an HDD, and a memory resource. In this case, the generation of the model has been conducted on the same machine that is simulated. Therefore, the CPU is set per default to a processing rate of 1000. Should the model predict the performance metrics on another machine, relative CPU capabilities must be calculated. In such a case, we use the SPEC CPU 2006. The memory resource in Figure 7.10 has two GC behaviors. One for minor and one for major GC in the corresponding runtime. The minor GC moves a certain amount of bytes per GC run to the major. The memory and GC values are approximated based on the GC measurements collected during a load test.

The sub-model generator for the PCM repository model requires an extension to support the newly introduced *MemoryResourceSpecification* and *GarbageCollectionBehavior*. Our model extension reacts to newly introduced signatures for *ResourceCalls* to allocate or release memory. The sub-model generator already supported rudimentary memory management by using *PassiveResources* (Brunnert/Krcmar, 2017). This approach calculated the mean amount of memory used per operation. We adapt this approach but use our newly introduced *ResourceCalls* signatures and equal distribution function to allocate the amount of memory. We add *InternalAction* using a *ResourceCall* with an *AllocSignature* at the beginning of each operation. We calculate  $n$  equally distributed quantiles by sorting all allocation measurements by the number of bytes, thus allocated objects, and selecting each  $n^{\text{th}}$  quantile. We add all quantiles with an equal probability to the corresponding *PCMRandomVariable* as a distribution function, as depicted in Equation 7.4.

$$P(1/n)_{(i)} = \text{quantile}_{(i)} \quad (7.4)$$

If dynamic memory management is used, we add another *InternalAction* representing release calls using the same approach to calculate the demand. For automatic memory management, the free calls are handled by the simulated GC runs. Therefore, memory is only released when a GC run is executed leading to the typical triangular memory profile, as depicted in Figure 7.4.

During simulation, these demands are evaluated, meaning the distribution functions are executed and the resulting allocation or release calls are forwarded to an instance of the *MemoryResourceSpecification*. Based on the signature, either the amount of available memory is reduced (allocation) or increased (release). Probes are spawned containing the current state of the resource for the evaluation framework of the simulation. Additionally, for automatic memory management, a thread per memory resource is started with the simulation start. This thread monitors the committed memory of the corresponding resource and checks if it exceeds one of the GC thresholds. If the committed memory exceeds the GC execution threshold, a GC run is simulated. The memory of this resource is released depending on the free ratio of the executed *GarbageCollectionBehavior*. Memory is moved from one memory space to another depending on the *moveRatio* and a CPU resource demand, depending on the number of bytes releases, and the GC behavior instance is placed on the CPU resource connected to the current *MemoryResourceSpecification* instance. Furthermore, the committed memory of each space grows or shrinks depending on the grow and shrink threshold and the current state of a memory space.



(a) RDSEFF representation using dynamic memory management

(b) RDSEFF representation using automatic memory management

**Figure 7.11:** *PCM RDSEFF representation of memory demands*

Figure 7.11 shows the same operation for dynamic and for automatic memory management. The first example Figure 7.11(a) uses dynamic memory management. A certain amount of bytes is allocated in the beginning of the operation using a *ResourceCall* and the *Allocate* signature. Afterwards, a CPU demand is placed, taking some time for execution. After the CPU execution, another *ResourceCall* is used to release the bytes allocated in the beginning of the operation. We use a simple mean value (calculating a single percentile using Equation 7.4) to reduce the complexity of the example. The next example Figure 7.11(b) shows the same operation using automatic memory management. In this example, no *Release* signature for a *ResourceCall* is added, but the GC behaviors of Figure 7.10 will release memory after the observed and defined threshold are met.

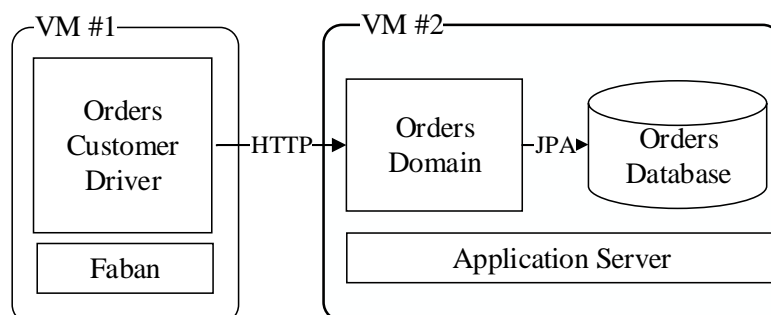
### 7.3.5 Limitations

Although we thoroughly conducted multiple experiments, different scenarios, and settings, our probabilistic approach has certain limitations. We use an abstraction based on measurements and observations of the actual behavior. This means that changes in the software, the parameters of the runtime, especially of the GC, and the resource environment influence the accuracy of our prediction.

In order to transfer our model to another resource environment (e.g., from on-premise to IaaS cloud provider), calibration must be conducted. This means the CPU, HDD, and network capabilities must be measured for each new resource container (e.g., bare-metal, Virtual Machine (VM), container) type (Brunnert/Krcmar, 2017). Otherwise, the performance metrics diverge from the actual behavior, and memory allocations or releases occur faster or slower (depending on the server) than simulated. However, this limitation is inherited by the PMG and easily resolved by conducting benchmarks like SPEC CPU 2006. Such a calibration is highly automated and takes only a couple of minutes per resource container. Once a certain container is calibrated, the data can be reused until changes to the hardware occur. Furthermore, such calibrations are only applied to the affected sub-model. In this case the resource environment model.

Our approach assumes that regular GC events occur and are able to release a stable amount of bytes. If the application contains errors, such as memory leaks, that prevent the GC from running or releasing objects, we are not able to detect this. Specialized approaches like HORA deal with the detection of memory leaks and are better suited for such a use case (Pitakrat et al., 2017). However, our approach is still able to detect increased memory usage, extensive GC runs, and corresponding CPU usage.

Some GC implementations can stop the complete runtime for a short amount of time to clean certain memory spaces. In such cases, the complete execution of all threads is suspended. The runtime can run in such scenarios for the major GC. We did not include this into our simulation approach, as this usually has little effect on the overall performance. The runtime tries to avoid this as often as possible and even when it occurs, it limits the suspension time to an absolute minimum (Libič et al., 2014). Depending on the measurement approach, it is sometimes not even possible to detect a stop at all (Libič et al., 2014).



**Figure 7.12:** *SPECjEnterprise2010 Orders Domain as an example EA*

Allocating and releasing memory consumes time just like CPU or HDD operations. The time taken for such operations is usually very small, which is why we decided to simplify our approach, therefore simulating these operations is instantly executed without any delay.

## 7.4 Evaluation

In this section, we present our evaluation starting with our experimental setup, followed by the evaluation process and the results. This section concludes with a discussion on these results.

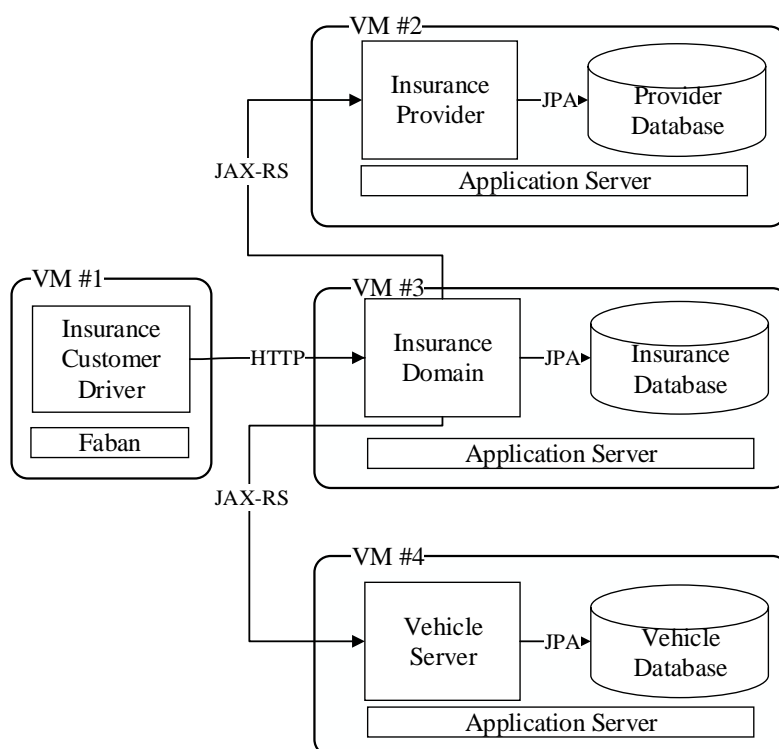
### 7.4.1 Experimental Setup

We use two different EAs to evaluate our approach. The so-called Orders domain of the SPECjEnterprise2010 benchmark and the Insurance-Domain of the SPECjEnterpriseNEXT benchmark. These benchmarks use most aspects of the Java EE specifications and are rather complex. Therefore, we use these industry benchmark applications to evaluate our research.

The *Orders Domain* of the 2010 benchmark is a classical monolithic application that allows one to browse and purchase cars. It consists of one deployment unit and one DB, as depicted in Figure 7.12. The benchmark also provides a load test and corresponding driver. This benchmark was originally designed to benchmark a Java EE application server. Our deployment uses an integrated DB based on Apache Derby<sup>13</sup> to increase the amount of processing and memory in the JVM.

The SPECjEnterpriseNEXT industry benchmark is the successor of the SPECjEnterprise2010 benchmark. Like the 2010 version, this is a Java EE application typically used

<sup>13</sup><https://db.apache.org/derby/>



**Figure 7.13:** *SPECjEnterpriseNEXT Insurance Domain as example EA*

to benchmark the performance of different Java EE application servers. We use a pre-release version<sup>14</sup> of the SPECjEnterpriseNEXT as an example EA for our evaluation. This benchmark mimics an insurance policy management system for car insurances. It consists of three different service components (*Insurance*, *Vehicle*, and *Insurance Service*) and three DBs (*Insurance*, *Vehicle*, and *Provider DB*) as depicted in Figure 7.13. We again use Apache Derby as an internal DB. This increases the amount of memory necessary for running the application.

Both benchmarks contain a load driver emulating customers. The *Orders Customer Driver* and the *Insurance Customer Driver* are based on Faban<sup>15</sup> and execute different business transactions. The transactions are conducted by sending Hypertext Transfer Protocol (HTTP) requests to the *Order Domain*, respectively, to the *Insurance Domain*. For the SPECjEnterpriseNEXT further HTTP calls using JAX-RS<sup>16</sup> REST are triggered and sent to the other two services. The DBs are connected using the Java Persistence API (JPA) protocol.

<sup>14</sup>version from 19.02.2016

<sup>15</sup><http://faban.org/>

<sup>16</sup><http://jax-rs-spec.java.net/>

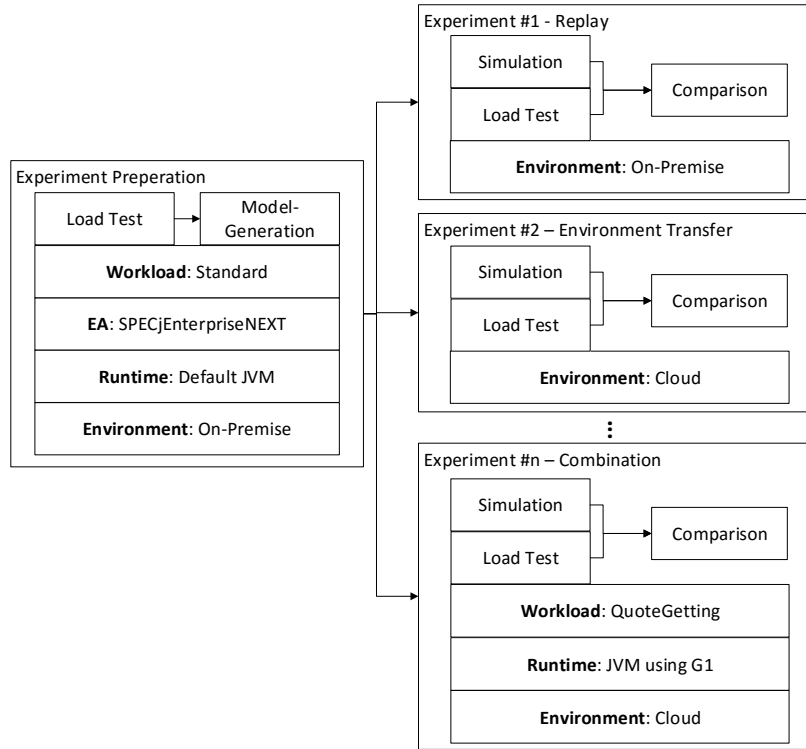


Figure 7.14: *Experiment design to evaluate memory model accuracy*

## 7.4.2 Evaluation process

We evaluate our approach using multiple simulation runs and comparing the results with actual executions of the applications. Our variables are as follows:

- (i) **Application:** Both applications presented in Section 7.4.1 are used to evaluate the approach. The SPECjEnterprise2010 is used as a monolithic application and the SPECjEnterpriseNEXT is used as an EA following the microservice approach (Fowler/Lewis, 2014).
- (ii) **Resource Environment:** We use different resource environments to show the feasibility of our approach. We execute the benchmark applications in a lab scenario on-premise and use AWS EC2 as an industry IaaS cloud provider.
- (iii) **Workload:** We change the workload executed by the load driver. This shows the robustness of our approach when workload changes occur or the tested/assumed workload diverges from the real workload.
- (iv) **JVM parameter:** GC is very dependent on the parameters that configure the runtime. We alter parameters and show that our prediction accuracy does not suffer from these changes.

We conduct a series of controlled experiments varying these parameters (Hevner et al., 2004). We prepare the experiments by conducting a load-test in our on-premise lab environment, as depicted in Figure 7.14. The application server is instrumented using the RETIT Java EE Agent and hosts either the SPECjEnterprise2010 or the SPECjEnterpriseNEXT EA. The measurement data is collected and stored for model generation,

**Table 7.3:** *Software and hardware configuration for model generation*

Server	Driver	Application Server 1	Application Server 2	Application Server 3
Application Server	Faban 1.3.0	JBoss Wildfly 8.1.0 Final		
Java Virtual Machine	Oracle JDK 1.7.0.79			-
Operating System	openSUSE Leap 42.1 (x86_64)			
CPU Cores	4 vCores (2.1 Gigahertz (GHz))	8 vCores (2.1 GHz)		
Memory	8 GB	16 GB		
Host System	IBM System X3755M3			
Network	1 Gigabit-per-second (GBit/s)			

**Table 7.4:** *List of all conducted experiments*

Exp. No	Application	Workload	Runtime	Environment	Description
1	SPECjEnterprise2010	Standard	Default JVM	On-Premise	2010 Replay
2	SPECjEnterprise2010	Standard	Default JVM	Cloud	2010 Resource Transfer
3	SPECjEnterprise2010	Standard	G1	Cloud	2010 G1 Garbage Collector
4	SPECjEnterprise2010	MultiPurchase	G1	Cloud	2010 Alternate Workload
5	SPECjEnterpriseNEXT	Standard	Default JVM	On-Premise	NEXT Replay
6	SPECjEnterpriseNEXT	Standard	Default JVM	Cloud	NEXT Resource Transfer
7	SPECjEnterpriseNEXT	Standard	G1	Cloud	NEXT G1 Garbage Collector
8	SPECjEnterpriseNEXT	QuoteGetting	G1	Cloud	NEXT Alternate Workload

as described in Section 7.3.4. The model is dependent on the environment, the runtime parameters, and the workload. The core part of the model is the *Repository Model* that depends on the EA. We create one model per EA. The other variables are changed in the model, e.g., by exchanging the *Resource Environment* model. To show the robustness of our approach, we deploy the EA in another environment, exchange the sub-model describing the resource containers, and execute a load test. The results of the load tests and of the simulation of the altered model are compared (Kroß et al., 2016). We present the accuracy of the four major resources CPU, HDD, memory, and network to demonstrate the accuracy of our approach and the RETIT tools and especially of the memory model and simulation. The comparison is automated using PET (Kroß et al., 2016).

The first experiment just covers a replay, where all variables conform to the model generation setup. We do this to validate the model generation. We continue by altering one variable after another, starting with the *Resource Environment* model. The last experiment is a full combination of all variables, demonstrating that workload, runtime, and environment can be exchanged without harming the prediction quality of our model and simulation. We conduct both experiment runs with both EAs to show that the memory model only depends on actual software but is robust to other influencing factors, as long as they are correctly specified in the corresponding sub-model. The environment we used for model generation is described in Table 7.3. We used up to three identical application servers for this generation. Only the first one was used for the SPECjEnterprise2010 application, as it covers only one component and all three for the SPECjEnterpriseNEXT benchmark.

In total, we conducted 8 experiments covering all 4 variables and combinations of those. Table 7.4 presents all experiment configurations. Experiment 1 and 5 are replay scenarios in which the model generation and the simulation and comparison environment were equal.



**Table 7.5:** *Measurement and simulation results for SPECjEnterprise2010 accessed by 200 concurrent users*

Resource	Metric	Replay	Cloud	G1	Alt. Workload
CPU	Mean measured utilization	73.72%	61.34%	58.23%	63.36%
	Mean simulated utilization	69.34%	56.33%	54.21%	56.46%
	Relative error	5.94%	8.17%	6.90%	10.89%
Memory	Mean committed memory	3.55 GB	3.63 GB	3.36 GB	3.63 GB
	Mean simulated memory	3.79 GB	3.93 GB	3.51 GB	3.92 GB
	Relative error	6.91%	8.26%	4.65%	7.96%
HDD	Mean measured demand	1.30%	1.59%	1.19%	2.09%
	Mean simulated demand	0.92%	1.09%	0.99%	1.56%
	Relative error	29.23%	31.45%	16.81%	25.36%
Throughput	Mean transactions per minute	409.30	431.34	455.87	419.98
	Mean simulated transactions per minute	399.19	403.25	421.85	390.98
	Relative error	2.47%	6.51%	7.46%	7.14%

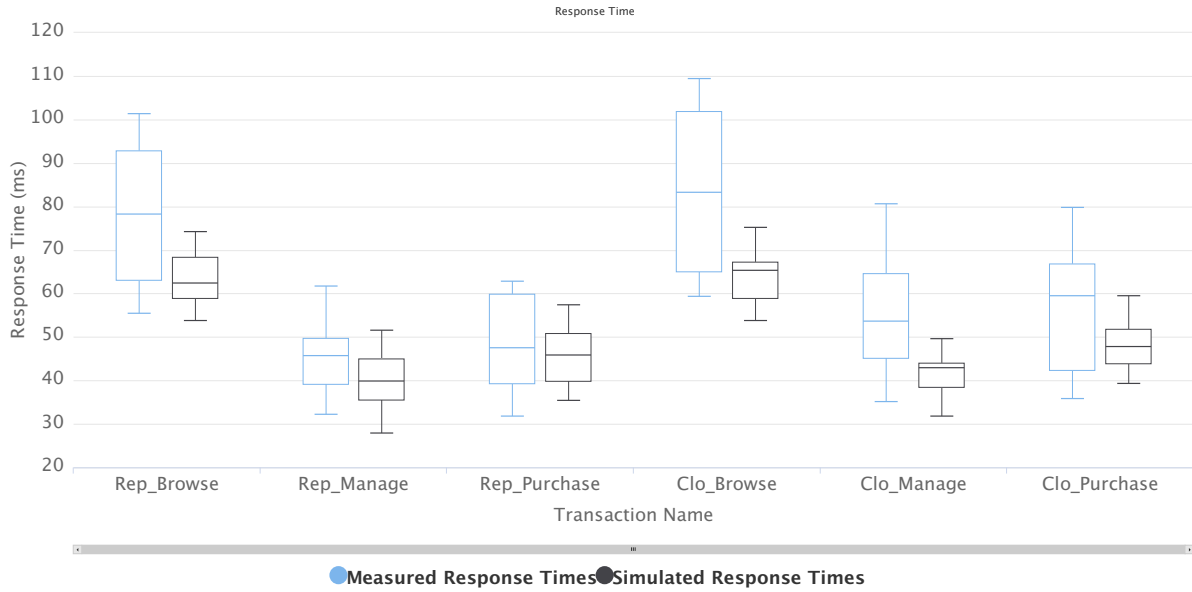
### 7.4.3 Evaluation Results

**Experiment 1 - SPECjEnterprise2010 Replay** For our first experiment we used the monolithic SPECjEnterprise2010 in an on-premise scenario. We used 200 concurrent users for this and the three following scenarios. Previous research demonstrated that up-scaling (e.g., from 200 to 600 users) is already possible with accurate results (Willnecker/Krcmar, 2016). Therefore, we concentrated on alternating other factors in the following experiments.

We collected fine-grained monitoring data for the model generation and afterwards simulated the resulting model. The simulation data was compared to coarse-grained memory data like CPU utilization to evaluate the accuracy of our model and simulation. We call this experiment the SPECjEnterprise2010 Replay scenario. The measurements and comparisons with the simulation results for CPU, HDD, memory, and throughput are depicted in Table 7.5.

The relative error is usually below 10%. Compared to previous research without any memory simulation, this is an improvement of at least 5% points or an reduction of the relative error by 33% (Willnecker/Dlugi et al., 2015c). Furthermore, the models and the corresponding simulation results are more accurate due to the fact that memory was evaluated. Only the HDD resource has a large relative error of up to 30%. The EA used here has practically no HDD usage and thus the total utilization is below 3%. Even though our prediction error is within 1% range, the relative error is quite high due to the low utilization.

The response times of this and the following experiment are depicted in Figure 7.15. We collect the response times per business transaction. SPECjEnterprise2010 has three of these transactions: Browse, Manage, and Purchase. The box-plots in this figure present measured and simulated response times next to each other. The left box-plot of each pair always represents the measured response times. For the replay scenario, the relative error is between 3.5% (Purchase) and 20.5% (Browse). The Interquartile range (IQR) in the measured response times is higher, as we work with mean values and distributed functions that only approximate the real resource demands. However, all predictions are



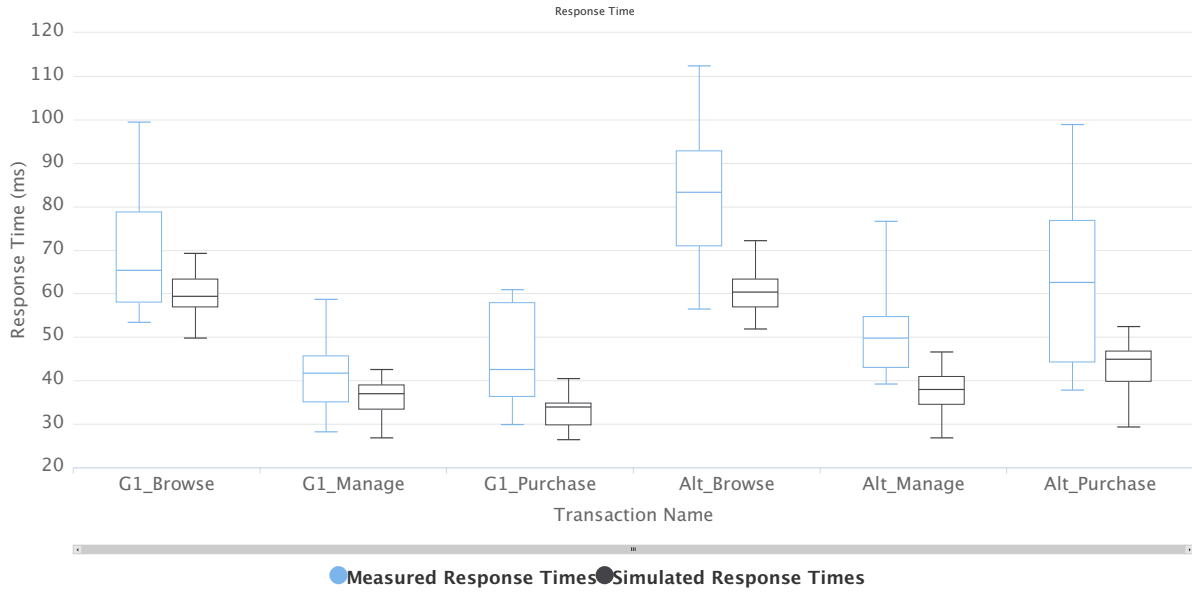
**Figure 7.15:** Response times SPECjEnterprise2010 Experiment 1 (Replay) & 2 (Cloud)

within close range to the actual response times. The simulation of a 15-min time frame takes less than a minute using EventSim on a laptop using an Intel I7-3520M CPU running at 2,90 GHz and requires about 2 GB of RAM (Merkle/Henss, 2011).

**Experiment 2 - SPECjEnterprise2010 Cloud** We modified the *Resource Environment* model to predict resource utilization and response times for a cloud deployment and compared the results of the simulation with a measurement run. Therefore, we installed the SPECjEnterprise2010 on an AWS EC2 host. We calculated the relative CPU speed compared to our on-premise deployment and the latency and bandwidth of the network connection to the EC2 host. The comparison of the resource utilization is presented in Table 7.5 and the response times are depicted in Figure 7.15. The prediction error for the resources was always below 10%, with the exception of the HDD with its very low utilization. The relative prediction error for the response times was at around 20%.

**Experiment 3 - SPECjEnterprise2010 G1** We altered the settings of our application server of the EC2 host to use the *G1* garbage collection algorithm of the JVM instead of the *Parallel Collector*, which is currently the default for Java server instances. We conducted two runs, one for monitoring the GC behavior and one for reasoning the thresholds of the GC behavior, which moves objects between memory spaces, free ratio per GC run, and the amount of CPU necessary to process a GC run. Afterwards, we altered our performance model by applying these variables in our *Resource Environment* model. The second run only used coarse-grained monitoring and the results were compared to a simulation run with the altered performance model.

The results of the resource utilization are depicted in Table 7.5, while the response times are presented in Figure 7.16. The prediction error for the resources was generally better than for the previous experiment. We did not conduct a GC monitoring for experiment #2. The transfer to another host slightly alters the GC behavior and was thus not



**Figure 7.16:** *Response times SPECjEnterprise2010 Experiment 3 (G1) & 4 (Alternate Workload)*

represented in the model of experiment #2. The response time error was between 10% and 20%, which is also an improvement compared to the results of experiment #2.

**Experiment 4 - SPECjEnterprise2010 Alternative Workload** The final experiment using the SPECjEnterprise2010 uses another workload distribution. Usually 50% of the transactions were Browse transactions, 25% Manage and 25% Purchase transactions. We changed to 70% Browse, 20% Manage, and 10% Purchase. The *Usage Model* and the load driver were altered to reflect these changes. We conducted a monitoring run and a simulation after applying these changes.

The results of this experiment are presented in Table 7.5 and Figure 7.16. The prediction error of the CPU utilization increased by 4% compared to the former experiment. The response time prediction error was between 23% and 28%.

**Experiment 5 - SPECjEnterpriseNEXT Replay** We continued our experiments by altering the EA from a monolithic to a distributed application. The SPECjEnterpriseNEXT industry benchmark is a distributed EA consisting of three components amongst which two are services. As with the SPECjEnterprise2010 benchmark, we installed these components on three application servers in our on-premise environment. Afterwards, we started a load test and applied a fine-grained monitoring to generate a performance model of this application. In a first step, we validated our model in a replay scenario. We applied the same load-test but only collected coarse-grained monitoring data and compared it with a simulation of the aforementioned generated model. The results of this comparison are listed in Table 7.6.

The relative prediction error is generally below 10% and often below 5% with just the exception of the HDD resource. This resource is utilized seldomly, making a small utilization error (<1%) a large relative error (16%-24%). The response time, depicted in

**Table 7.6:** *Measurement and simulation results for SPECjEnterpriseNEXT in an on-premise environment accessed by 140 concurrent users*

Resource	Metric	Application Server (AS)		
		Insurance Domain	Insurance Provider	Vehicle Service
CPU	Mean measured utilization	51.42%	34.54%	29.31%
	Mean simulated utilization	48.94%	31.91%	27.46%
	Relative error	4.82%	7.61%	6.31%
Memory	Mean committed memory	6.15 GB	1.43 GB	2.39 GB
	Mean simulated memory	6.31 GB	1.54 GB	2.59 GB
	Relative error	2.73%	7.68%	8.15%
HDD	Mean measured demand	2.79%	1.02%	2.15%
	Mean simulated demand	2.32%	1.02%	1.94%
	Relative error	16.85%	23.88%	10.23%
Throughput	Mean transactions per minute	313,42%	-	-
	Mean simulated transactions per minute	298,82	-	-
	Relative error	4.66%	-	-

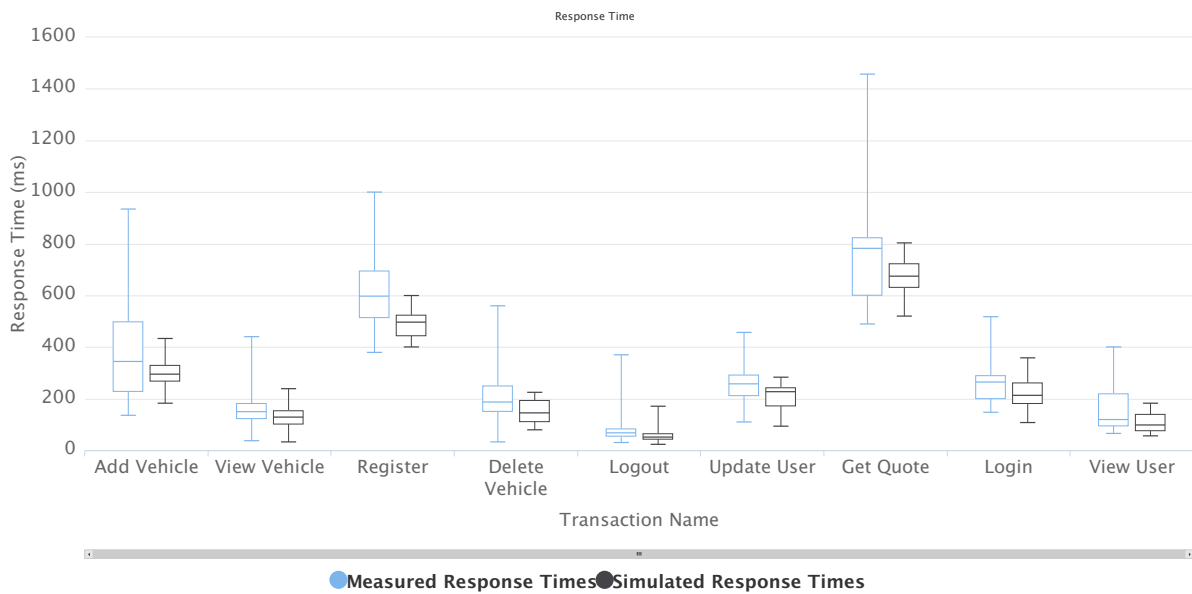
**Figure 7.17:** *Response times SPECjEnterpriseNEXT Experiment 5 (Replay)*

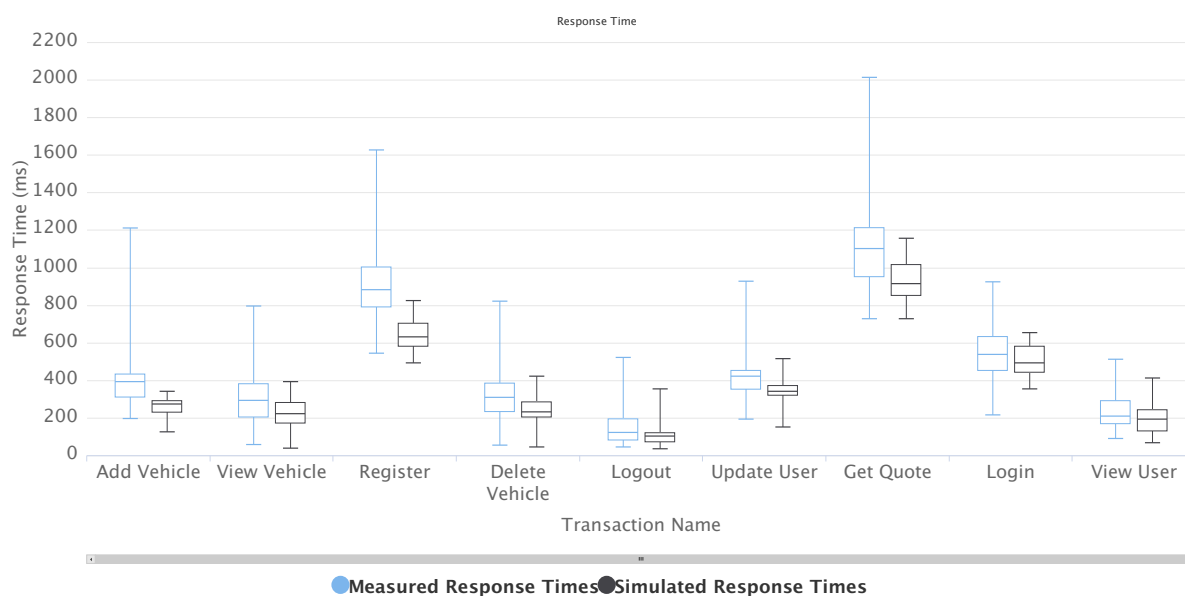
Figure 7.17 are below 20% for most of the 9 transactions. Only *Delete Vehicle* and *Logout* have an error of above 20%. Both transactions are relatively fast, which results in a median prediction error of between 15 and 30ms and thus still very close to the actual system behavior.

**Experiment 6 - SPECjEnterpriseNEXT Cloud** Afterwards, we installed the benchmark on AWS EC2 hosts and altered the *Resource Environment* model according to the previously conducted calibration. Again, we conducted a load test using coarse-grained monitoring and compared it to the simulation of the altered performance model. The results of this experiment are listed in Table 7.7.

The relative error increases by about 5% compared to the replay scenario due to the transfer and the coarse-grained calibration. The prediction error is below 15% with the exception of the HDD resource due to the low utilization. The response times in general went up due to larger latency from our load driver to the EC2 hosts. Also, the prediction error for the response times increased by about 5%, as shown in Figure 7.18.

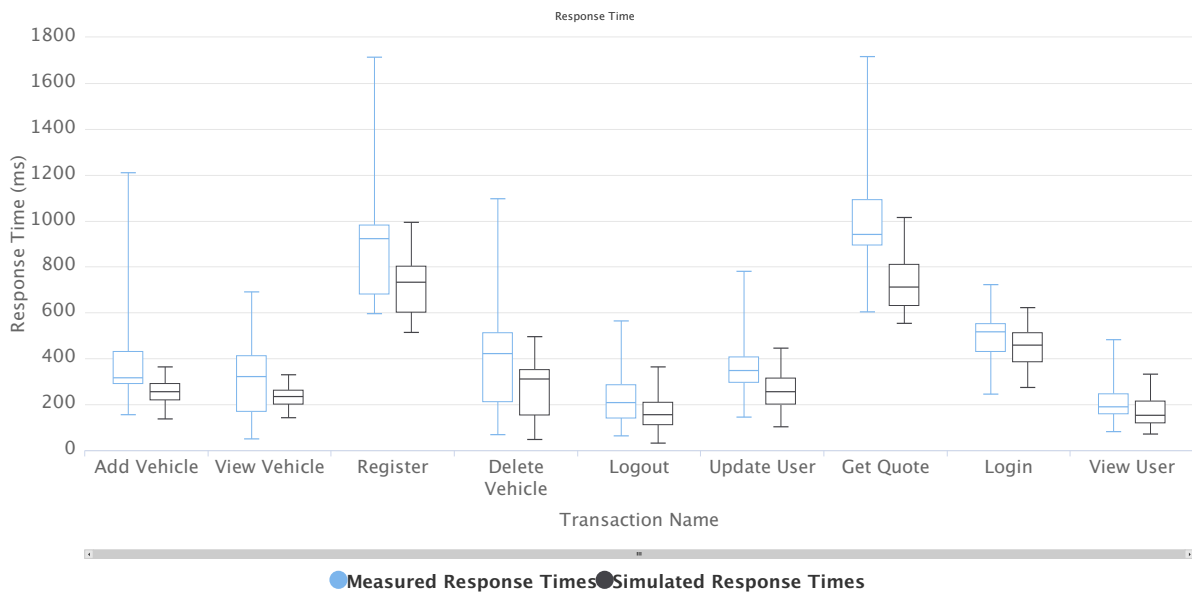
**Table 7.7:** Measurement and simulation results for SPECjEnterpriseNEXT in a cloud environment accessed by 140 concurrent users

Resource	Metric	Application Server (AS)		
		Insurance Domain	Insurance Provider	Vehicle Service
Deployment	-			
CPU	Mean measured utilization	45.43%	28.92%	25.04%
	Mean simulated utilization	41.24%	24.83%	21.73%
	Relative error	9.22%	14.14%	13.22%
Memory	Mean committed memory	6.32 GB	1.64 GB	2.49 GB
	Mean simulated memory	6.59 GB	1.41 GB	2.62 GB
	Relative error	4.40%	14.06%	5.22%
HDD	Mean measured demand	2.32%	1.19%	2.43%
	Mean simulated demand	1.89%	1.19%	2.43%
	Relative error	18.53%	29.41%	20.99%
Throughput	Mean transactions per minute	378,39%	-	-
	Mean simulated transactions per minute	330,72	-	-
	Relative error	12.60%	-	-

**Figure 7.18:** Response times SPECjEnterpriseNEXT Experiment 6 (Cloud)

**Table 7.8:** *Measurement and simulation results for SPECjEnterpriseNEXT using the G1 GC accessed by 140 concurrent users*

Resource	Metric	Application Server (AS)		
		Insurance Domain	Insurance Provider	Vehicle Service
Deployment	-			
CPU	Mean measured utilization	48.98%	30.62%	31.21%
	Mean simulated utilization	42.69%	27.03%	26.32%
	Relative error	12.84%	11.72%	15.67%
Memory	Mean committed memory	6.14 GB	1.43 GB	2.49 GB
	Mean simulated memory	6.43 GB	1.58 GB	2.64 GB
	Relative error	4.67%	10.41%	5.90%
HDD	Mean measured demand	2.48%	1.23%	2.91%
	Mean simulated demand	1.82%	0.93%	2.01%
	Relative error	26.61%	24.39%	30.93%
Throughput	Mean transactions per minute	354.43%	-	-
	Mean simulated transactions per minute	324.51	-	-
	Relative error	8.44%	-	-

**Figure 7.19:** *Response times SPECjEnterpriseNEXT Experiment 7 (G1)*

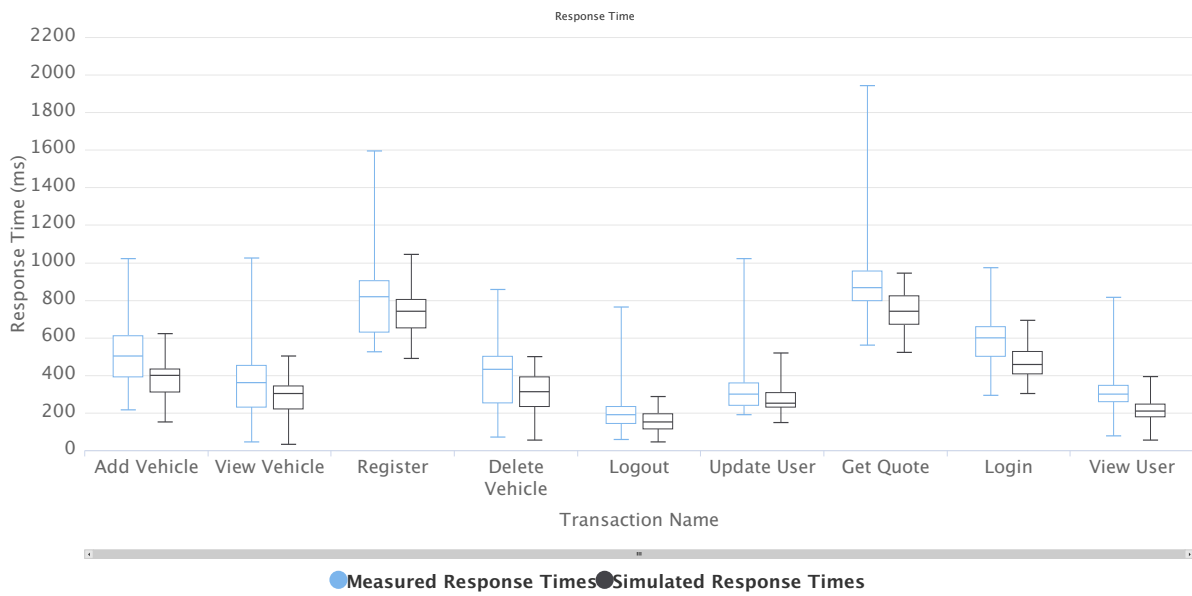
**Experiment 7 - SPECjEnterpriseNEXT G1** We changed the GC algorithm to the newest *G1* algorithm in all three JVMs running our three components. We conducted a monitoring run to collect GC behavior data and calibrate the *Resource Environment* accordingly. Finally, we conducted a load test with only coarse-grained monitoring and compared the results to the simulation of the altered performance model. The results are presented in Table 7.8.

The prediction error was reduced for the memory resource compared to the previous experiment. The effects on CPU and HDD prediction were minimal and the throughput prediction error was below 8.5%. The response time prediction error was decreased by about 8% as shown in Figure 7.19.

**Experiment 8 - SPECjEnterpriseNEXT Alternative Workload** For the final experiment, we altered the workload distribution of the SPECjEnterpriseNEXT benchmark. The original workload used an equal distribution amongst the 9 business transactions. We increased the usage of the *Get Quote* and the *View Vehicle* each to 15% and the rest to

**Table 7.9:** *Measurement and simulation results for SPECjEnterpriseNEXT using an alternated workload accessed by 140 concurrent users*

Resource	Metric	Application Server (AS)		
		Insurance Domain	Insurance Provider	Vehicle Service
Deployment	-			
CPU	Mean measured utilization	55.43%	34.75%	32.43%
	Mean simulated utilization	51.80%	30.65%	28.73%
	Relative error	6.55%	11.80%	11.41%
Memory	Mean committed memory	6.54 GB	1.75 GB	2.68 GB
	Mean simulated memory	6.78 GB	1.93 GB	2.95 GB
	Relative error	3.65%	10.15%	10.19%
HDD	Mean measured demand	3.51%	2.83%	1.83%
	Mean simulated demand	3.51%	2.83%	1.20%
	Relative error	20.80%	28.62%	34.43%
Throughput	Mean transactions per minute	322.12%	-	-
	Mean simulated transactions per minute	293.29	-	-
	Relative error	8.95%	-	-

**Figure 7.20:** *Response times SPECjEnterpriseNEXT Experiment 8 (Alternative Workload)*

10% of the transactions. This reflects the key functions of the system (calculating quotes for certain vehicles) in a more realistic way. The altered workload was configured in the load test and in the *Usage Model* of our performance model. We again compared coarse-grained monitoring data of a load test with a simulation run. The results are listed in Table 7.9.

The two increasingly used business transactions are more CPU intensive. This is very well reflected in our performance model. Therefore, we could decrease the prediction error for CPU and memory by about 5%. The prediction for the response times could also be decreased, as presented in Figure 7.20.

#### 7.4.4 Discussion

The controlled experiments demonstrated the accuracy and feasibility of our approach (Hevner et al., 2004). The simulated memory profile approximates well to the actual profile observed during load tests. We tend to under-predict CPU demands but over-predict memory demands for the application servers. The under-prediction is a result of overhead tasks of the servers that are not part of the model (e.g., DB pool management, load-balancer health checks, etc.) (Willnecker/Krcmar, 2016). The over-prediction of memory demands is a result of the delay between a threshold detection and the GC, the memory growth, or the memory shrink execution in the simulation.

The CPU prediction accuracy has been enhanced compared to previous research (Brunert/Krcmar, 2017; Willnecker/Dlugi et al., 2015c). This shows, that the influence of GC on the CPU resource is significant and requires the consideration of such effects, when predicting the CPU consumption of an EA. Furthermore, the influence on the response times prediction quality is as well positive, due to the indirect time consumption of GC runs when utilizing the CPU resource.

Predicting response times is the most complex task, as the response times are the result of all aspects of an application including all resources and suspension periods. Even though, our model contains all major resources, the model is still an approximation of the reality and thus, to some extent blurred. For instance, DB thread pools are not part of the model, which can lead to delays if they are utilized to capacity, and also the time taken for allocating and releasing memory is not part of the model. Therefore, we tend to under-predict the real response times. However, the relative error is still below 30%, which is adequate for capacity planning tasks (Woodside/Franks/Petriu, 2007; Smith, 2007).

## 7.5 Conclusion

Our work successfully demonstrates an approach for predicting memory and garbage collection behavior. We present a meta-model extension for the well-established architecture-level performance model PCM, a monitoring approach for extracting memory and GC probability distribution, and an extended model generator creating memory-aware models of multiple EAs. We demonstrated the accuracy of this approach in a series of controlled experiments in an on-premise and industry cloud environment. The results approximate the actual memory behavior very well and show that this approach is suitable for memory and GC behavior prediction.

Predicting the impact of software changes, workload changes, or changes to the runtime does not require extensive load tests. A short load test for generating a performance model results in an accurate memory model. This model can be compared to previous models to predict the impact on the memory profile or it can be used to predict the profile in other environments or for longer periods. The latter allows for accurate capacity planning for large-scale EA. Integrated into a continuous integration pipeline, performance bug detection for CPU, HDD, network utilization, and memory allow one to continuously



evaluate the quality of an application's architecture and to adjust the capacity planning (Brunnert/Krcmar, 2017).

Automatic analysis can be integrated into the continuous delivery pipeline of an application. Warnings whenever significant changes occur as feedback for the developer would improve the overall quality of the shipment. Memory issues are detected before production but without extensive load tests and in such fosters fast release cycles (Brunnert et al., 2015).

Future work could improve the model by introducing time consumption for memory allocation and release operations as well as stop-the-world scenarios in certain GC implementations. This would further increase the prediction quality of the model. Furthermore, integrating this model in applications using performance model-based decision making could significantly improve their prediction quality. Most models today are biased, as they do not consider memory or ignore GC behavior and thus predict deployment topologies that require extensive memory or crash at a certain point. Architecture optimizers, runtime decision making, or capacity planning tools benefit from the result of this work (Koziolek/Koziolek/Reussner, 2011; Brunnert/Krcmar, 2017; Brosig/Huber/Kounev, 2014; Aleti et al., 2013).

Models from distributed (enterprise) applications, In-Memory databases, and Big Data systems benefit from our approach when predicting the utilization of all four major resources. Based on these predictions, capacity adjustments or optimizations of the topology are possible and accurate (Willnecker/Krcmar, 2016). General performance improvements as well as cost savings result in a better user experience and lower application operation costs (Roussel/Branson, 2017).

# Chapter 8

## Optimization of Deployment Topologies for Distributed Enterprise Applications

Authors	Willnecker, Felix <sup>1</sup> (willnecker@fortiss.org) Krcmar, Helmut <sup>2</sup> (krcmar@in.tum.de)  <sup>1</sup> fortiss GmbH, Guerickestraße 25, 80805 München, Germany <sup>2</sup> Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Proceedings of 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA) 2016
Status	Accepted
Keywords	Performance Model Generation, Deployment Topologies, Deployment Optimization, Palladio Component Model
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

**Table 8.1:** *Bibliographic details for P5*

**Abstract** Enterprise applications are typically implemented as distributed systems composed of several components. Deciding where to deploy which component is a difficult task that today is usually assisted by logical topology recommendations. Choosing inefficient topologies allocates the wrong amount of resources, leads to unnecessary operation costs, or results in poor performance. Testing different topologies to find good solutions takes a lot of time and might delay productive operations. Therefore, this work introduces a software based deployment topology optimization approach for distributed enterprise applications. We use an enhanced performance model generator that extracts models from running applications. The extracted model is used to simulate performance metrics (e.g., resource utilization, response times, throughput) of an enterprise application. Subsequently, we introduce a deployment topology optimizer, which selects an optimized topology for a specified workload. The following two optimization goals are presented in this work: (i) minimum response time for an optimized user experience and (ii) maximize resource utilization for cost-effective topologies. To evaluate the approach we use the

SPECjEnterpriseNEXT industry benchmark as distributed enterprise application. The evaluation demonstrates the accuracy of the simulation compared to the actual deployment and the pre-eminence of the selected topology compared to runner-up topologies.

## 8.1 Introduction

Distributed architectures are state of the art in large scale Enterprise Applications (EAs) (Brunnert/Krcmar, 2017; Denaro/Polini/Emmerich, 2004; Brunnert et al., 2015). These applications typically consist of multiple deployment units. Each deployment unit is composed of several components and is movable from one runtime instance to another. Furthermore, these units can be replicated to cope with increased workload. Selecting the right amount of deployment unit replications and corresponding runtime instances is a difficult task. Numerous different combinations, so called deployment topologies, exist (Denaro/Polini/Emmerich, 2004; Brunnert et al., 2015). Not only the right amount of unit replications must be selected, but also the right amount of resource containers (e.g., Virtual Machines (VMs), bare-metal server, or application containers), which depends on the demand a component operation places on its resources (Denaro/Polini/Emmerich, 2004). The most important resources Central Processing Unit (CPU), Hard Disk Drive (HDD), memory and network and their demands have to be considered for deployment topology decisions (Koziolek et al., 2014; Brunnert et al., 2015).

These decisions are today usually assisted by logical recommendations or simply based on estimations. Such estimations and resulting topologies often rely on peak demands (Speitkamp/Bichler, 2010), which leads to under-utilized data centers. Different studies estimate the current data center utilization between 6% and 20% (Huang/Masanet, 2015; Speitkamp/Bichler, 2010). Hence, servers are most of their uptime simply idle. On the other hand, over-utilized servers are not desirable, as this results in overly long response times or unstable systems. Therefore, selecting the right amount of resource containers and optimizing the utilization of their resources is important when running EAs (Ardagna et al., 2014).

Instead of deploying an EA in-house, managed infrastructures like cloud environments are available today and can provide extensive reliability and cost reduction (Ardagna et al., 2014). Managed infrastructure providers now invoice the usage of runtime instances. In contrast to once purchased servers, these providers charge current costs. Thus, EA operators have a vested interest to optimize their topologies in order to reduce hosting costs. Different providers apply different cost models based on the number of machines, requests processed, number of user sessions, or simply aggregated uptime. Depending on which provider is chosen, the optimization goal for the deployment might change to save costs. A high utilization might be preferred if uptime of servers is basis of the cost model instead of optimized response times.

In practice deployment topology considerations require a lot of effort. Planning and testing topology changes in a production environment comprise risks for the stability of the EA. Risk assessment compared to the potential savings in operation costs or

performance gains might be negative. Evaluating topologies in test environments require productive-alike environments. Such environments are as expensive as the production environment itself. Furthermore, such testing environments are often used to capacity by various projects executing load tests or may simply not yet be available when new hardware or managed infrastructures like cloud environments are introduced as new target environments (Ardagna et al., 2014).

This work proposes to use performance models extracted from small scale test environments and subsequently size and optimize available resource containers to specified workloads. To accomplish this goal, we combine automatic performance model generators (PMGs) and architecture optimizers to automatically detect optimized deployment topologies. This approach allows to use accurate resource demands from generated performance models and established architecture optimization algorithms. We use the PMG of the RETIT<sup>1</sup> Capacity Manager, the Palladio Component Model (PCM) as performance meta-model and an opt4j<sup>2</sup> based approach to optimize the deployment topologies for EAs (Brunnert/Krcmar, 2017; Koziolok/Koziolok/Reussner, 2011; Lukasiewicz et al., 2011).

Performance models can be used to predict performance metrics by evaluating alternative deployment topologies, resource environments, and simulate the effects on these metrics (Brunnert/Krcmar, 2017). Building such models manually often outweighs their benefits (Kounev, 2005). Recent research created PMGs for EAs to limit the effort of building such models (Brunnert/Krcmar, 2017). However, no current available PMG considers all four main resources (CPU, HDD, memory, and network). The here used PMG has the most comprehensive approach, but lacks automatic memory management simulation. Therefore, we introduce an extension to PCM and the contemplated PMG for dynamic and automatic memory management modeling and simulation.

PMGs focus on the extraction of the software architecture of an EA, but disregard deployment topology decisions. Selecting the right amount of resources and evaluating the selected topologies requires again manual effort. Manual selection soon becomes impossible, as the number of potential topologies grows exponentially with the number of deployment units and available resource containers (Koziolok/Koziolok/Reussner, 2011). To automate this process architecture optimizers have been introduced to the scientific community (Aleti et al., 2013; Koziolok/Koziolok/Reussner, 2011). These optimizers require an already created (performance) model to conduct optimizations (Koziolok/Koziolok/Reussner, 2011). Such models can be derived from design specifications or created manually. However, the actual resource demands are usually estimated and therefore error-prone. Generated performance models provide high accuracy for predictions, but are not yet compatible with architecture optimizers. A combination of PMG approaches and architecture optimizers to automate deployment topology decisions is the main contribution of this work.

Our results allow architects to evaluate different resource environments (e.g., in-house, hosted, cloud), to evaluate different deployment topologies, and to automatically size EAs without deploying the application in a production or production-like environment.

---

<sup>1</sup><http://www.ret.it.de/>

<sup>2</sup><http://opt4j.sourceforge.net/>

For the evaluation we conducted a series of controlled experiments using the industry standard benchmark SPECjEnterpriseNEXT<sup>3</sup> as distributed EA.

The contributions of this work are as follows:

- (i) Combination of performance model generation and architecture optimization.
- (ii) System design and evaluation of automatic deployment topology selection in different resource environments.
- (iii) Dynamic and automatic memory management simulation approach.

## 8.2 Related Work

First PMGs have already been demonstrated in the work of Rolia et al. (1999) (Rolia, 1999). Their work focuses on layered queueing networks (LQNs) which do not separate workload, software components and resource environments (Rolia, 1999). Without such a separation exchanging the resource environment model or changing the deployment topology are hard to accomplish. Therefore, architecture-level performance models such as PCM introduced separated sub-models for workload, software architecture and resource environments (Becker/Koziolk/Reussner, 2009). The work of Brosig et al. (2014) generates performance models for PCM and simulates CPU, HDD and memory demands, but lacks automatic memory management and network demands (Brosig/Huber/Kounev, 2014). Especially in distributed environments the network latency and bandwidth can have a huge impact on the performance of the system (Brunnert/Krcmar, 2017).

Another performance model generation approach has been introduced by Brunnert et al. (2015) (Brunnert/Krcmar, 2017). The generated models are called resource profiles and consider CPU, HDD, and network demands (Brunnert/Krcmar, 2017). The approach generates accurate models from running EAs but lacks automatic memory management (Brunnert/Krcmar, 2017). A thoroughly conducted capacity planning requires to take the resource memory into account in order to charge the capacity of available systems effectively. Therefore, we extend this generator with automatic memory management simulations in our deployment topology optimization architecture.

Speitkamp et al. (2010) identified the need to consolidate resource usage and proposed a mathematical model to optimize resource allocation using VMs (Speitkamp/Bichler, 2010). VMs with high CPU utilization could be run on the same host together with VMs utilizing other resources having low CPU utilization (Speitkamp/Bichler, 2010). This concept should optimize the utilization of all resources in a data center. However, the proposed model is not aware of the workload or the EAs running in the VMs and their dependencies. The model requires a re-calculation and allocation of the VMs when the resource utilization of the hosted applications changes. Such changes occur frequently

---

<sup>3</sup>SPECjEnterpriseNEXT is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterpriseNEXT 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.

as new versions are deployed or the workload executed changes. An architecture aware optimization approach can produce better results.

The architecture optimization approach PerOpteryx evaluates design alternatives based on PCM models (Koziolek/Koziolek/Reussner, 2011). The number of decisions is large as hardware, network and software architecture is taken into account. PerOpteryx has a broad variety of optimization goals and degrees of freedom. We adapt the PerOpteryx approach for deployment optimization but with certain changes (e.g., simulations instead of solvers and limitation on deployment topology decisions). This allows us to not only evaluate CPU utilization and response times, but to also take network, automatic memory management and HDD demands into account.






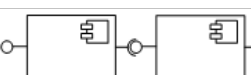
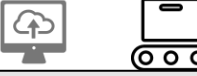
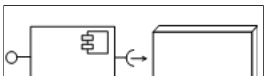

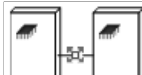
Aleti et al. (2013) provide a comprehensive review on software architecture optimization methods (Aleti et al., 2013). The work recommends to provide evidence for resulting architectures in order to proof the validity of these strategies and suggests to establish holistic tool support (Aleti et al., 2013). Current work fails on providing evidence as the presented approaches are hard to compare to real environments. Our work addresses this gap by evaluating architecture optimization with a real EA and combining architecture detection (performance model generation) with architecture optimization into a holistic tool for automatic deployment topology optimization.

### 8.3 Enterprise application components

A deployment unit is a packaged artifact installable on a middleware instance or directly on an operating system (OS). Such units consist of several components and operations and build the core of any EA. A typical EA consists of many deployment units distributed throughout a middleware.

In order to analyze the deployment topology of an EA its context has to be taken into account. The context comprises (i) the resource profile of the EA and (ii) the workload executed on the EA. Hence, an optimized topology always depends on both factors. Figure 8.1 depicts the main components of an EA. We use Java Enterprise Edition (EE) as an example middleware, even though the depicted concepts are applicable for other technologies as long as monitoring technology is available (Spinner et al., 2015; Willnecker/Brunnert et al., 2015a). Furthermore, Figure 8.1 shows how we map the different parts of an EA to the PCM meta-model.

The workload describes the number of users and how they use the EA, which ultimately causes the resource utilization of the EA. These users can be real users accessing the system or virtual users when executing a (load) test to analyze the behavior of an EA. We use Application Performance Management (APM) data during a monitoring run to reason an initial workload. However, this workload is changeable in order to size an EA according to the expected workload in production.

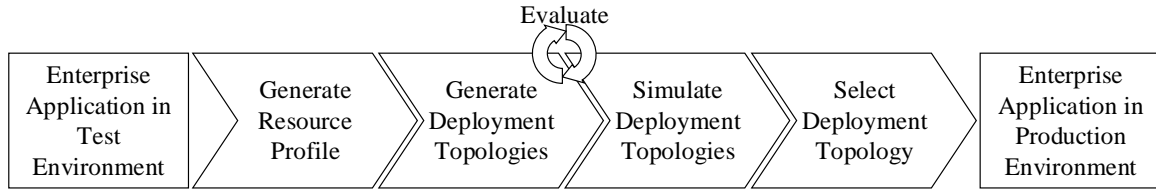
	Concept	Java EE	Palladio Component Model
Workload	Number of Users and User Behavior	Real Users or Virtual Users 	Usage Model 
Resource Profile	Components and Operations	EJBs, JSPs, Servlets 	Repository Model 
	Deployment Units	EARs/WARs 	System Model 
Deployment Topology	Deployment Unit to Resource Container Relationship	Installation on Application Server 	Allocation Model 
	Resource Container, Resources and Network Conn.	Virtual Machine/ Bare-Metal Server 	Resource Env. 

**Figure 8.1:** *Enterprise application components (adapted from Becker/Koziolek/Reusser (2009); Brunnert/Krcmar (2017)).*

The resource profile of an EA describes how an operation of a component utilizes different resources (Brunnert/Krcmar, 2017). The profile consists of a basic workflow and the deployment unit structure of the EA. Each operation of the EA is modeled in the profile including its resource demands for several resources. Resource profiles are the core result of PMGs (Brunnert/Krcmar, 2017).

The runtime of the EA representation is defined by the deployment topology. For each deployment unit of a distributed EA exists at least one instance during runtime. If the workload necessitate replicas of this deployment unit might exist. If depicted as a graph, each deployment unit node needs at least one edge to a resource container node. The node is replicated if multiple edges from a deployment unit node exist.

A deployment topology describes the structure and relationship of a set of these deployment units installed and executed on a number of resource containers. This relationship is in PCM defined by the allocation model. The containers are organized in a resource



**Figure 8.2:** *Deployment topology optimization process.*

environment. This environment consists of the hosting machines with their capabilities (e.g., CPU processing rate, available memory, HDD speed) and the network connection (focusing on bandwidth and latency) between the resource containers. Two deployment units, which are dependent, can only be deployed on two resource containers that are linked via a network connection. Therefore, the number of potential topologies depends on the number of deployment units  $du$  and the valid resource containers  $rc$ . Equation 8.1 calculates the number of possible deployment topologies depending on  $du$  and  $rc$ , when all resource container targets are valid for all deployment units.

$$DT_{du,rc} = (2^{rc} - 1)^{du} \quad (8.1)$$

$2^{rc} - 1$  describes each possible installation combination of an deployment unit on one or more resource containers. As any permutation with other deployment unit installations is possible, we have to add  $du$  as an exponent. Given 10 resource containers and 5 deployment units the number of possible topologies is already greater than  $1^{15}$ . The number of combinations in this scenario already prohibits a manual selection. Deployment topology optimization requires an automated approach to save time and costs.

## 8.4 Deployment Topology Optimization Process

An automated approach requires an holistic tool to optimize deployment topologies, which consists of three basic components:

- (i) Performance model generator to detect the resource profile of an EA including its resource demands, system behavior, current deployment topology, and current workload.
- (ii) Architecture optimizer to evaluate different target deployment topologies and to select the best topology in terms of the optimization goal.
- (iii) Simulation service for parallel predicting performance metrics of multiple performance models.

Figure 8.2 illustrates the optimization process. We deploy a distributed EA in a test environment to conduct the process. This EA is instrumented with APM software and set under load in order to get meaningful APM data. In a first step, this APM data is used to generate a performance model. The model consists of the detected workload, the



detected resource profile and a specification of the resources in the test environment. The generated model represents the current state of the EA in the test environment.

In a second step the architecture optimizer based on an optj4 uses an evolutionary algorithm for selecting an initial number of possible topologies (initial population) (Lukasiewicz et al., 2011). This initial topology set is based on the generated model and the available resource containers. Each topology is validated in order to check if all deployment units are at least instantiated once and can communicate with all dependent deployment units via a network connection. The deployment topologies are packaged for simulation including workload and resource profile. We constructed a distributed simulation cluster that can simulate multiple topologies in parallel.

After each simulation run the optimizer evaluates the results. We currently support two optimization goals: the algorithm assess either the mean resource utilization per resource over all containers or the response time per transaction. Thus, our weighting function either maximizes the utilization per container (reduces the number of used resource containers) or minimizes the response time. A topology can be invalidated if one resource is utilized above a certain threshold to prevent over-utilizing certain containers. The optimizer mutates new topologies based on the evaluation results and delegates the simulation. This process is repeated until the initial population and the number of generations are processed.

The best topology in terms of the optimization goal (min. number of resource containers or min. response time) is selected and provided as a result. The final step is deploying this topology in the production environment.

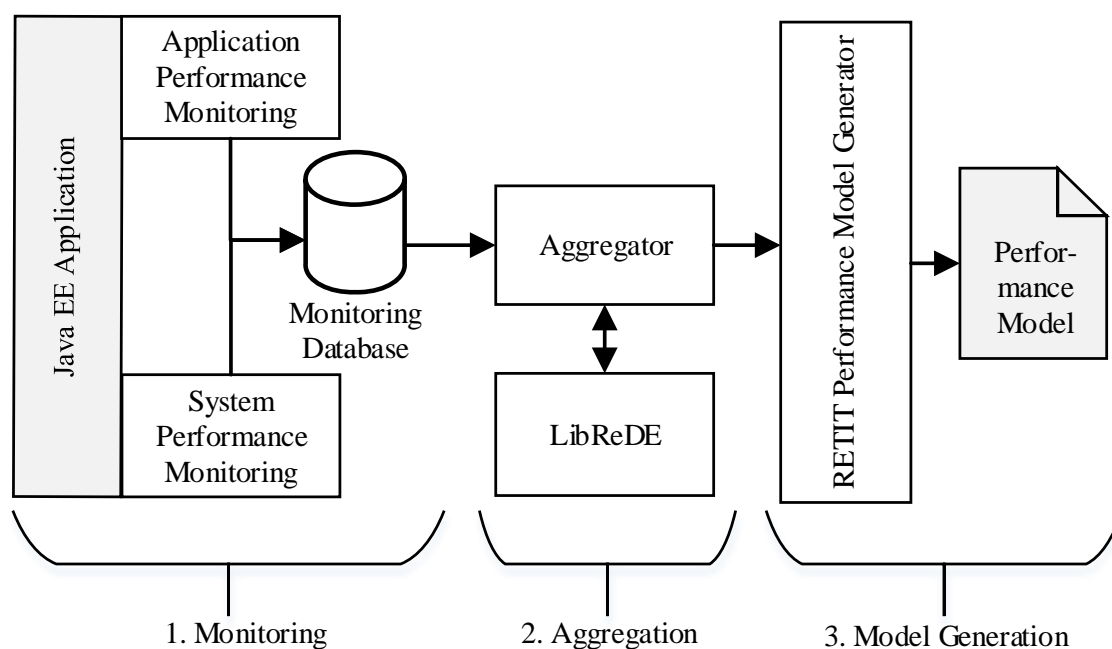
## 8.5 Performance Model Generator

This section explains the PMG and extensions we added to support generating a comprehensive performance model. The generated model considers the most important resources: CPU, HDD, network and memory. Furthermore, a transaction flow throughout a distributed EA is detected and used for reasoning the system control-flow.

We use and extend the PMG of the RETIT Capacity Manager and the corresponding monitoring solution RETIT Java EE. Both solutions are based on the Performance Management Work Tools (PMWT) PMG introduced by Brunnert et al. (2015) (Brunnert/Krcmar, 2017). As depicted in Figure 8.3 the generation process consists of three phases (Willnecker/Dlugi et al., 2015c):

- (i) Monitoring the instrumented EA,
- (ii) Aggregating the monitoring data per operation, and
- (iii) Generating the performance model based on the aggregated monitoring data.

The result of the three phases is a resource profile, a workload description representing the usage of the EA during the monitoring phase, and a resource environment describing



**Figure 8.3:** *Performance model generator framework (adapted from Willnecker/Dlugi et al. (2015c)).*

the current deployment of the EA. All three model parts are stored as a PCM instance (Brunnert/Krcmar, 2017; Becker/Koziolek/Reussner, 2009).

### 8.5.1 Monitoring

The monitoring step collects operation invocations of the instrumented EA. We distinguish between resource demand measurement and resource demand estimation (Spinner et al., 2015; Willnecker/Dlugi et al., 2015c). Resource demand measurement uses fine grained monitoring data per operation invocation to measure the exact demand an operation places on a resource. These measurements can be collected with standard APM software like Dynatrace<sup>4</sup> Application Monitoring (AM) (Willnecker/Dlugi et al., 2015c). Resource demand estimations use coarse grained monitoring data like total resource utilization and response time series per operation and distribute the utilization throughout the operations (Spinner et al., 2015). Such coarse grained resource utilization data can be collected using standard system monitors like System Activity Reporter (SAR), procs<sup>5</sup>, or monitoring and control interfaces of virtual machines like Java Management Extensions (JMX). Load drivers like jMeter<sup>6</sup> or access logs of web servers provide response time series of operations invoked on system-entry level. For more detailed (e.g., component-level) response time series, custom filter, or logger are necessary.

<sup>4</sup><http://www.dynatrace.com/>

<sup>5</sup><https://www.kernel.org/doc/Documentation/filesystems/proc.txt>

<sup>6</sup><http://jmeter.apache.org/>

The PMG supports data from different data sources as depicted in Figure 8.3:

- (i) Application Performance Monitoring for fine grained application data. We use the RETIT Java EE Monitoring solution in this work. Previous work demonstrated the applicability of industry standard solutions like Dynatrace AM (Willnecker/Dlugi et al., 2015c).
- (ii) System Performance Monitoring for coarse grained application data. Standard system tools or custom host agents are possible. We use Apache Webserver<sup>7</sup> access logs, RETIT Host Monitoring and JMX in this work.

The collected data is stored ongoing in a monitoring database based on the Apache Cassandra<sup>8</sup> project. The large amount of data requires a scalable, yet simple database structure. Each row in the database corresponds to an operation invocation or a measurement record from system monitoring. The next phase uses the monitoring database as single source of input.

### 8.5.2 Aggregation

The aggregation phase concentrates all the single operation invocations as a preparation for the the model generation by consolidating all invocations of the same operation. The mean demand per resource (e.g., mean CPU demand) is calculated for every operation of every component. Furthermore, the calculation of network demands at the deployment unit boundaries and transaction flows based on unique transaction IDs is conducted in this step.

In order to support resource demand measurement and resource demand estimation approaches, we extended RETIT Java EE monitoring with the Library for Resource Demand Estimation (LibReDE) (Spinner et al., 2015; Willnecker/Dlugi et al., 2015c). Figure 8.3 shows the currently supported data sources in the monitoring step, however, other APM solutions or coarse grained monitoring providers can be added to the PMG. The demand calculation is either done by the aggregator, or delegated to LibReDE in a post-processing after initial aggregation. After this phase, all operation invocations, their resource demands and the transaction flow are prepared for the model generation phase.

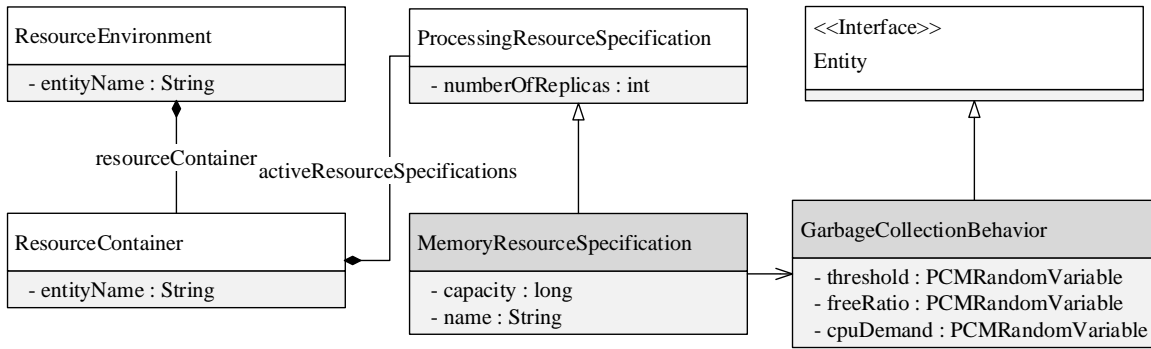
### 8.5.3 Model Generation

This final phase transforms the operation invocations, demands and the transaction flow into a PCM instance. This allows changing and/or simulating, hence predicting the system behavior using the Palladio-Bench (Becker/Koziolok/Reussner, 2009). PCM supports CPU, HDD and network demands. The in-built features of PCM are used for the the above mentioned three resources (Brunnert/Krcmar, 2017).

---

<sup>7</sup><https://httpd.apache.org/>

<sup>8</sup><http://cassandra.apache.org/>



**Figure 8.4:** *PCM extension for memory resources.*

The generator calculates CPU demands using the mean CPU demand per operation (Brunnert/Krcmar, 2017). Each operation invocation results in at least one action with a CPU demand. In contrast, for HDD demands we distinguish between write and read demands, as the write speed of a HDD is very different from the read speed (Brunnert/Krcmar, 2017). Network demands are only measured at the deployment unit boundaries. We calculate the mean request and response size to simulate the time this request travels through the network based on available bandwidth and latency (Brunnert/Krcmar, 2017).

Memory demands and simulation are more complex due to automatic memory management (Libič et al., 2015). We can calculate the mean memory demand of an operation by measuring the memory demand of each operation invocation and forming the average demand similar to CPU demands. To place the demands a dedicated resource is necessary, which supports dynamic memory management scenarios as in C-based systems and automatic memory management scenarios as in virtualized runtimes like the Java Virtual Machine (JVM).

We extended the PCM meta-model in order to add a memory resource representation as depicted in Figure 8.4. This resource works for dynamic memory management scenarios and supports different types of automatic memory management methods, like Garbage Collections (GCs). GCs delay the release of memory, leading to a larger memory utilization during runtime. Thus, memory is more likely to become a bottleneck. To simulate this effect we extended PCM (Becker/Koziolek/Reussner, 2009; Willnecker et al., 2015b). We added two classes to the meta-model to support this behavior:

- (i) *MemoryResourceSpecification* to specify the attributes of a memory resource.
- (ii) *GarbageCollectionBehaviour* to define the behavior of automatic memory management. No behavior is specified in dynamic memory management scenarios.

A typical GC collects and stores released objects in different memory spaces (Libič et al., 2015). The spaces are cleaned in different intervals. For instance, the JVM executes two types (minor and major) of GCs to clean different spaces or promote objects to another space (Libič et al., 2015). Objects which cannot be released, due to references to active objects or objects in other spaces, are either contained in the same space or promoted to the next higher space. Objects without any active reference are released and the used space in memory is free to be allocated by new objects. As long as objects are stored

in one of the GC spaces, allocated bytes are not released and are, thus not available for new objects. A memory simulation containing garbage collection requires to monitor GC events and to generate instances of the memory resource and the GC behavior in PCM.

For Java EE, we monitor the GC events of the running application using the `GarbageCollectorMXBean`<sup>9</sup> via JMX and measure the following metrics:

- (i) The type of garbage collection that is executed. For Java EE this is either a minor GC or a major GC. Other GC implementations or technologies can have different GC types.
- (ii) Size of total memory available in the JVM.
- (iii) Size of allocated memory before and after the GC execution. This is a simplification of the actual mechanism, as we do not simulate object movements in the fine grained GC spaces. This probabilistic approach enables automatic memory management simulation with low overhead compared to complex object movement simulations (Libič et al., 2015; Willnecker et al., 2015b).
- (iv) CPU time necessary to execute the GC.

The measurement data is aggregated and processed for the performance model generation. The resource environment generation creates a *MemoryResourceSpecification* instance. If GC behavior is detected, the capacity of this resource is set to the total available memory detected. For dynamic memory management no further generation is conducted. For automatic memory management we extract the GC types and create a *GarbageCollectionBehavior* instance for each GC type. For each behavior instance we calculate the mean CPU demand per byte released, the mean free ratio and the threshold leading to a GC execution. Threshold and free ratio are calculated in percent and are, hence independent from the current memory size. This implementation automatically adapts *GarbageCollectionBehavior* to other resource containers representing larger or smaller servers with less or more available memory for the JVM.

In order to access the newly introduced memory resource we extended *ResourceCalls* in PCM. We added two more signatures to execute allocation and free calls on this resource. Each operation in the performance model calls the alloc signature of the corresponding memory resource. No free call is necessary for automatic memory management, as this is handled by the *GarbageCollectionBehavior*. In dynamic memory scenarios the free operation is called after each operation. The available memory is immediately increased by the amount specified in the free call. We extended the PMG to generate such *ResourceCalls* automatically for every operation that allocates memory.

For automatic memory management a thread per memory resource is started with the simulation. This thread checks in frequent intervals if the threshold of the corresponding resource exceeds. In this case, a GC run is simulated. The memory of this resource is freed depending on the free ratio of the executed *GarbageCollectionBehavior*. A CPU demand depending on the CPU demand per byte of the *GarbageCollectionBehavior* and the number of freed bytes is placed on the CPU resource of the same *ResourceContainer*.

---

<sup>9</sup><https://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/GarbageCollectorMXBean.html>

In addition to the extraction of the system behavior and resource demands, we are able to extract the workload executed on the EA during instrumentation. Therefore, we measure the number of calls at system-entry point per operation. The PCM usage model supports so called *OpenWorkloadScenarios*. We create such a scenario per operation on system-entry point. To calculate the workload per operation we compute the mean inter arrival time *IAT* between two calls of this operation. The *IAT* is calculated by dividing the total time of the monitoring *MonTime* by the total number of calls of this operation *TotalCalls<sub>Op</sub>* as stated in Equation 8.2.

$$IAT_{Op} = \frac{MonTime}{TotalCalls_{Op}} \quad (8.2)$$

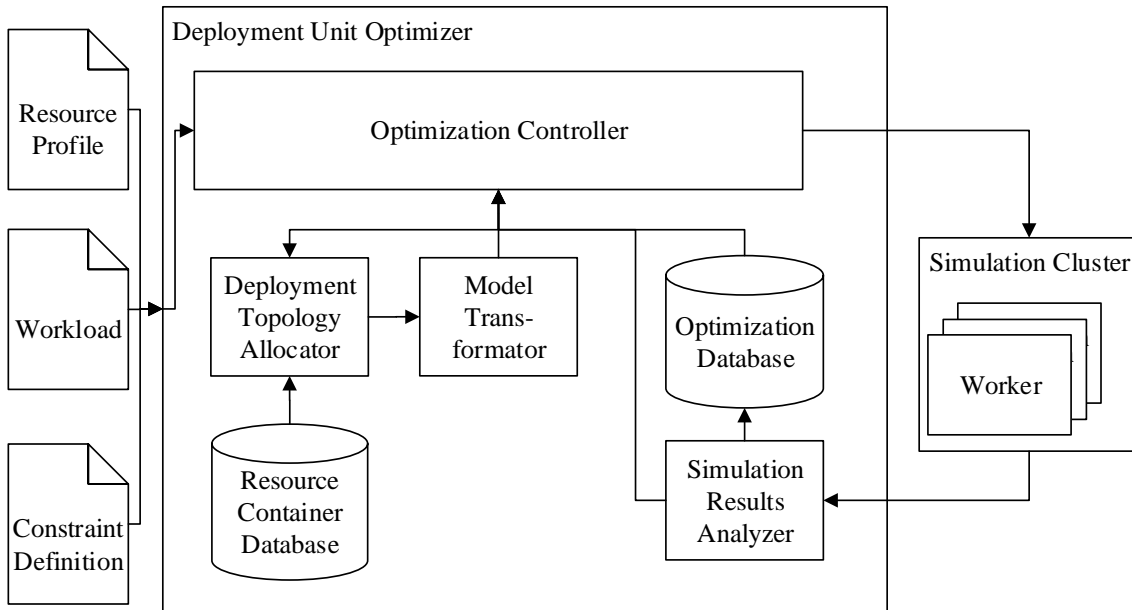
## 8.6 Architecture Optimizer

The architecture optimizer is the core of the deployment topology optimizer. It is based on the opt4j framework (version 3.1.4) for evolutionary computation (Lukasiewicz et al., 2011). While the technological foundation is similar to PerOpteryx, we use simulations instead of solvers for the evaluation of design alternatives (Koziolek/Koziolek/Reussner, 2011). The Palladio solver works only for single user scenarios. Alternatively, we could transform our PCM model to a LQN and use LQN solver. However, model transformations are never loss free and we know from previous research, that small changes in the system can have large impact on the overall performance and the decision process (Brunnert/Krcmar, 2017). Simulations provide high accuracy considering the predictions results although they are computational intensive compared to solvers. Therefore, we decided to use simulations in our approach. To limit the effect on the time it takes to find an optimized solution we added a simulation cluster to process several simulations in parallel on a dedicated system.

To conduct an optimization run three input artifacts are necessary as depicted in Figure 8.5:

- (i) The resource profile created by the PMG containing the software architecture and resource demands of the EA that should be optimized.
- (ii) The expected workload of the system to deploy. The simulations use this workload to predict the utilization of the available resources. This may be the same workload as generated by the PMG but usually the workload of the generation run is smaller than the expected workload in the production system.
- (iii) Constraints to the optimization like the minimum/maximum utilization of a resource, the minimum/maximum number of systems, and logical constraints that can for instance prohibit deploying the database on a resource container that comprises an Application Server (AS).

The *Optimization Controller* acts as workflow controller using obj4j and triggers the sub components of the system (Lukasiewicz et al., 2011). In a first step it requests design alternatives from the *Deployment Topology Allocator*. Therefore, this allocator takes a



**Figure 8.5:** *Deployment Unit Optimizer.*

number of resource containers and network connections between these containers from the *Resource Container Database*. The database consists of a list of all available containers that should be considered for the optimization run. Such a database is filled with available instances in a data center or with instances offered by a cloud provider. The capability of the CPUs of these containers must be calibrated in comparison to the CPUs used in the test environment where the monitoring for model generation was conducted. Brunnert et al. (2015) showed that CPU benchmarks provide accurate results for transforming CPU resource representations from one machine to another (Brunnert/Krcmar, 2017).

The initial population is calculated by the *Deployment Topology Allocator* out of the list of containers, deployment units, and constraints. The initial population is input of the evolutionary algorithm implementation of opt4j, the so called genotype (Lukasiewicz et al., 2011). Our genotype can be described as a  $du \times rc$  matrix  $G$ . Each column represents one available resource container  $rc$  and each row represents a deployment unit  $du$  of the EA. The values of the cells of the matrix are either 0 or 1. A 1 in the cell  $G_{i,j}$  indicates that the deployment unit  $du_i$  is deployed on resource container  $rc_j$ . The genotypes are created randomly, but invalid selections are discarded. A valid matrix has at least one 1 in each row, so that each deployment unit is at least deployed once and complies to the constraints (e.g., no database (DB) deployment unit on the same container as an application deployment unit).

The second step is the transformation of the matrix into a PCM instance. The deployment topology is transformed into resource environment and allocation model instances. The PMG only creates one instance per deployment unit. Therefore, additional deployment unit instances are generated in the repository and system model if necessary. The result of this process is a complete PCM instance ready for simulation acting as the phenotype of the evolutionary algorithm (Lukasiewicz et al., 2011).

The evaluator of the evolutionary algorithm is implemented in multiple components (Lukasiewicz et al., 2011). To prevent simulating the same topology twice, the *Optimization Controller* checks against the *Optimization Database* if results for this topology are available. The DB is used to detect already simulated topologies in order to prevent duplicated simulations. A topology is considered equal if the deployment units are distributed throughout equal resource containers. Two resource containers are considered equal, if their resources have the same capabilities. If an equal topology is detected, the simulation results from the *Optimization Database* are used.

If no equal topology is detected a simulation job is dispatched to a simulation cluster. The cluster consists of several worker nodes running the Palladio-Bench simulations in an headless eclipse instance (Becker/Koziolek/Reussner, 2009). The simulation job is dispatched to a worker and automatically queued and executed when resources are available. The cluster consists of at least one worker node that executes the simulations. This worker node is connected to a load balancer and stores the results in a shared folder. Therefore, each worker node is able to simulate a PCM meta-model instance and provide results for every conducted simulation.

The *Optimization Controller* sends an archive containing all model elements as depicted in Figure 8.1 to the cluster. The load balancer uses a session sticky round robin approach to balance the load across all workers. This means that new requests will be placed randomly on one of the workers. Follow-up requests, like requesting the status or exporting the results will be executed on the same worker node. Furthermore, as the results of the simulation are stored in a shared folder results can be retrieved even after a worker was shutdown.

The bottleneck of a simulation run is usually the memory. Hence, each worker node is memory aware and only starts a new headless eclipse instance if enough memory is available. The parameter of a simulation job are part of the initial request. A new job is queued, if the maximum required memory of the simulation job exceeds the available memory in the JVM. The *Simulation Results Analyzer* can request the status of each job based on its unique *jobID*. The job can either run, be queued, be finished or be failed. If a job is finished the results of the simulation are returned as an archive containing all simulation metrics and results.

The analyzer parses these raw results of the simulation and calculates aggregates, e.g., total resource utilization or response times per operation. Afterwards, it stores the aggregated results together with the topology in the *Optimization Database* and forwards the results to the *Optimization Controller*. The controller spawns new topology based on the evaluation results and the configuration of the evolutionary algorithm (Lukasiewicz et al., 2011). We currently support two optimization goals:

- (i) Minimize the total response time of the system.
- (ii) Maximize the resource utilization of all resources in all used resource containers.

For the first optimization goal we calculate the mean response time over all simulated operations. The deployment topology with the smallest median response time is selected



**Table 8.2:** *Software and hardware configuration of the deployment*

Server	Load Balancer	Insurance Customer Driver	Insurance Domain	Vehicle Service	Insurance Agent	Insurance Database	Vehicle Agent
Application Server/Database	Apache 2.2.31	Faban 1.3.0		JBoss Wildfly 8.1.0		PostgreSQL 9.4.4	
Java Virtual Machine	-			OpenJDK 1.8.0_40			
Operating System	CentOS 6.7			openSUSE 13.2 (x86_64)			
CPU Cores		4 vCores (2.1 Gigahertz)				8 vCores (2.1 Gigahertz)	
Memory	8 Gigabyte	16 Gigabyte				8 Gigabyte	
Host System		IBM System X3755M3					
Network		1 gigabit-per-second (Gbit/s)					

as optimized deployment topology. This optimization goal tends to require more resources and generally generates more costs when executed in production.

The second optimization goal creates more resource efficient topologies. It is important to limit the maximum CPU utilization up to 70% or 80%, otherwise the resulting response times are not representative as the system becomes unpredictable at maximum load. This optimization sorts for example memory intensive deployment units to CPU intensive deployment units. In total the number of resources used is lower with this optimization goal compared to the first goal.

Although, the proposed approach tries to handle automatic deployment topology optimizations in a comprehensive way, it still has certain limitations. (i) Distributed Database Management System (DBMS) have not been considered. Instead of sharding or replicating the DBs, we can today only size VMs for the DBs according to the workload. A more advanced DB performance model and dedicated monitoring solutions are necessary to dissolve this limitation. (ii) The results calculated by the architecture optimizer are never certain to be the best solution. Such algorithms can run into local optima and never find the best possible solution or it might take too much time to find a good solution. However, such a structured approach optimizes effectively based on all EA components (workload, resource profile, and available resource containers) (Koziolek/Koziolek/Reussner, 2011).

## 8.7 Evaluation

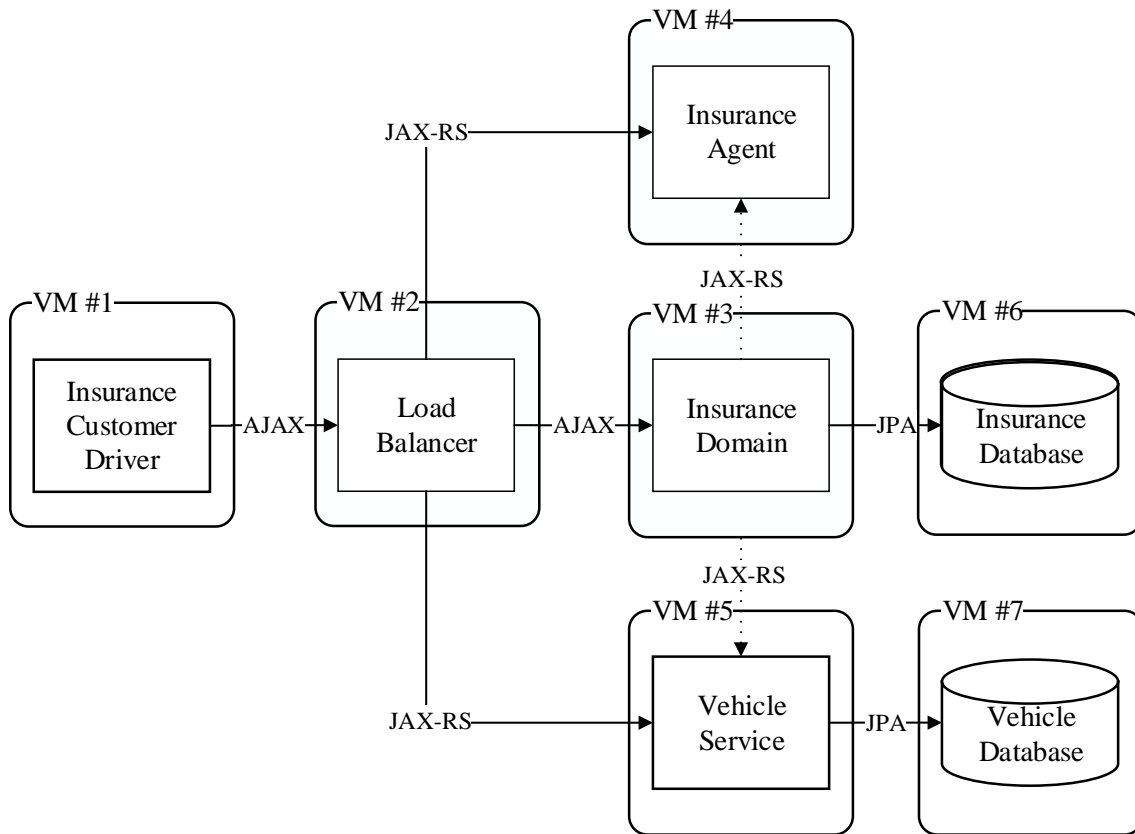
The SPECjEnterpriseNEXT industry benchmark is the successor of the SPECjEnterprise2010 benchmark. Both are Java EE applications typically used to benchmark the performance of different Java EE AS. We use a pre-release version<sup>10</sup> of the SPECjEnterpriseNEXT as example EA. This application represents an insurance policy management system for car insurances. It consists of three different components (Insurance Domain, Vehicle Service, Insurance Agent) and two databases (Insurance and Vehicle Database) as depicted in Figure 8.6. Furthermore, we deploy a load balancer before the AS instances to handle multiple AS instances in the optimized topologies. Each component is deployed as one deployment unit in one VM for the performance model generation. The initial deployment is described in Table 8.2.

The Insurance Customer Driver is based on Faban<sup>11</sup> and executes five different business transactions on the Insurance Domain server, which triggers JAX-RS<sup>12</sup> REST calls on the

<sup>10</sup>version from 29.06.2015

<sup>11</sup><http://faban.org/>

<sup>12</sup><https://jax-rs-spec.java.net/>



**Figure 8.6:** *SPECjEnterpriseNEXT* test deployment.

other two servers (Fielding, 2000). We conducted a run with active instrumentation and 100 virtual users to collect APM data for the model generation. We used the RETIT Java EE solution for the ASs. This includes a Java Database Connectivity (JDBC) wrapper to collect response times from the DB servers. For the load balancer and database VMs we used the RETIT host monitoring to collect CPU utilization. The load balancer response times are collected from Apache Webserver access logs<sup>13</sup>. With the response times and utilization we estimated the resource demands for the DB servers and the load balancer using LibReDE (Willnecker et al., 2015b).

After initial generation we conducted two optimization runs. One for each optimization goal. Furthermore, we increased the workload to 500 users. The initial population of the architecture optimizer consisted of 100 different topologies. We selected a cross-over rate of 0.95, an offspring population size  $\lambda$  and a parent population size  $\mu$  of 25 each. The algorithm calculated 100 generations resulting in a total of 2500 tested topologies. We used 4 worker nodes in the simulation cluster resulting in about 8 hours of computation.

The topology for the minimum response time goal required 8 ASs, 2 DB servers and 1 load balancer. We deployed the system as specified and conducted a run without instrumentation to compare the response times with the simulation. Figure 8.7 shows the comparison of measured response times (MRTs) and simulated response times (SRTs) for

<sup>13</sup><https://httpd.apache.org/docs/2.2/en/logs.html>

**Table 8.3:** *Measurement and simulation results*

Resource	Metric	Load Balancer	AS 1	AS 2	AS 3	AS 4	AS 5	DBMS 1	DBMS 2
CPU	Measured utilization	21.21%	67.37%	73.78%	71.13%	64.37%	61.27%	22.46%	45.34%
	Simulated utilization	20.19%	60.19%	65.05%	64.54%	57.98%	53.78%	21.79%	44.32%
	Relative error	4.81%	10.66%	11.83%	9.26%	9.93%	12.22%	2.98%	2.25%
Memory	Measured demand	-	3627.23 MB	2212.19 MB	2879.76 MB	3987.05 MB	4430.45 MB	-	-
	Simulated demand	-	3291.78 MB	2349.77 MB	3123.23 MB	3719.78 MB	3987.70 MB	-	-
	Relative error	-	9.25%	6.22%	8.45%	6.70%	9.99%	-	-
HDD	Measured demand	-	1.13%	0.45%	0.89%	1.45%	2.34%	-	-
	Simulated demand	-	0.98%	0.37%	0.80%	1.33%	1.98%	-	-
	Relative error	-	13.27%	17.78%	10.11%	8.28%	15.38%	-	-

each business transaction. The simulation tends to under-predict the response times, as not all performance-relevant aspects are part of the model (e.g., DB queues as well as HDD demands of the database are not part of the model). The median response time error is around 18% and well below the 30% acceptable error propagated by Menascé and Almeida (2002) (Menascé, 2008). To ensure that runner up topologies do not produce better results, we compared the top 5 deployment topologies. These topologies had only minor differences and produced almost identical or worse results in the simulation. Monitoring the solutions produced similar results for the top 3 deployment topologies, but worse for the 4<sup>th</sup> and 5<sup>th</sup> best deployment topologies.

We used the same model and workload for the second evaluation. Instead of minimum response time we optimized for maximum resource utilization, but limited the maximum CPU utilization to 70%. The best topology needed 5 ASs, 2 DBMS servers and 1 load balancer. We deployed the same system and executed a monitoring run to compare it with the simulation. Table 8.3 shows the result of this evaluation. The memory simulation has only been conducted for the ASs as the memory demand for the load balancer and DBMS servers are not available in simulation. This is due to the fact that LibReDE calculates demands only for the CPU resource and no dedicated monitoring solution was available for these systems. The same fact is true for HDD demands. The accuracy is quite high for CPU demands, especially when LibReDE is used. This confirms the results of our previous research (Willnecker et al., 2015b). We tend to under-predict CPU demands but over-predict memory demands for the ASs. The under-prediction results from overhead tasks of the servers that are not part of the model. The over-prediction of memory demands originates from the time between a threshold detection and the GC execution in the simulation.

## 8.8 Conclusion

This work successfully connected PMGs and architecture optimizer for optimizing deployment topologies of EAs. This approach allows software architects to detect the current software architecture and demands of an EA and to automatically size and optimize target environments. This holds true in up-scaling scenarios as presented in Section 8.7. For practitioners, this allows to reduce response times, increase resource utilization and, thus, reduce costs. For academia, this approach can be used to test optimization approaches against real world applications.

In addition, this work presented an extension for the PCM meta-model for memory resources. The extension supports dynamic and automatic memory management and introduces a probabilistic GC simulation. Monitoring and detecting GC events as well as generating performance models with an accurate memory model have been integrated into the RETIT PMG and the PMWT. The accuracy and feasibility of this approach have been demonstrated.

In contrast to previously introduced architecture optimizers, the here presented approach uses simulations instead of solvers. The high accuracy of the simulations presented in Section 8.7 justify this decision. However, this accuracy comes with computational costs. To limit the effect on the decision time we introduced parallel execution in a simulation cluster.

Future extensions of this deployment topology optimizer can speed up the calculation by improving the simulator or by introducing a two-phase optimization. A faster approach (e.g., solver, simplified model) could be used to calculate a good initial population instead of random topologies. This population would be the input for the optimization approach presented in this work. This concept could lead to faster or better results.

Faster calculations would allow to use this concept for runtime decisions. Optimized topologies could be calculated on-the-fly, when continuously monitoring and analyzing the workload of an EA. Current runtime models tend to allocate new resources when spikes in the workload occur. A workload and software architecture aware can produce more resource efficient results and therefore reduce overhead and costs in distributed EAs.

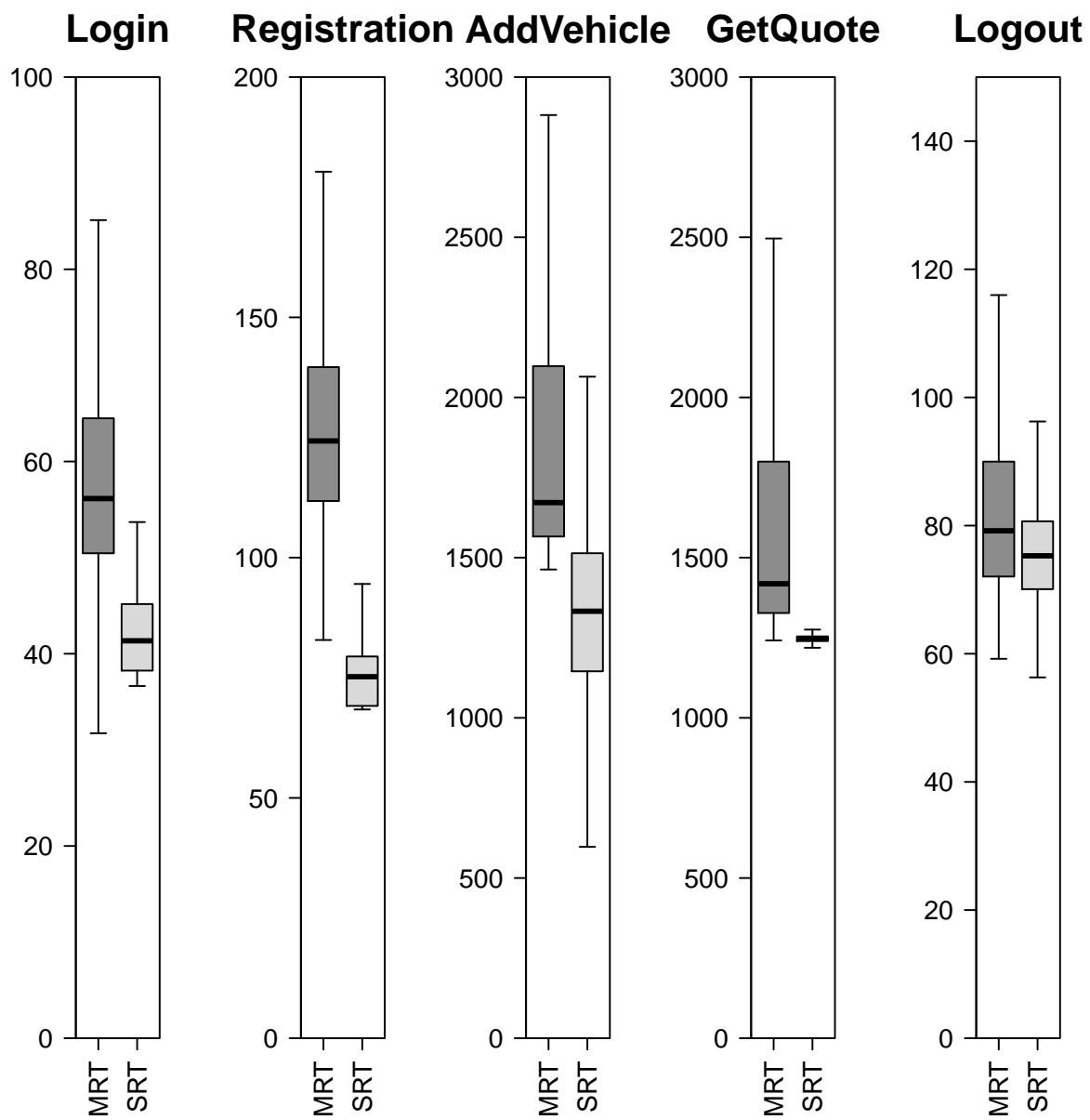


Figure 8.7: Response time evaluation.

# Chapter 9

## SiaaS: Simulation as a Service

Authors	Willnecker, Felix <sup>1</sup> (willnecker@fortiss.org) Vögele, Christian <sup>1</sup> (voegele@fortiss.org) Krcmar, Helmut <sup>2</sup> (krcmar@in.tum.de)  <sup>1</sup> fortiss GmbH, Guerickestraße 25, 80805 München, Germany <sup>2</sup> Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Proceeding of the Symposium on Software Performance (SSP) 2016
Status	Accepted
Keywords	Performance Simulation, Microservice Architecture, Scalable Simulation Engine, Palladio Component Model
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

**Table 9.1:** *Bibliographic details for P6*

**Abstract** One major advantage of performance models over tests using real systems is the ability to simulate design alternatives by simply modifying or exchanging parts of such models. However, the evaluation of numerous design alternatives can be time consuming depending on the number of alternatives and the complexity of the model. To overcome this challenge, this work presents a scalable simulation service for the Palladio Component Model (PCM) workbench based on a headless Eclipse instance, a Java EE application server, packaged in a docker container and run in kubernetes. The simulation service supports parallel simulation runs, multiple PCM instances in the same container and scales out automatically, when resources of one container instance exceed. Simulation jobs are triggered by a platform-independent REST interface and can be re-used by other applications. This allows to simulate a vast amount of model instances in parallel on cloud or on-premise installations.

## 9.1 Introduction

Performance models and corresponding simulation techniques are able to predict performance metrics (e.g., resource utilization, response times, throughput) of applications systems (Brunnert et al., 2015). Simulations become time consuming and resource intensive when multiple simulations are executed, or the complexity of the simulated model, the number of simulated users or the simulation time increases (Brunnert et al., 2015). Today, these simulations are computed on the workstation of the user of this model in a workbench application one after each other (Becker/Koziolk/Reussner, 2009). Parallel simulations are merely possible, only by running multiple workbench instances in parallel including the resource overhead for running multiple applications. Furthermore, the workstation is busy running the simulations and restrains normal usage. A job scheduler to run several simulations overnight is typically not included making the usage of these workbenches improper for large amount of simulation runs.

A large amount of simulation runs is necessary when the evaluation of a vast amount of design alternatives is required. Such alternatives concern among others granularity of components, deployment units, or deployment topologies (Koziolk/Koziolk/Reussner, 2011; Willnecker/Krcmar, 2016). The number of possible simulations increases exponentially with the number of possible decisions (Willnecker/Krcmar, 2016). We propose a distributed and scalable simulation service in order to cope with these increased resource requirements. This service runs on an application server, requires less overhead per simulation, and can schedule and execute a vast amount of jobs in parallel. Furthermore, the system is designed to auto-scale to the required number of application servers as long as resources are available. We call this service SaaS: Simulation as a Service<sup>1</sup>.

## 9.2 Related Work

Guo et al (2011) designed a service-oriented architecture for simulation services, named SimSaaS (Guo/Bai/Hu, 2011). Although, technology has evolved since 2011, the core architecture principles of SimSaaS can be applied to the here presented service.

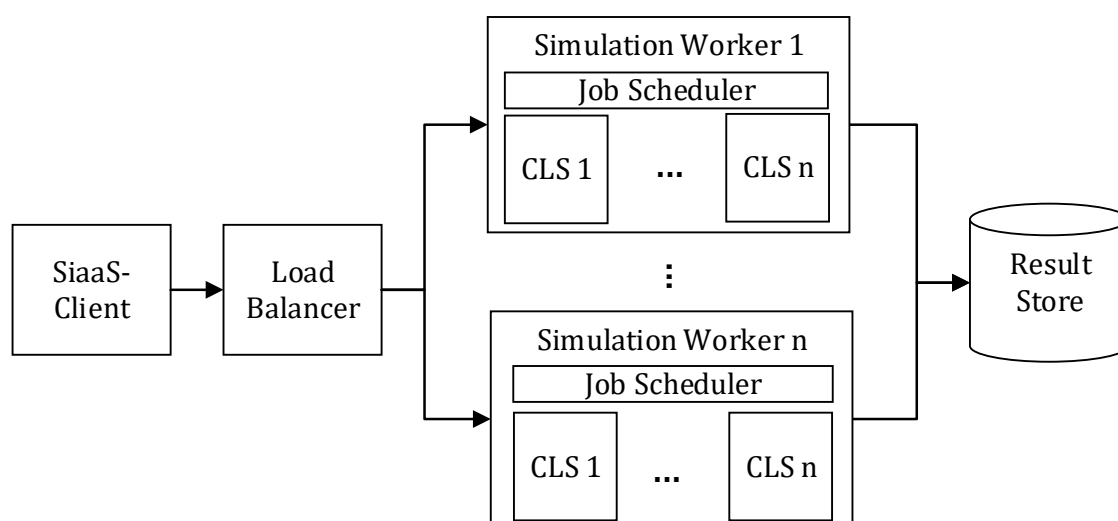
Dlugi et al. (2015) presented a headless Palladio Component Model (PCM) simulation engine in order to integrate it in a continuous delivery pipeline (Dlugi/Brunnert/Krcmar, 2015). Their work is based on a headless Eclipse<sup>2</sup> instance requiring no user interface. This headless simulator runs on a build server instance and is started via command line. Our work reuses an advanced version of this Command Line Simulator (CLS) to execute simulations on application servers.

The microservices architecture style became increasingly popular as a core architecture principle (Balalaie/Heydarnoori/Jamshidi, 2016). Small, manageable and independent services are easier to develop and to maintain (Balalaie/Heydarnoori/Jamshidi, 2016).

---

<sup>1</sup><http://pmw.fortiss.org/tools/siaas/>

<sup>2</sup><http://eclipse.org/>



**Figure 9.1:** *Simulation cluster architecture*

Therefore, we adopt the service thought but implement it in a modern architecture style for simulating PCM instances.

## 9.3 Simulation Service

SiaaS is based on Java Enterprise Edition (EE) and is accessed using a RESTful service Application Programming Interface (API). We use the Java API for RESTful Web Services (JAX-RS) and Contexts and Dependency Injection (CDI) to provide the API either run on a standard Java EE application server or packaged as Wildfly-Swarm<sup>3</sup> Microservice (Balalaie/Heydarnoori/Jamshidi, 2016). Therefore, this service can be part of a larger service infrastructure as in the Performance Management Work (PMW) tools<sup>4</sup> architecture. An instance can be run in a docker<sup>5</sup> container inside a kubernetes<sup>6</sup> instance to scale out automatically (Verma et al., 2015).

The complete SiaaS architecture as depicted in Figure 9.1 consists of three core components: (i) A load balancer distributing the jobs among the simulation worker instances, (ii) one or many worker executing simulation jobs, and (iii) a shared result store saving the simulation results. When SiaaS is run in a kubernetes cluster the load balancer is automatically part of each instance and becomes obsolete in such a setup as depicted in Figure 9.2 (Verma et al., 2015).

We designed SiaaS to support multiple PCM versions in parallel, so that older and newer versions can be run on the same service cluster and to test extensions not yet part of

<sup>3</sup><http://wildfly-swarm.io/>

<sup>4</sup><http://pmw.fortiss.org>

<sup>5</sup><http://www.docker.com/>

<sup>6</sup><http://kubernetes.io/>



the PCM release in a larger scale. A SiiaaS cluster consists of SiiaaS worker instances. Each worker instance has a job scheduler, which controls a number of independent CLS instances thus control simulation jobs in separated processes. This architecture allows to separate simulation instances from each other on the same SiiaaS instance without common dependencies.

The CLS uses a headless Eclipse instance started by a shell script (Dlugi/Brunnert/Krcmar, 2015). As input the script requires a complete PCM model, the simulation time, the maximum available heap size for one simulation, and the name of the used simulation engine (Dlugi/Brunnert/Krcmar, 2015). It currently supports the two main simulation engines of PCM: SimuCom and EventSim (Becker/Koziolok/Reussner, 2009). It is important that all SiiaaS worker of a SiiaaS cluster use the exact same CLSs. This can be achieved by storing the simulators in a common folder. The results of a simulation run are stored as a file archive to the disk and can be collected when a simulation job is finished. This archive is stored in the results store, which is a common file share between all instances. The same share can be used to store the CLSs.

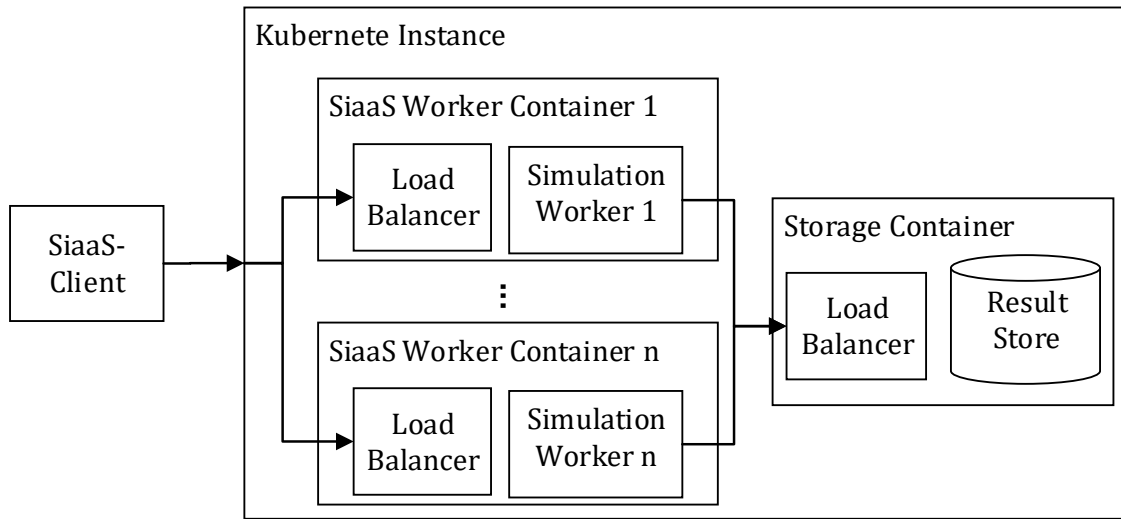
The CLS is controlled by the job scheduler of the SiiaaS. Each simulation worker controls a local job scheduler. When a new simulation is triggered, the job scheduler receives the necessary parameter to execute the simulation. If different versions of the CLS are available the correct version must be selected when starting a job. The job scheduler starts a new process using the selected CLS and assigns a unique job ID based on the session ID of the client. The scheduler continuously monitors the status of the job and provides this information via the SiiaaS REST interface. Valid job status are *started*, *running*, *finished*, *failed* or *queued*.

The job scheduler is resource-aware and checks the available memory resources of the host before a new job is started. A job will be queued if less memory is available then necessary to execute a simulation. The queued job will be started as soon, as enough memory is available, after other simulation jobs have finished or failed. This allows clients to upload a batch of jobs which are automatically scheduled and executed based on the available resources.

SiiaaS can run in a docker container inside a kubernetes instance (Verma et al., 2015). New SiiaaS workers can be spawned as soon as jobs need to be queued out of resource shortages. This allows SiiaaS to adapt to the current load and scale out to the necessary number of instances. These two technologies are leveraged to deploy SiiaaS as an elastic microservice (He et al., 2012). Figure 9.2 depicts the deployment structure when using docker and kubernetes (Verma et al., 2015).

SiiaaS is controlled and monitored using a REST interface. This allows to integrate SiiaaS into all sorts of applications independent from technology or platform constraints. The REST interface of SiiaaS provides four main interfaces: (i) schedule a new simulation job, (ii) get status of simulation job, (iii) abort a simulation job and, (iv) retrieve the results of a simulation job.

Scheduling simulation jobs requires to submit a model, the simulation time and the identifier of the used CLS. This initial request returns the unique job ID of the newly scheduled



**Figure 9.2:** *Simulation cluster in kubernetes instance*

simulation job. The job ID is the only parameter necessary for the interfaces *ii-iv*. As long as a job is running, only the computing worker can provide the job status correctly. Therefore, it is important that the client is routed to the same worker instance where the simulation job was initially scheduled. Valid routing is enforced by session stickiness of the load balancer, while each new job creates a new session. The client must send its session ID with every status or abort request, in order to get a correct answer. Regular status polls must be executed by the client to avoid session timeouts. Session timeouts can occur, if a simulation runs longer without intermediate status polls. However, results are still stored in the *Result Store* after the job is finished and are available from all worker instances.

## 9.4 Evaluation

SiaaS is part of the PMW tools. It has been used and evaluated as a simulation cluster for two other PMW tools projects: (i) Deployment Unit Optimizer (DUO) and Load Test Selector (LTS) (Willnecker/Krcmar, 2016; Vögele et al., 2015).

The DUO project automatically selects an optimal deployment topology for an Enterprise Application (EA) for a given set of resource containers (Willnecker/Krcmar, 2016). The number of potential topology grows exponentially with the number of deployable components and the number of resource containers available (Willnecker/Krcmar, 2016). Therefore, a vast amount of simulations is required to detect an optimal solution (Willnecker/Krcmar, 2016). SiaaS computed about 1200 simulations on 4 nodes in 8 hours using the SimuCom engine (Willnecker/Krcmar, 2016). SiaaS was deployed in a fixed setup, with 4 application servers and one Apache Webserver<sup>7</sup> as load balancer. DUO acted as a client

<sup>7</sup><http://httpd.apache.org/>

and queued new simulation runs based on the results from previous runs. The results were collected and analyzed and evaluated by DUO.

LTS searches for load tests design candidates matching given performance objectives like finding a minimum test set with a good component test coverage and/or high resource utilization. LTS uses a single SaaS instance as simulation service. As with DUO new simulations are triggered by the results from simulations computed previously, which are analyzed and evaluated by LTS. LTS uses the SimCom engine with another CLS as DUO as different PCM versions are required (Vögele et al., 2015).

First tests using EventSim instead of SimuCom finish already in about 30% of the time a SimuCom simulation took for the same models. Thus, future usage of SaaS will leverage the speed increase to compute more simulation runs in a shorter period of time or with less resources.

## 9.5 Conclusions

We showed a scalable simulation service called SaaS as part of the PMW tool chain. SaaS can compute PCM simulations as a distributed service that is resource aware and auto scales to necessary size when run in a kubernetes instance. SaaS is controlled via a simple REST interface allowing developers to easily integrate SaaS into their tool chain. We demonstrated this with two PMW tools: DUO and LTS, both using SaaS as distributed simulation service.

SaaS is able to support multiple versions of PCM CLSs. Therefore, SaaS is a multi-tenant application, which allows to use the same service instance for multiple purposes as demonstrated by DUO and LTS. Furthermore, this feature allows to run SaaS as a Software as a Service (SaaS) application for multiple institutes or to run tests of multiple PCM instances. The general architecture allows developers to integrate other performance simulations or solving techniques for performance models.

Future work mainly concerns integrating SaaS into other tools and increase the stability of the service and its infrastructure components. Furthermore, the resource-awareness of SaaS can be extended to consider CPU utilization instead of only memory consumption for scheduling jobs and spawning new simulation instances.

# Chapter 10

## Multi-Objective Optimization of Deployment Topologies for Distributed Applications

Authors	Willnecker, Felix <sup>1</sup> (willnecker@fortiss.org) Krcmar, Helmut <sup>2</sup> (krcmar@in.tum.de)  <sup>1</sup> fortiss GmbH, Guerickestraße 25, 80805 München, Germany <sup>2</sup> Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	ACM Transactions on Internet Technology
Status	Accepted
Keywords	Deployment Topology Optimization, Performance Model, Distributed Enterprise Applications, Performance Model Generation, Memory Simulation
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

**Table 10.1:** *Bibliographic details for P7*

**Abstract** Modern applications are typically implemented as distributed systems comprising several components. Deciding where to deploy which component is a difficult task that today is usually assisted by logical topology recommendations. Choosing inefficient topologies allocates the wrong amount of resources, leads to unnecessary operation costs, or results in poor performance. Testing different topologies to find good solutions takes a lot of time and might delay productive operations. Therefore, this work introduces a software-based deployment topology optimization approach for distributed applications. We use an enhanced performance model generator that extracts models from operational monitoring data of running applications. The extracted model is used to simulate performance metrics (e.g., resource utilization, response times, throughput) and runtime costs of distributed applications. Subsequently, we introduce a deployment topology optimizer, which selects an optimized topology for a specified workload and considers on-premise, cloud, and hybrid topologies. The following three optimization goals are presented in this work: (i) minimum response time for an optimized user experience, (ii) approximate

resource utilization around certain peaks, and (iii) minimum cost for running the application. To evaluate the approach, we use the SPECjEnterpriseNEXT industry benchmark as distributed application in an on-premise and in a cloud/on-premise hybrid environment. The evaluation demonstrates the accuracy of the simulation compared to the actual deployment by deploying an optimized topology and comparing measurements with simulation results.

## 10.1 Introduction

Distributed architectures are state of the art in large-scale and modern internet applications (Brunnert/Krcmar, 2017; Brunnert et al., 2015). Such applications typically comprise multiple deployment units. Each deployment unit is composed of several components and is movable from one server instance to another. Furthermore, these units can be replicated to cope with increased workload based on data from operations. Selecting the right amount of deployment unit replications and corresponding runtime instances is a difficult task and requires developers (dev) and operations (ops) expertise. Numerous different combinations, so-called deployment topologies, exist (Brunnert et al., 2015). Not only the right amount of unit replications must be selected, but also the right amount of resource containers (e.g., Virtual Machines (VMs), bare-metal server, or application containers), which depends on the demand a component operation places on its resources (Brunnert et al., 2015). The most important resources, including Central Processing Unit (CPU), Hard Disk Drive (HDD), memory and network, and their demands have to be considered for deployment topology decisions (Koziolek et al., 2014; Brunnert et al., 2015).

At present, these decisions are assisted by logical recommendations or simply based on estimations instead of continuous measurements from operations (Brunnert et al., 2015). Such estimations and resulting topologies often rely on peak demands, which leads to under-utilized data centers (Speitkamp/Bichler, 2010). Different studies estimate the current average CPU utilization in data centers between 6% and 20% (Huang/Masanet, 2015; Speitkamp/Bichler, 2010). Hence, servers are idle for most of their uptime. This situation serves well for Infrastructure as a Service (IaaS) cloud providers as they can over-provision their physical capacities, but the situation produces unnecessary costs for operators of distributed applications (DAs). By contrast, over-utilized servers are not desirable as this results in overly long response times or unstable systems. Therefore, selecting the right amount of resource containers and optimizing the utilization of their resources is important when running DAs (Ardagna et al., 2014).

Instead of deploying a DA on-premise, managed infrastructures like cloud environments are available today and can provide extensive reliability and cost reduction (Ardagna et al., 2014). Managed infrastructure providers now invoice the usage of runtime instances. In contrast to previously purchased servers, these providers charge current costs. Thus, DA operators have a vested interest in optimizing their topologies in order to reduce operation costs. Different providers apply different cost models based on the number of machines, requests processed, number of user sessions, or simply aggregated uptime. Depending on

which provider is chosen, the optimization goal for the deployment might change to save costs. A high utilization might be preferred if the uptime of servers forms the basis of the cost model instead of optimized response times.

In practice, deployment topology considerations require a great amount of effort and expert knowledge about operation data and the software that has been developed. The role of the DevOps engineer is designed for these type of tasks. Planning and testing topology changes in a production environment may incur risks for the stability of the DA. The potential savings in operation costs or performance gains compared to the risks might not be worth it. Evaluating topologies in test environments requires production-like environments, although these environments are as expensive as the production environment itself. Furthermore, such testing environments are often used to capacity by various projects executing load tests or may simply not yet be available when new hardware or managed infrastructures like cloud environments are introduced as new target environments (Ardagna et al., 2014).

This work proposes to use performance models extracted from small scale test environments and subsequently size and optimize available resource containers to specified workloads. To accomplish this goal, we combine performance model generators (PMGs) and architecture optimizers to automatically detect optimized deployment topologies. This approach allows to use accurate resource demands from generated performance models and established architecture optimization algorithms. We use the PMG of the RETIT<sup>1</sup> Capacity Manager, the Palladio Component Model (PCM) as performance meta-model, and an opt4j based approach to optimize the deployment topologies for DAs (Brunnert/Krcmar, 2017; Koziolok/Koziolok/Reussner, 2011; Lukasiewicz et al., 2011).

Performance models can be used to predict performance metrics by evaluating alternative deployment topologies and resource environments, and simulate the effects on these metrics (Brunnert/Krcmar, 2017). Building such models manually often outweighs any potential benefit (Kounev, 2005). Recent research created PMGs for DAs to limit the effort of building such models (Brunnert/Krcmar, 2017). However, no currently available PMG considers all four main resources (CPU, HDD, memory, and network). The PMG we use has a comprehensive approach, but lacks automatic memory management simulation. Therefore, we introduce an extension to PCM and the contemplated PMG for dynamic and automatic memory management modeling and simulation.

PMGs focus on the extraction of the software architecture of a DA but disregard deployment topology decisions. Selecting the right amount of resources and evaluating the selected topologies again requires manual effort. Manual selection soon becomes impossible as the number of potential topologies grows exponentially with the number of deployment units and available resource containers (Koziolok/Koziolok/Reussner, 2011). To automate this process architecture optimizers have been introduced to the scientific community (Aleti et al., 2013; Koziolok/Koziolok/Reussner, 2011). These optimizers require an already created (performance) model to conduct optimizations (Koziolok/Koziolok/Reussner, 2011). Such models can be derived from design specifications or created manually. However, the actual resource demands are usually estimated and therefore error-

---

<sup>1</sup><http://www.ret.it.de/>

prone. While generated performance models provide high accuracy for predictions, they are not yet compatible with architecture optimizers. Development and evaluation of an automated approach based on PMG and architecture optimizers to identify ideal deployment topology is the main contribution of this work.

Our results allow DevOps engineers to evaluate different resource environments (e.g., in-house, hosted, cloud), to evaluate different deployment topologies, and to automatically size DAs without deploying the application in a production or production-like environment. For the evaluation we conducted a series of controlled experiments using the industry standard benchmark SPECjEnterpriseNEXT<sup>2</sup> as DA.

To what is already known in this area we contribute:

- (i) An automated approach to identify optimal deployments.
- (ii) The combination of performance model generation and architecture optimization.
- (iii) System design and evaluation of automatic deployment topology selection in different resource environments.
- (iv) A dynamic and automatic memory management simulation approach.
- (v) A cost model for on-premise and IaaS cloud environments.

This paper builds on our previous work (Willnecker/Krcmar, 2016; Willnecker et al., 2015b) on deployment topology optimization and contains the following major improvements and extensions:

- (i) Multi-objective optimization that calculates the Pareto-front along the optimal results for three goals: minimum response time, minimum costs, optimal resource utilization.
- (ii) A flexible cost-model for cloud and on-premise environments based on actual usage of the resources.
- (iii) An extension for our garbage collection approach that allows growing and shrinking committed memory.
- (iv) An evaluation of a newer and more complex SPECjEnterpriseNEXT version in an on-premise and industry cloud environment.

## 10.2 Related Work

Early PMGs have been demonstrated in the work of Hrischuk et al. in 1999 (Hrischuk/Woodside/Rolia, 1999). Their work focuses on layered queueing networks (LQNs) which do not separate workload, software components and resource environments (Hrischuk/Woodside/Rolia, 1999). Without such a separation, exchanging the resource environment model or changing the deployment topology is difficult to accomplish. Therefore,

---

<sup>2</sup>SPECjEnterpriseNEXT is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjEnterpriseNEXT 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.

architecture-level performance models, such as PCM, introduce separated sub-models for workload, software architecture and resource environments (Becker/Koziolk/Reussner, 2009). The work of Brosig et al. (2014) generates performance models for PCM and simulates CPU, HDD and memory demands, but lacks automatic memory management and network demands (Brosig/Huber/Kounev, 2014). Especially in distributed environments the network latency and bandwidth can have a huge impact on the performance of the system (Brunnert/Krcmar, 2017).

Another performance model generation approach has been introduced by Brunnert et al. (2015) (Brunnert/Krcmar, 2017). The generated models are called resource profiles and consider CPU, HDD, and network demands (Brunnert/Krcmar, 2017). The approach generates accurate models from running DAs but lacks automatic memory management (Brunnert/Krcmar, 2017). Thoroughly conducted capacity planning requires taking the memory resource into account in order to charge the capacity of available systems effectively. We extend this generator with automatic memory management simulations in our deployment topology optimization architecture.






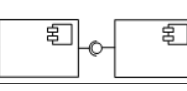

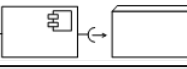

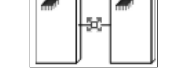
Huber/Brosig et al. (2016) describe a performance model-based approach for self-aware systems (Huber/Brosig et al., 2016). An endless monitoring, model deduction, and prediction loop allows the constant optimization of the performance of an application system (Huber/Brosig et al., 2016). The effects rely on the quality of the workload prediction. If the correct workload for a certain timeframe is predicted, the topology can be adjusted. Nevertheless, complex optimizations are often prohibited as decisions must be made fast.

Speitkamp/Bichler (2010) identified the need to consolidate resource usage and proposed a mathematical model to optimize resource allocation using VMs (Speitkamp/Bichler, 2010). VMs with high CPU utilization could be run on the same host together with VMs utilizing other resources having low CPU utilization (Speitkamp/Bichler, 2010). This concept should optimize the utilization of all resources in a data center. However, the proposed model is not aware of the workload or the DAs running in the VMs and their dependencies. The model requires a re-calculation and allocation of the VMs when the resource utilization of the hosted applications changes. Such changes occur frequently as new versions are deployed or the executed workload changes.

Chen et al. (2015) demonstrate a tool called StressCloud to model and optimize cloud deployments. The focus on performance and energy consumption as well as load test generation (Chen et al., 2015). This approach allows to model workload, software and target infrastructure in a new meta-model (Chen et al., 2015). The accuracy of such approaches relies on the quality of the model. A model generation approach based on measurement results may increase the quality of the StressCloud models.

The architecture optimization approach PerOpteryx evaluates design alternatives based on PCM models (Koziolk/Koziolk/Reussner, 2011). The number of decisions is large as hardware, network, and software architecture are taken into account. PerOpteryx has a broad variety of optimization goals and degrees of freedom. We adapt the PerOpteryx approach for deployment optimization but with certain changes (e.g., simulations instead of analytical solvers and limitation of deployment topology decisions). This adaptation



	Concept	Java EE	Palladio Component Model
Workload	Number of Users and User Behavior	Real Users or Virtual Users 	Usage Model 
	Components and Operations	EJBs, JSPs, Servlets 	Repository Model 
Resource Profile	Deployment Units	EARs/WARs 	System Model 
	Deployment Unit to Resource Container Relationship	Installation on Application Server 	Allocation Model 
Deployment Topology	Resource Container, Resources and Network Conn.	Virtual Machine/ Bare-Metal Server 	Resource Env. 

**Figure 10.1:** *Distributed application structure adapted from (Becker/Koziolek/Reussner, 2009; Brunnert/Krcmar, 2017)*

allows us to not only evaluate CPU utilization and response times, but also to take network, automatic memory management, and HDD demands into account.

Aleti et al. (2013) provide a comprehensive review on software architecture optimization methods (Aleti et al., 2013). Their work recommends providing evidence for resulting architectures in order to prove the validity of these strategies and suggests establishing holistic tool support (Aleti et al., 2013). Current work fails to provide evidence as the presented approaches are difficult to compare to real environments. Our work addresses this gap by evaluating architecture optimization with a real DA and combining architecture detection (performance model generation) with architecture optimization into a holistic tool for automatic deployment topology optimization.

### 10.3 Distributed application components

A deployment unit is a packaged artifact installable on a server instance or directly on an operating system (OS). Such units consist of several components and operations and

build the core of any DA. A typical DA consists of many deployment units distributed throughout multiple servers.

In order to analyze the deployment topology of a DA, its context has to be taken into account. The context comprises (i) the resource profile of the DA and (ii) the workload executed on the DA. Hence, an optimized topology always depends on both factors. Figure 10.1 depicts the main components of a DA. We use Java Enterprise Edition (EE) as an example middleware even though the depicted concepts are applicable for other technologies as long as monitoring technology is available (Spinner et al., 2015; Willnecker/Krcmar, 2016). Figure 10.1 also shows how we map the different parts of a DA to the PCM meta-model. We selected PCM as it is a mature and stable meta-model and corresponding simulation environment. Our previous technology, like the PMG, extensions to simulate HDDs, and network have already been applied to PCM and we can easily reuse these accomplishments (Becker/Koziolok/Reussner, 2009; Brunnert/Krcmar, 2017).

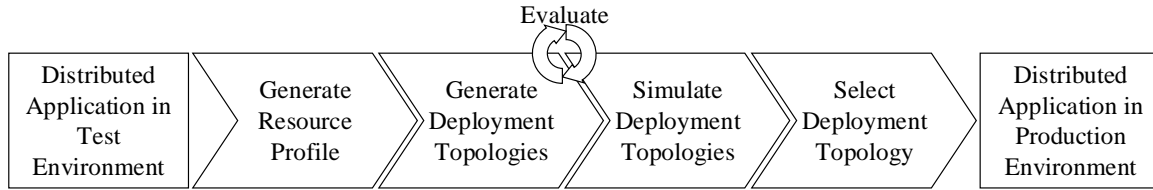
The workload describes the number of users and how they use the DA, which ultimately causes the resource utilization of the DA. These users can be real users accessing the system or virtual users executing a (load) test to analyze the behavior of a DA. We use Application Performance Management (APM) data during a monitoring run to derive an initial workload. This workload is editable in order to size a DA according to the expected workload in production.

The resource profile of a DA describes how an operation of a component utilizes different resources (Brunnert/Krcmar, 2017). The profile consists of a basic workflow and the deployment unit structure of the DA. Each operation of the DA is modeled in the profile including its resource demands for several resources. Resource profiles are the core result of PMGs (Brunnert/Krcmar, 2017).

The runtime of the DA representation is defined by the deployment topology. For each deployment unit of a DA at least one instance exists during runtime. If required by the workload, replicas of a deployment unit might exist. If depicted as a graph, each deployment unit node needs at least one edge to a resource container node. The node is replicated if multiple edges from a deployment unit node exist.

A deployment topology describes the structure and relationship of a set of these deployment units installed and executed on a number of resource containers. The containers are organized in the so-called resource environment model. This environment consists of the hosting machines, their capabilities (e.g., CPU processing rate, available memory, HDD speed), and the network connections (focusing on bandwidth and latency) between the resource containers. Two deployment units, which are dependent, can only be deployed on two resource containers that are linked via a network connection. Therefore, the number of potential topologies depends on the number of deployment units  $du$  and the valid resource containers  $rc$ . Equation 10.1 calculates the number of possible deployment topologies depending on  $du$  and  $rc$ , when all resource container targets are valid for all deployment units.

$$DT_{du,rc} = (2^{rc} - 1)^{du} \quad (10.1)$$



**Figure 10.2:** *Deployment topology optimization process.*

$2^{rc} - 1$  describes each possible installation combination of a deployment unit on one or more resource containers. As any permutation with other deployment unit installations is possible, we have to add  $du$  as an exponent. Given 10 resource containers and 5 deployment units, the number of possible topologies is already greater than  $10^{15}$ . The number of combinations in this scenario prohibits a manual selection. Deployment topology optimization requires an automated approach.

## 10.4 Deployment Topology Optimization Process

An automated approach requires a holistic tool to optimize deployment topologies, which comprises three basic components:

- (i) Performance model generator to detect the resource profile of a DA including its resource demands, system behavior, current deployment topology, and current workload.
- (ii) Architecture optimizer to evaluate different target deployment topologies and to select the best topology in terms of the optimization goal.
- (iii) Simulation service for parallel predicting performance metrics of multiple performance models.

Figure 10.2 illustrates the optimization process. We deploy a DA in a test environment to conduct the process. This DA is instrumented with APM agents and set under load in order to obtain meaningful APM data. In a first step, this APM data is used to generate a performance model. The model consists of the detected workload, the detected resource profile, and a specification of the resources in the test environment. The generated model represents the current state of the DA in the test environment.

In a second step, the architecture optimizer based on optj4 uses an evolutionary algorithm for selecting an initial number of possible topologies (initial population) (Lukasiewicz et al., 2011). This initial population set is based on the generated model and the available resource containers. Each topology is validated in order to check if all deployment units are at least instantiated once and can communicate with all dependent deployment units via a network connection. The deployment topologies are packaged for simulation including workload and resource profile. We constructed a distributed simulation cluster that can simulate multiple topologies in parallel.

After each simulation run the optimizer evaluates the results. We currently support three optimization goals: (i) approximate the mean resource utilization per resource over of all containers to a certain level (e.g., 70% CPU utilization and 80% memory utilization), (ii) minimize the costs, or (iii) minimize the response time per transaction. A topology can be invalidated if one resource is utilized above a certain threshold to prevent over-utilizing certain containers. The optimizer mutates new topologies based on the evaluation results and delegates the simulation. This process is repeated until the initial population and the number of generations are processed.

The best topology in terms of the optimization goals is a Pareto-front. Therefore, multiple solutions are possible and can be selected by the DevOps engineer. The final step is deploying this topology in the production environment.

## 10.5 Performance Model Generator

This section explains the PMG and extensions we added to support generating a comprehensive performance model. The generated model considers the most important resources: CPU, HDD, network and memory. Furthermore, a transaction flow throughout a DA is detected and used for reasoning the systems control-flow.

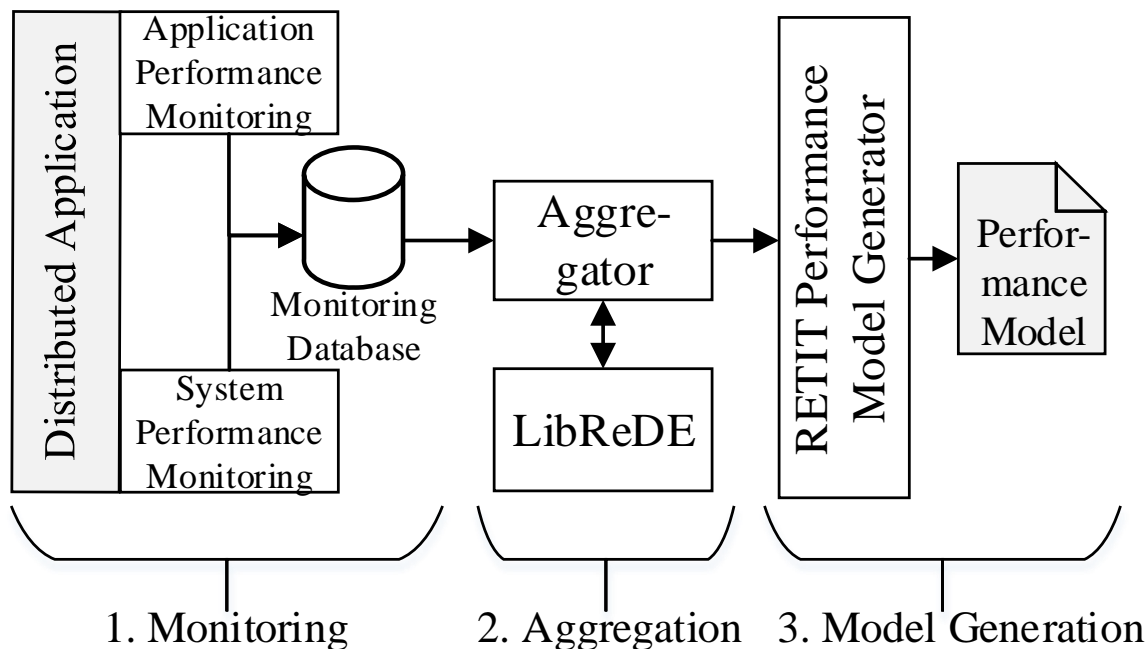
We use and extend the PMG of the RETIT Capacity Manager and the corresponding monitoring solution RETIT Java EE. Both solutions are based on the Performance Management Work Tools (PMWT) PMG introduced by Brunnert et al. in 2015 (Brunnert/Krcmar, 2017). As depicted in Figure 10.3(a), the generation process consists of three phases (Willnecker/Krcmar, 2016):

- (i) monitoring the instrumented DA,
- (ii) aggregating the monitoring data per operation, and
- (iii) generating the performance model based on the aggregated monitoring data.

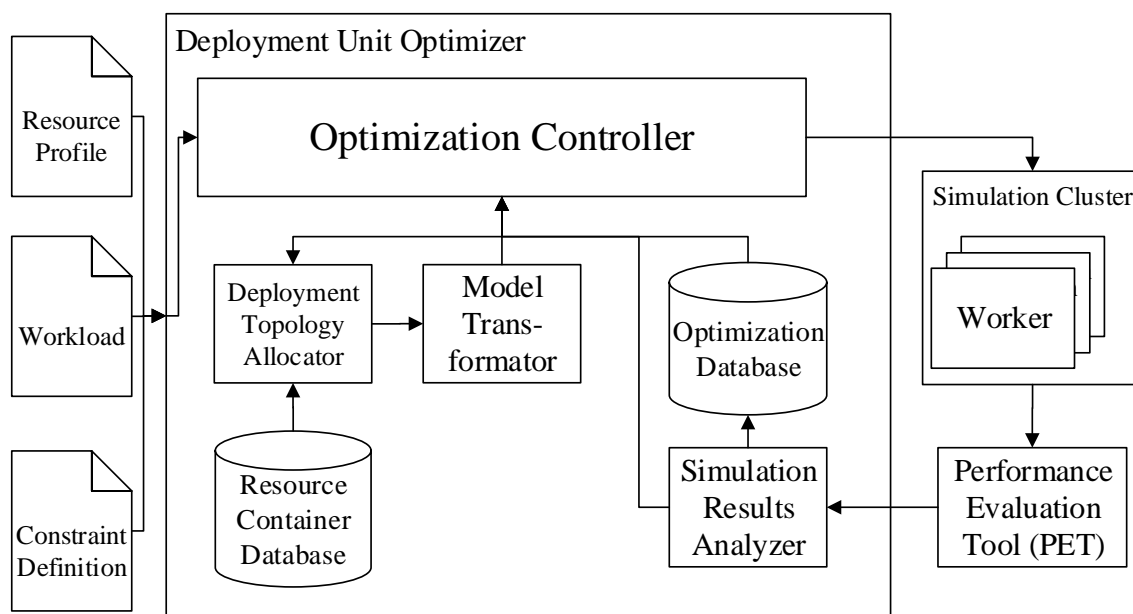
The result of the three phases is a resource profile, a workload description representing the usage of the DA during the monitoring phase, and a resource environment describing the current deployment of the DA. All three model parts are stored as a PCM instance (Brunnert/Krcmar, 2017; Becker/Koziolek/Reussner, 2009).

### 10.5.1 Monitoring

The monitoring step collects operation invocations of the instrumented DA. We distinguish between resource demand measurement and resource demand estimation (Spinner et al., 2015; Willnecker/Krcmar, 2016). Resource demand measurement uses fine-grained monitoring data per operation invocation to measure the exact demand an operation places on a resource. These measurements can be collected with standard APM soft-



(a) Performance model generator (adapted from (Willnecker/Krcmar, 2016))



(b) Deployment Unit Optimizer (adapted from (Willnecker/Krcmar, 2016))

**Figure 10.3:** Core subsystem of deployment topology optimization

ware like Dynatrace<sup>3</sup> Application Monitoring (AM) (Willnecker/Krcmar, 2016). Resource demand estimation uses coarse-grained monitoring data like total resource utilization and response time series per operation and distributes the utilization throughout the operations (Spinner et al., 2015). Such coarse-grained resource utilization data can be collected using standard system monitors like System Activity Reporter (SAR), or monitoring and control interfaces of virtual machines like Java Management Extensions (JMX). Load drivers like jMeter<sup>4</sup> or access logs of web servers provide response time series of operations invoked on system-entry level. For more detailed (e.g., component-level) response time series, custom filter or logger are necessary.

The PMG supports data from different data sources as depicted in Figure 10.3(a):

- (i) Application Performance Monitoring for fine-grained application data. We use the RETIT Java EE Monitoring solution in this work. Previous work demonstrated the applicability of industry standard solutions like Dynatrace AM (Willnecker/Krcmar, 2016).
- (ii) System Performance Monitoring for coarse-grained application data. Standard system tools or custom host agents are possible. We use Apache Webserver<sup>5</sup> access logs, RETIT Host Monitoring and JMX in this work.

The collected data is stored ongoing in a monitoring database based on the Apache Cassandra<sup>6</sup> project. The large amount of data requires a scalable, yet simple database structure. Each row in the database corresponds to an operation invocation or a measurement record from system monitoring. The next phase uses this monitoring database as a single source of input.

## 10.5.2 Aggregation

The aggregation phase concentrates all the single operation invocations as a preparation for the model generation. The mean demand per resource (e.g., mean CPU demand) is calculated for every operation of every component. Furthermore, the calculation of network demands at the deployment unit boundaries and transaction flows based on unique transaction IDs is conducted in this step.

Figure 10.3(a) shows the currently supported data sources in the monitoring step; other APM solutions or coarse-grained monitoring providers can be added to the PMG. The demand calculation is either done by the aggregator, or delegated to Library for Resource Demand Estimation (LibReDE) in a post-processing after initial aggregation (Spinner et al., 2015). After this phase, all operation invocations, their resource demands, and the transaction flow are prepared for the model generation phase.

---

<sup>3</sup><http://www.dynatrace.com/>

<sup>4</sup><http://jmeter.apache.org/>

<sup>5</sup><http://httpd.apache.org/>

<sup>6</sup><http://cassandra.apache.org/>

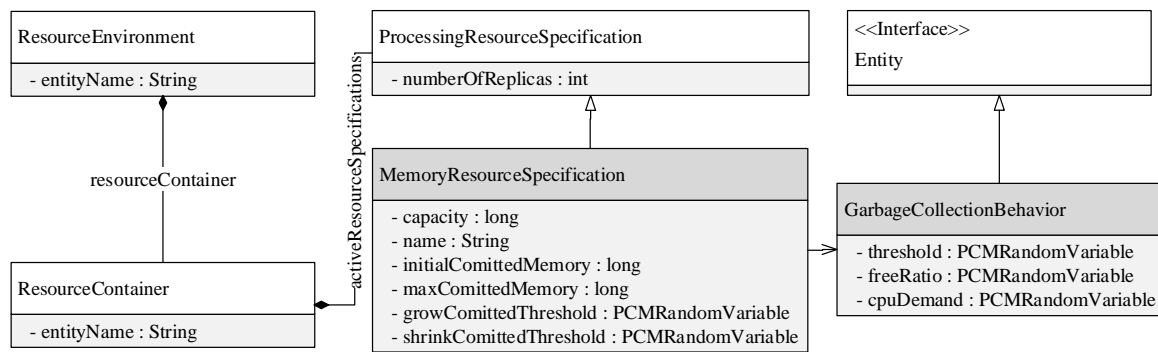


Figure 10.4: PCM extension for memory resources.

### 10.5.3 Model Generation

This final phase transforms the operation invocations, resource demands, and the transaction flow into a PCM instance. This transformation allows changing and/or simulating, hence predicting the system behavior using the Palladio-Bench (Becker/Koziolek/Reussner, 2009). PCM supports CPU, HDD and network demands. The built-in features of PCM are used for the above mentioned three resources (Brunnert/Krcmar, 2017).

Each operation invocation results in at least one action with a CPU demand. In contrast, for HDD demands we distinguish between write and read demands as the write speed of a HDD is very different from the read speed (Brunnert/Krcmar, 2017). Furthermore, we use the mean request and response size to simulate the time this request travels through the network based on available bandwidth and latency (Brunnert/Krcmar, 2017).

Memory demands and simulation are more complex due to automatic memory management (Libič et al., 2015). We can calculate the mean memory demand of an operation by measuring the memory demand of each operation invocation and forming the average demand similar to CPU demands. A dedicated resource is necessary to place the demands, which supports automatic memory management scenarios as in virtualized runtimes like the Java Virtual Machine (JVM).

We extended the PCM meta-model in order to add a memory resource representation as depicted in Figure 10.4. This resource works for dynamic memory management scenarios and supports different types of automatic memory management methods, like Garbage Collections (GCs). GCs delay the release of memory, leading to a larger memory utilization during runtime. Thus, memory is more likely to become a bottleneck. We extended PCM to simulate this effect (Becker/Koziolek/Reussner, 2009; Willnecker/Krcmar, 2016). We added two classes to the meta-model to support this behavior:

- (i) *MemoryResourceSpecification* to specify the attributes of a memory resource.
- (ii) *GarbageCollectionBehaviour* to define the behavior of automatic memory management. No behavior is specified in dynamic memory management scenarios.

A typical GC collects and stores released objects in different memory spaces (Libič et al., 2015). The spaces are cleaned in different intervals. For instance, the JVM executes two types of GCs (minor and major) to clean different spaces or promote objects to another space (Libič et al., 2015). A memory simulation containing garbage collection requires monitoring GC events and generating instances of the memory resource and the GC behavior in PCM.

For Java EE, we monitor the GC events of the running application using the `GarbageCollectorMXBean`<sup>7</sup> via JMX and measure the following metrics:

- (i) The type of garbage collection that is executed. For Java EE this is either a minor GC or a major GC. Other GC implementations or technologies can have different GC types.
- (ii) Size of total memory available in the JVM.
- (iii) Size of allocated memory before and after the GC execution. This is a simplification of the actual mechanism as we do not simulate object movements in the fine-grained GC spaces. This probabilistic approach enables automatic memory management simulation with low overhead compared to complex object movement simulations (Libič et al., 2015; Willnecker/Krcmar, 2016).
- (iv) CPU time necessary to execute the GC.

The measurement data is aggregated and processed for the performance model generation. The resource environment generation creates a *MemoryResourceSpecification* instance per resource container. This resource contains the initial and the maximum available memory. Furthermore, an optional grow and shrink threshold is part of the resource for simulating changes to the committed memory. For dynamic memory management no further generation is conducted. For automatic memory management we extract the GC types and create a *GarbageCollectionBehavior* instance for each GC type. For each behavior instance we calculate the mean CPU demand per byte released, the mean free ratio and the threshold leading to a GC execution. Threshold and free ratio are calculated in percent and are independent from the current memory size. This implementation automatically adapts *GarbageCollectionBehavior* to other resource containers representing larger or smaller servers with less or more available memory for the JVM.

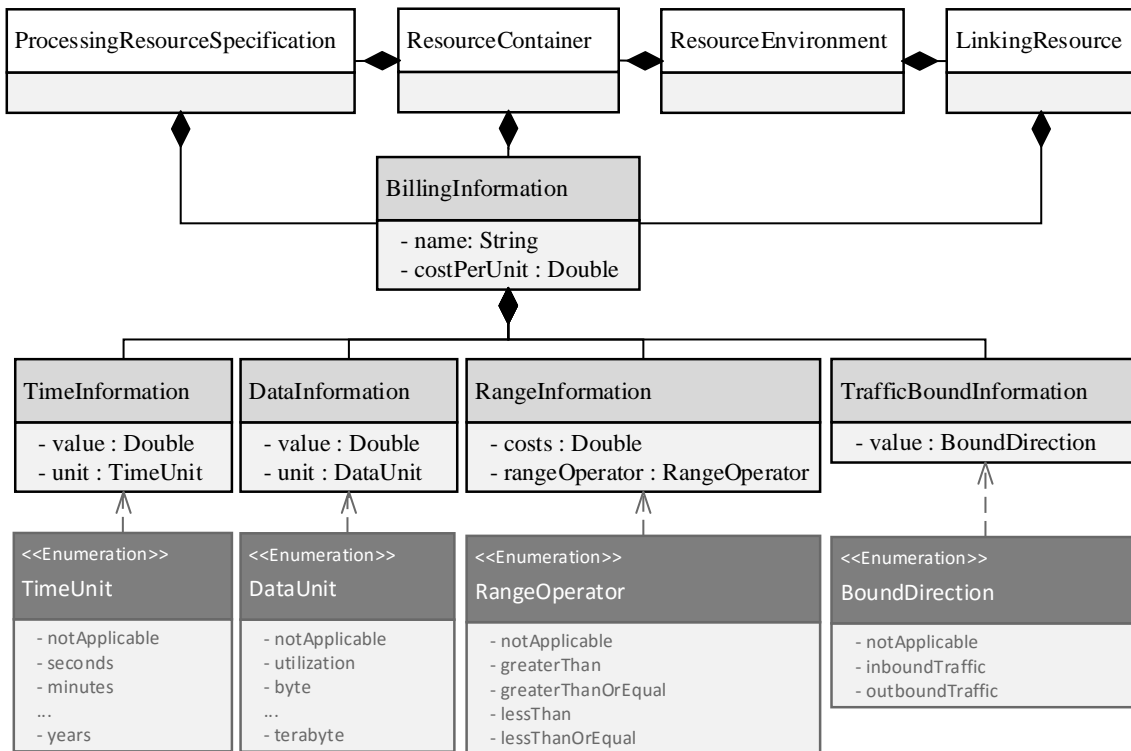
In order to access the newly introduced memory resource we extended *ResourceCalls* in PCM. We added two more signatures to execute allocation and free calls on this resource. Each operation in the performance model calls the alloc signature of the corresponding memory resource. No free call is necessary for automatic memory management as this is handled by the *GarbageCollectionBehavior*. In dynamic memory scenarios, the free operation is called after each operation. The available memory is immediately increased by the amount specified in the free call. We extended the PMG to generate such *ResourceCalls* automatically for every operation that allocates memory.

For automatic memory management a thread per memory resource is started with the simulation. This thread watches if the committed memory of the corresponding resource container exceeds the configured threshold. If the committed memory exceeds the GC ex-

---

<sup>7</sup><http://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/GarbageCollectorMXBean.html>





**Figure 10.5:** *Cost model extension*

ecution threshold, a GC run is simulated. The memory of this resource is freed depending on the free ratio of the executed *GarbageCollectionBehavior*. A CPU demand depending on the CPU demand per byte of the *GarbageCollectionBehavior* and the number of freed bytes is placed on the CPU resource of the same *ResourceContainer*.

### 10.5.4 Cost model

The costs of on-premise installations are usually flat and thus easy to calculate. The number of servers, their energy consumption, and the maintenance and administration costs are the core influencing factors. In managed infrastructures like IaaS cloud environments, the cost structure gets more complicated. The accounting items are no longer hardware and the staff maintaining it, but uptime of (virtual) instances, different types of network traffic, or average utilization of certain resources. In previous research, we reduced the number of server instances by optimizing the deployment topologies of the software running on it (Willnecker/Krcmar, 2016). We used a simple optimization goal: approximate the resource utilization as close to 70% as possible. 70% was selected as higher utilization usually leads to longer and unpredictable response times due to missing/reduced peak tolerance head room. Using 70% reduced the costs for simple cost structures in on-premise scenarios. Complex cost structures required a more sophisticated method. Therefore, we constructed a flexible cost model for on-premise, cloud and hybrid environments.

Figure 10.5 depicts our model. We connected the model to the PCM meta-model so that our performance model already contains the cost structure of the target model. Our deployment topology optimizer can now use a holistic cost and performance model and select cost effective deployment topologies for cloud environments.

We connect resource containers (e.g., VMs), resources (e.g., CPUs), and linking resources to a *BillingInformation*. Thus, each resource or container can be accounted for. The model allows attaching costs based on time using *TimingInformation* objects. This allows defining the costs over time for resource container for example. The amount of data processed or transferred can be accounted for with the *DataInformation* class. This is especially useful to model the costs of network traffic. We further distinguish between inbound and outbound traffic. Typical cloud providers offer cheaper costs per Gigabyte (GB) when the data is transmitted within the data centers of the cloud provider (*inbound*) instead of through the open Internet (*outbound*).

Several cloud providers have different pricing tiers. The first GBs of network traffic are free, the next couple of GBs are expensive, until the costs decrease for heavy users. To model these tiers, we added a *RangeInformation* allowing definition of the prices for different tiers. This comprehensive approach allows modeling and calculating the costs of deployment topologies in on-premise and IaaS cloud environments. Therefore, our optimizer can consider resource utilization, response times, and costs when searching for a Pareto-optimal topology of a DA.

## 10.6 Architecture Optimizer

The core of our deployment topology optimization approach is the architecture optimizer. We use the evolutionary computation algorithms of the opt4j framework (version 3.1.4) (Lukasiewicz et al., 2011). The technological foundation is similar to design space exploration tools like PerOpteryx, but we use simulations instead of solvers for the evaluation of design alternatives (Koziolek/Koziolek/Reussner, 2011). We chose opt4j for its collection of optimization algorithms amongst which we use the Evolutionary Algorithm module (Lukasiewicz et al., 2011). Previous work of Koziolek et al. (2014) and our own experiments produced the best results using this module (Koziolek/Koziolek/Reussner, 2011; Lukasiewicz et al., 2011). Opt4j in combination with PCM-conform performance models allows us to simulate the performance and cost effects on the topology simulating hundreds of users accessing the system. The simulation approach provides high prediction accuracy although such simulations are computational intensive compared to solver-based approaches. We constructed a simulation cluster to process several simulations in parallel on a dedicated system. This simulation cluster reduces time it takes to find an optimized solution.

The architecture optimizer requires three artifacts to conduct an optimization run as depicted in Figure 10.3(b):

- (i) The resource profile generated by the PMG containing the software architecture and resource demands for all four major resources of the DA.
- (ii) The expected workload of the system in the target environment. The topology is optimized according to this workload. In general, the workload in the target environment is expected to be higher compared to the workload executed for the model generation.
- (iii) Constraints to the optimization like the minimum/maximum utilization of a resource, the minimum/maximum number of systems, and logical constraints that can, for instance, prohibit deploying the database on a resource container that already contains an Application Server (AS).

The *Optimization Controller* acts as workflow controller and triggers all sub components of the system. In a first step the *Deployment Topology Allocator* creates a random set of valid design alternatives. The allocator considers available resource containers and the network connections between these containers stored in the *Resource Container Database*. The database consists of a list of all containers available for the optimization run. It can also consist of instances in a local data center or of instances offered by a cloud provider. The capability of the resources of these containers must be calibrated compared to the resources used during the generation. Brunnert et al. (2015) showed that benchmarks provide accurate results for transforming resource capabilities from one machine to another (Brunnert/Krcmar, 2017). We calibrate network, HDD, and CPU capabilities. This allows us to make performance predictions for an application running in another environment. Therefore, we run HDD, CPU and network benchmarks in both environments and calculate the HDD read and write speed, the relative CPU capability, and the network bandwidth and latency. This calibration has to be done for each host or VM configuration and the network they are connected to.

The initial population of the evolutionary algorithm is calculated by the *Deployment Topology Allocator* from the list of available containers, deployment units, and constraints. Our population can be represented as a  $du \times rc$  matrix  $G$ . Each column stands for one available resource container  $rc$  and each row for a deployment unit  $du$  of the DA. The cell values are either 0 or 1. A 1 in the cell  $G_{i,j}$  indicates that in this topology the deployment unit  $du_i$  is deployed on resource container  $rc_j$ . The initial topologies are created randomly but invalid selections are discarded immediately. A valid matrix has at least one 1 in each row so that each deployment unit is at least deployed once and fulfills the constraints (e.g., no database (DB) deployment unit on the same container as an AS deployment unit).

The second step is the model-to-model transformation of the matrix into a PCM instance. The matrix representation is transformed into resource environment and allocation model instance. The PMG only creates one instance per deployment unit. Therefore, additional deployment unit instances are created in the repository and system model if necessary during the transformation. The resource environment and allocation model are packaged with the resource profile and workload. The result of this process is a complete PCM instance ready for simulation.

Our evolutionary algorithm evaluates the quality of a topology in multiple components (Lukasiewicz et al., 2011). First of all, the *Optimization Controller* checks against the

*Optimization Database* if an equal topology was already simulated. A topology is considered equal if the same deployment units are distributed throughout equal resource containers. Two resource containers are considered equal if their resources have the same capabilities (e.g., same number and speed of the CPU). If an equal topology is detected, the simulation results from a previous run are returned instead of a full simulation run.

The controller dispatches a new simulation job to the simulation cluster if no equal topology has been detected. The cluster consists of a load-balancer and several worker nodes executing the Palladio-Bench in a headless Eclipse instance (Becker/Koziolok/Reussner, 2009). The load-balancer assigns the simulation job to one worker. If not enough resources for the execution are available, the simulation job is queued. The job is started when resources are free again after, for instance, another simulation job on this worker has been finished. After a job run, the worker stores the results in a shared folder. Each worker node is equally able to simulate a PCM meta-model instance and provide results for already conducted simulations.

The *Optimization Controller* sends an archive containing all model elements as depicted in Figure 10.1 to the cluster to start a simulation job. The load-balancer uses a session sticky round-robin approach to balance the load across all simulation workers. This means that new requests will be placed per round-robin on one of the workers. Follow-up requests, like requesting the status or exporting the results, are executed on the same worker node on which the job was started. Furthermore, as the results of the simulation are stored in a shared folder, results can be retrieved from every worker even if the worker that executed the simulation is already shutdown.

The bottleneck of a simulation run is usually the memory resource. Hence, each worker node is memory-aware and only starts a simulation job when enough memory is available. The parameters of a simulation job are part of the initial request and contain the maximum amount of required memory for the simulation. A new job is queued if the maximum required memory of the simulation job exceeds the available memory in the JVM of the worker. To obtain the status of a job the *Simulation Results Analyzer* queries the cluster using the simulations *jobID*. The job can either run, be queued, be finished, or has failed. After a job has been finished, the results of the simulation are available as an archive containing all simulation metrics and results.

The Performance Evaluation Tool (PET) is used to analyze the raw results of the simulation and calculates e.g., total resource utilization per resource and container, response times per operation, and total costs (Kroß et al., 2016). The aggregated results together with the topology are stored in the *Optimization Database*. The controller spawns new topologies based on the evaluation results and the configuration of the evolutionary algorithm (Lukasiewicz et al., 2011). We currently support three optimization goals:

- (i) Approximate the resource utilization of all resources in all used resource containers to a certain level.
- (ii) Minimize the total costs of the system.
- (iii) Minimize the total response time of the system.

The first optimization goal creates resource efficient topologies. It is important to limit the maximum CPU utilization to 70% or 80%, otherwise the resulting topologies are not representative as the response times become unpredictable due to the high system load. This optimization sorts, for example, memory intensive deployment units to CPU intensive deployment units.

The second optimization goal creates cheaper, more efficient topologies. The number of resource containers here is usually relatively low. Deployment units that interact frequently are deployed on the same server if network traffic is accounted for. The cost optimization goal has a loose relation with the first optimization goal.

For the last optimization goal, we calculate the mean response time over all simulated operations. A deployment topology with a smaller median response time is considered superior to a topology with a larger median response time. This optimization goal tends to require more resources and generates more costs when executed in production.

The applied framework for evolutionary computation supports multi-objective optimization. Hence, our approach supports combinations of the above stated optimization goals. This usually creates multiple optimal solutions along a Pareto-front (Koziolok/Koziolok/Reussner, 2011). The user must pick one of the solutions or let the optimizer select one solution randomly. Selecting one resulting topology implies accepting trade-offs. Fast topologies are usually more expensive than slower topologies. Depending on the application and its performance requirements, which are not part of the model, one solution might be superior to another. Thus, a manual selection is recommended.

Although the proposed approach attempts to handle deployment topology optimizations in a comprehensive way, it has two limitations. (i) Distributed Database Management Systems (DBMSs) have not been considered. We can today only size VMs for the DBs according to the workload instead of sharding or replicating the DBs. A more advanced DB performance model and dedicated monitoring solutions are required to resolve this limitation. (ii) The results of the architecture optimizer are never certain to be the best solution. Such algorithms can run into local optima and never find the best possible solution or might take extensive time to find the best solution. However, such a structured approach optimizes effectively based on all DA components (workload, resource profile, and available resource containers) (Koziolok/Koziolok/Reussner, 2011).

## 10.7 Evaluation

### 10.7.1 Evaluation System

The SPECjEnterpriseNEXT industry benchmark is the successor of the SPECjEnterprise2010 benchmark. Both are Java EE applications typically used to rate the performance of different Java EE ASs. We use a pre-release version<sup>8</sup> of the SPECjEnter-

---

<sup>8</sup>version from 19.02.2016

**Table 10.2:** *Software and hardware configuration for model generation*

Server	Load-balancer	Driver	Insurance/Vehicle/Provider Server	Insurance/Vehicle/Provider Database
Application Server	Apache 2.2.31	Faban 1.3.0	JBoss Wildfly 8.1.0 Final	-
Database	-	-	-	PostgreSQL 9.4.4
Java Virtual Machine	-	Oracle JDK 1.7.0.79		-
Operating System	CentOS 6.7	openSUSE Leap 42.1 (x86_64)		
CPU Cores	4 vCores (2.1 Gigahertz (GHz))			8 vCores (2.1 GHz)
Memory	8 GB	16 GB		8 GB
Host System	IBM System X3755M3			
Network	1 Gigabit-per-second (Gbit/s)			

priseNEXT as example DA for our evaluation. This benchmark mimics an insurance policy management system for car insurances. It consists of three different service components (*Insurance*, *Vehicle*, and *Insurance Service*) and three databases (*Insurance*, *Vehicle*, and *Provider Database*). Furthermore, we added a load-balancer handling requests to the AS instances to enable replicas of the ASs.

The benchmark contains a load driver emulating insurance customers. The so-called *Insurance Customer Driver* is based on Faban<sup>9</sup> and executes different business transactions on the *Insurance Domain*, which triggers JAX-RS<sup>10</sup> Representational State Transfer (REST) calls to the other two services and Java Persistence API (JPA) calls to the DBs.

## 10.7.2 Evaluation Approach

We conducted two evaluations to demonstrate the capabilities of our approach. One for on-premise installations and one for cloud environments. The second evaluation was conducted in the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) environment to demonstrate the applicability of our approach in industry cloud environments. AWS EC2 environment is the largest public IaaS provider<sup>11</sup>. Applying our research on this providers promises the best leverage to impact industry usage of our approach.

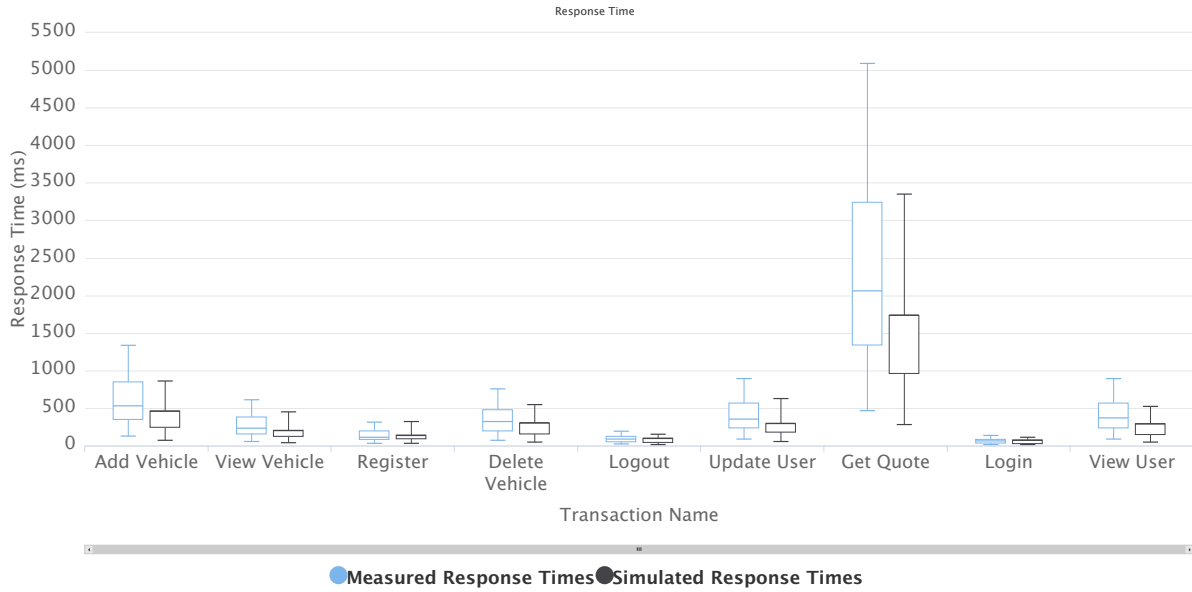
The evaluation process was similar for both environments. In a first step, we deployed a test system in our on-premise environment and executed load on this system to collect monitoring data. The complete configuration is described in Table 10.2. We conducted a run with 100 virtual users to collect APM data. We applied the RETIT Java EE solution as APM software for the ASs. This solution contains a Java Database Connectivity (JDBC) wrapper to collect response times from external DBs. For the load-balancer and database VMs we used the RETIT host monitoring, which collects CPU utilization time series. The load-balancer response times were extracted from Apache HTTP Server<sup>12</sup> access logs. With the response times and utilization we estimated the resource demands for the DB servers and the load-balancer using LibReDE (Willnecker/Krcmar, 2016).

<sup>9</sup><http://faban.org/>

<sup>10</sup><http://jax-rs-spec.java.net/>

<sup>11</sup><http://www.cloudcomputing-news.net/news/2016/aug/02/aws-microsoft-ibm-and-google-own-more-half-global-cloud-infrastructure-market/>

<sup>12</sup><http://httpd.apache.org/docs/2.2/en/logs.html>



**Figure 10.6:** *Response time evaluation - On-premise environment*

Two cost model instances were created. One for each environment considering their different characteristics. The cost-model for the on-premise environment uses only flat costs for running VMs, while the cloud cost-model considers uptime of different VM types, networking costs, and cost-free sections (e.g., first 10 GB traffic free, further traffic 0.16 ct/GB). We attached the cost models to the generated model. The complete model, an increased workload, and a list of available resource containers were used as input for our deployment topology optimizer.

Afterwards, the optimizer calculated topologies for both environments. The optimizer considered three optimization goals: (i) approximate the utilization of the resources at around 70%, (ii) minimize total costs of operations, and (iii) minimize the median response time over all business transactions. The result of the optimizer was a list of possible deployment topologies. Previous research showed that the optimization algorithm produces accurate results for single optimization goals (Willnecker/Krcmar, 2016). Therefore, we picked one of the best topologies close to the Pareto-front for each environment and deployed it.

Finally, we executed the increased workload on the newly deployed systems. The systems were monitored using system monitoring, which minimizes the influence of the monitoring on the system. Finally, we compared the system monitoring results of a test run in the real environment with the simulation results.

### 10.7.3 On-premise Evaluation

We used the topology described in Table 10.2 for the model generation run emulating 100 users. After initial generation, we conducted an optimization run discarding all CPU utilization above 70%, minimizing the response times and costs. Furthermore, we increased the workload from 100 to 500 users. The *Deployment Topology Allocator* created

**Table 10.3:** *Measurement and simulation results for a selected topology on-premise accessed by 500 users*

Resource	Metric	Balancer	Application Server (AS)					Database (DB)		
			AS 1	AS 2	AS 3	AS 4	AS 5	DB 1	DB 2	DB 3
Deployment	-	Balancer	Vehicle Insurance	Vehicle Insurance	Provider Vehicle	Provider Insurance	Insurance	Insurance	Vehicle	Provider
CPU	Measured util.	15.34%	71.23%	69.27%	68.19%	72.89%	72.64%	18.27%	34.47%	8.32%
	Simulated util	13.61%	65.37%	64.02%	62.87%	64.79%	63.65%	16.91%	30.85%	7.82%
	Relative err.	11.28%	8.23%	7.58%	7.80%	11.11%	12.38%	7.44%	10.50%	6.01%
Memory	Measured dem.	1 GB	5.95 GB	6.13 GB	4.25 GB	3.76 GB	6.79 GB	8 GB	8 GB	8 GB
	Simulated dem.	-	6.59 GB	6.73 GB	4.86 GB	4.04 GB	7.50 GB	-	-	-
	Relative err.	-	10.76%	9.79%	14.35%	7.45%	10.46%	-	-	-
HDD	Measured dem.	0.24%	1.29%	0.54%	1.09%	1.51%	2.99%	8.21%	10.92%	4.39%
	Simulated dem.	-	0.92%	0.41%	0.69%	1.10%	2.67%	-	-	-
	Relative err.	-	28.68%	21.15%	36.70%	27.15%	10.70%	-	-	-

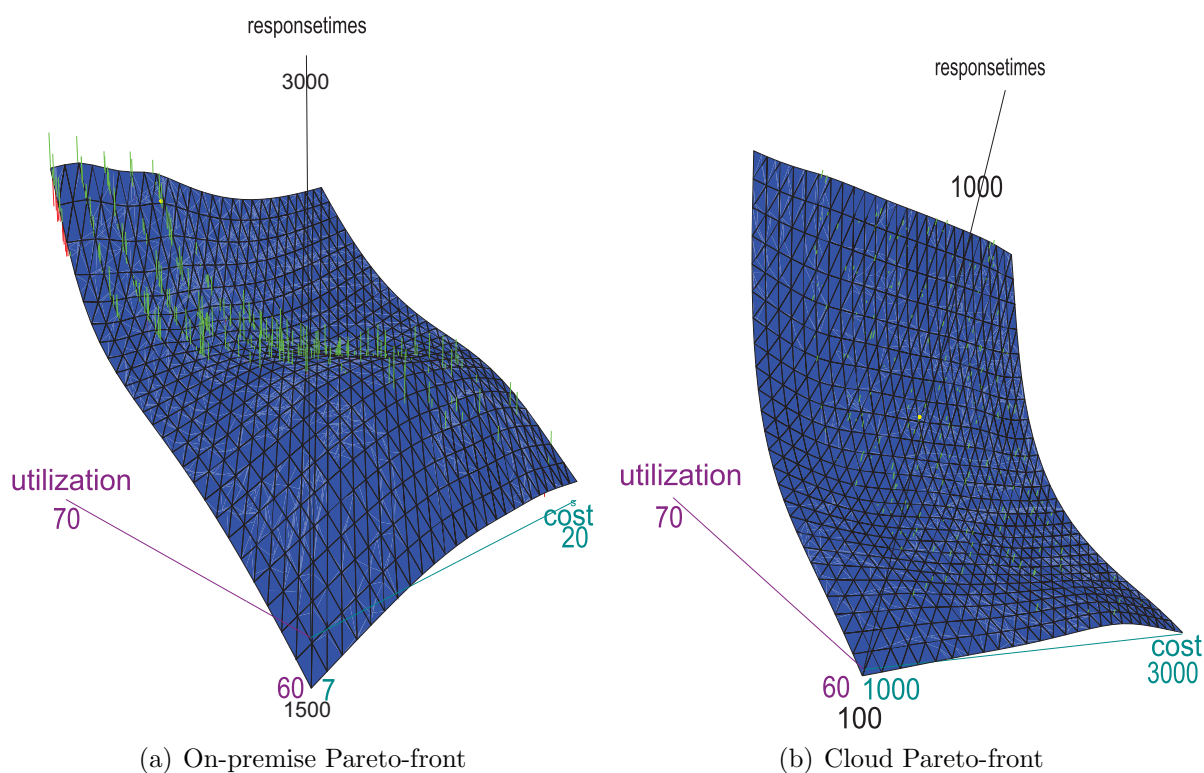
an initial population of 50 different topologies for the evolutionary algorithm. We selected a cross-over rate of 0.95, an offspring population size  $\lambda$  of 25 and a parent population size  $\mu$  of 25. The algorithm calculated 50 generations resulting in a total of 1250 tested topologies. We used 16 worker nodes in the simulation cluster resulting in about 5 hours of computation.

The resulting topologies and their evaluation criteria are depicted in Figure 10.7(a). As the costs did not make a huge difference in this scenario we selected a topology with high utilization (meaning less VMs) and relatively good response times. Especially response times can vary from an average response time of 1500 to 2500 ms as depicted on the y-axis in Figure 10.7(a).

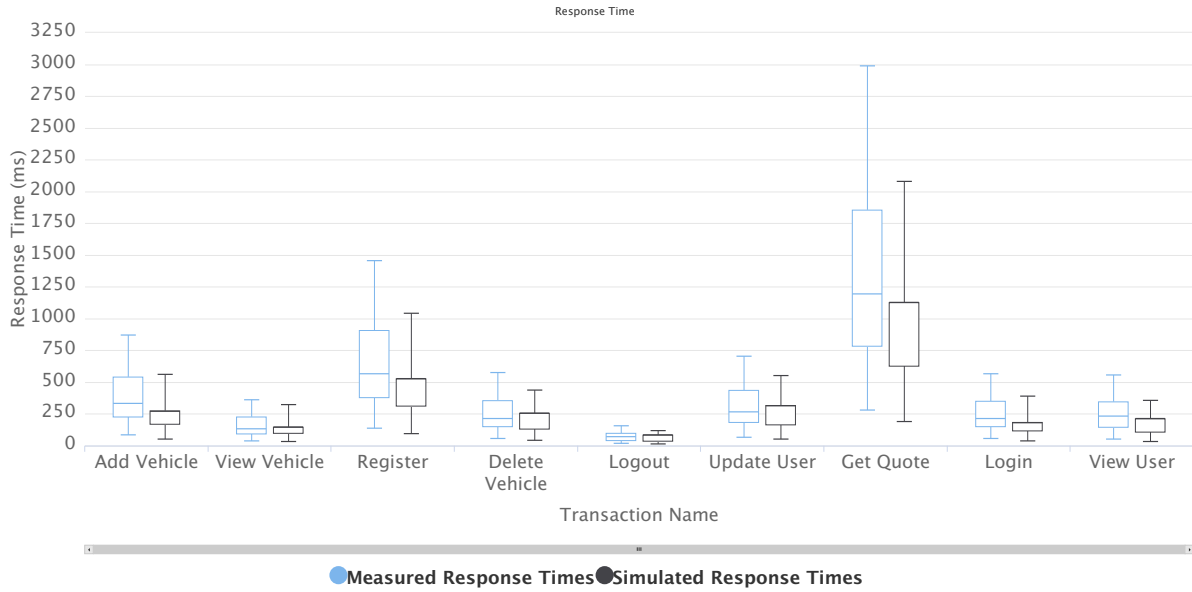
The selected topology required 5 ASs, 3 DB servers and 1 load-balancer. We deployed the system as specified and conducted a run with only system monitoring as instrumentation. Afterwards, we compared the response times and resource utilization from the system monitoring with the simulation results for this topology. The median response time error was about 15% for most of the transactions as depicted in Figure 10.6. Only the *Register* and *View User* business transactions had a larger error of above 20%, still below the 30% acceptable error for capacity planning propagated by Menascé and Almeida (2008) (Menascé, 2008).

Table 10.3 shows the comparison of simulated and measured resource utilization. The HDD and memory simulation has only been conducted for the ASs due to the fact that LibReDE calculates demands only for the CPU resource and no dedicated monitoring solution was available for these technologies. However, we added our measurements for all resources on all servers, even though memory measurements on the DB servers are only available on process-level. The accuracy is high for CPU demands, especially when LibReDE is used. This confirms previous research comparing LibReDE and resource demand monitoring solutions (Willnecker/Krcmar, 2016). We tend to under-predict CPU demands but over-predict memory demands for the ASs. The under-prediction is a result of overhead tasks of the servers that are not part of the model (e.g., database pool management, load-balancer health checks, etc.) (Willnecker/Krcmar, 2016). The over-prediction of memory demands is a result of the delay between a threshold detection and the GC, the memory grow, or the memory shrink execution in the simulation.





**Figure 10.7:** *Pareto-fronts along optimal deployments in both environments. X-Axis: costs, Y-Axis: average response times, Z-Axis: average resource utilization. Dots represent actual measurements, blue plane represents calculated Pareto-front)*



**Figure 10.8:** *Response time evaluation - Cloud environment*

### 10.7.4 Cloud Environment Evaluation

We could reuse large parts of the model definition from the on-premise setting. We had to re-calibrate CPU frequencies in the resource environment model as well as network bandwidth and latency. Therefore, we conducted CPU calibrations using the SPEC CPU 2006<sup>13</sup> benchmark on our local VMs as well as on the different EC2 instances. This allowed us to calculate the relative CPU processing rate between the VMs used for the model generation and the EC2 instances that are simulated. Furthermore, we conducted bandwidth and latency calibrations using `lmbench`<sup>14</sup>. This sort of calibration allows us to transform the model from one hardware environment to another and provides accurate results even for virtualized environments (Brunnert/Krcmar, 2017). The load driver was still installed in our on-premise environment as a usual customer would access the system from outside of the AWS EC2 environment.

Using the AWS EC2 environment increased the number of possible resource containers (complete M4 instance types<sup>15</sup> except for `m4.10xlarge`) and regions (e.g., Europe - Ireland, US-East). Therefore, we increase the number of generations calculated by our optimizer to 100 (from 50 in the previous scenario). This resulted in 2500 tested topologies and took about 10 hours to compute using our 16 simulation worker nodes.

The results of the optimization run are depicted Figure 10.7(b). As shown, similar response times can be achieved with costs ranging from less than \$1500 per month and up to \$3300. We selected a topology with very good response times and upper mid-range costs (about \$2800 per month). The response times are depicted in Figure 10.8. The simulation results are even more accurate compared to the on-premise installation. The relative error of the media response time is below 20% for all cases. The *Register* and *Get*

<sup>13</sup><http://www.spec.org/cpu2006/>

<sup>14</sup><http://lmbench.sourceforge.net/>

<sup>15</sup><http://aws.amazon.com/de/ec2/instance-types/>

**Table 10.4:** *Measurement and simulation results for a selected cloud topology accessed by 500 users users*

Resource	Metric	Balancer	Application Server (AS)				Database (DB)		
			AS 1	AS 2	AS 3	AS 4	DB 1	DB 2	DB 3
-	Deployment	Balancer	Vehicle Provider Insurance	Vehicle Provider	Provider Insurance	Vehicle	Provider	Vehicle	Insurance
CPU	<b>Measured util.</b>	8.78%	58.73%	45.47%	53.99%	55.38%	12.17%	28.88%	6.34%
	<b>Simulated util.</b>	8.09%	50.91%	39.01%	47.60%	50.07%	12.05%	27.41%	5.93%
	<b>Relative err.</b>	7.86%	13.32%	14.21%	11.84%	9.59%	0.99%	5.09%	6.47%
Memory	<b>Measured dem.</b>	1 GB	7.81 GB	8.56 GB	8.09 GB	7.95 GB	8 GB	8 GB	8 GB
	<b>Simulated dem.</b>	-	8.13 GB	8.94 GB	8.48 GB	8.46 GB	-	-	-
	<b>Relative err.</b>	-	4.10%	4.44%	4.82%	6.42%	-	-	-
HDD	<b>Measured dem.</b>	0.12%	1.79%	1.32%	1.53%	2.89%	6.92%	9.18%	8.02%
	<b>Simulated dem.</b>	-	1.37%	1.07%	1.27%	2.31%	-	-	-
	<b>Relative err.</b>	-	23.36%	18.94%	16.99%	20.07%	-	-	-
Costs	<b>Measured dem.</b>	116.64 €	193.25 €	773.00 €	773.00 €	386.50 €	96.63 €	386.50 €	96.63 €
	<b>Simulated dem.</b>	113.97 €	190.09 €	764.23 €	764.23 €	380.38 €	95.23 €	380.38 €	95.23 €
	<b>Relative err.</b>	2.29%	1.64%	1.13%	1.13%	1.58%	1.45%	1.58%	1.45%

*Quote* business transactions have an error of only 5%. As in the on-premise evaluation, we tend to under-predict the actual response times. Our model lacks certain overhead tasks of the application server and OS. Computation time needed by these tasks are not considered in our model. Therefore, CPU utilization and response times are slightly under-predicted.

The resource utilization results are depicted in Table 10.4. The results are comparable to the previous evaluation. To evaluate the cost estimation, we approximated the costs for a month and compared this with the AWS cost calculator<sup>16</sup>, as the calculator also provides costs per month. This included costs for the containers and the outbound traffic. Inbound traffic between the EC2 instances was without cost. The cost estimation error was below 2%. Both the cost and the performance model produce accurate results allowing to predict and thus optimize topologies in cloud environments. Using the network benchmarking even allows prediction of the performance of hybrid environments with components deployed in multiple cloud environments or parts deployed in the cloud and other parts on-premise.

To demonstrate the advantages of our approach, we tested two logical topologies. These topologies were derived by scaling up small instances of AWS EC2 instance and large instances for the second topology. We scaled up deployed 3 m4.large instances until the CPU utilization was below 90% on each instance. This led to 12 application server instances and 3 DB server instances. The costs were 41.90% higher compared to our selected deployment and the median response times increased by about 135ms. The CPU utilization was at about 68% only one of the DB servers was at about 80% already. To select the second topology we increased the instance type until the CPU utilization was below 90%. This topology used 3 m4.xlarge machines for the application servers and the same m4.large servers for the DBs. The response times were similar to the response times from the optimized topology but the costs increased by 152.11%. This is due to the relatively small CPU utilization of about 45% on the application servers side. These two topologies use simple strategies but such default topologies are easy to derive as more

<sup>16</sup><http://calculator.s3.amazonaws.com/index.html>

complex strategies lack of tools to support as the here presented deployment topology optimizer.

## 10.8 Conclusion

This work successfully connected PMGs and architecture optimizers for optimizing deployment topologies of DAs. This approach allows DevOps engineers to detect the current software architecture and demands of a DA and to size and optimize for on-premise, cloud or hybrid environments. This holds true in up-scaling scenarios as presented in Section 10.7. This leads to reduced response times, increased resource utilization, and/or reduced costs.

In addition, this work presented an extension for the PCM meta-model for memory resources. The extension supports dynamic and automatic memory management and introduces a probabilistic GC simulation. Monitoring and detecting GC events as well as generating performance models with an accurate memory model have been integrated into the RETIT PMG and the PMWT. The accuracy and feasibility of this approach have been demonstrated.

We presented a flexible cost model as an extension for the PCM meta-model. This cost model can be extended with other billable items and supports tiered pricing models. We demonstrated the accuracy for the AWS EC2 environment with a prediction error below 2%.

In contrast to previously introduced architecture optimizers, our approach uses simulations instead of solvers. The high accuracy of the simulations presented in Section 10.7 justify this decision. However, this accuracy comes with computational costs. To limit the effect on the decision time we introduced parallel execution in a simulation cluster. We demonstrated the scalability of this cluster with up to 16 worker nodes. Furthermore, optimizations of the simulation process as well as using analytical solvers for a coarse-grained topology estimation before simulating the best candidates would allow to speed up the computation. This would allow to search for more topologies or evaluate the same amount of topologies in shorter period.

Compared to our previous research, we improved the calculation speed by switching from the SimuCom to the EventSim simulation engine (Willnecker/Krcmar, 2016). This increased the amount of simulations that can be conducted by a single worker. Pre-calculating several topologies and selecting the one that best suits the current workload could reduce the disadvantage of computational intensity of our approach. This would allow to use our approach also for runtime decisions.

A continuous monitoring of workload and resource demands in combination with the above mentioned optimizations allow using this approach for runtime decisions. Smart application-aware resource provisioning can increase average resource utilization in data centers, reduce the cost of operations and reduce the carbon footprint due to more efficient

---

typologies. Furthermore, geographic characteristics could be considered during runtime. Latency and thus response times of the application can be reduced by considering the current workload per region and assigning VMs geographically close to the majority of the users. Global workforces can benefit from smart and proactive resource provisioning which takes working hours in different regions of the world into consideration.

# Part C

# Chapter 11

## Discussion

In this chapter the results of the publications are further discussed. First, the results are summarized and then limitations, contributions and future research are described.

### 11.1 Summary

This section provides an overview of the findings for each embedded publication. The key results are ordered per paper and listed in Table 11.1.

**P1** proposed an integration of an industry Application Performance Management (APM) solution with an established performance model generation framework. The integration was realized using an abstraction layer that allows to integrate different input formats and levels of granularity. The application of this extension to industry standard solutions demonstrates that the generator and its interface are generally applicable and other APM or monitoring solutions. As the Dynatrace solution is in widespread use, the monitoring technology is very mature and allows to use performance model generation in varied operation environments. The generated models can be used to simulate different scenarios, such as new workloads or other resource environments. This enhances the Dynatrace solution with capacity planning capabilities.

**P2** compared three different techniques for collecting resource demands for performance model generation. In this work, we compared a monitoring approach from academia, the industry monitoring solution Dynatrace, and Library for Resource Demand Estimation (LibReDE), a library combining six different estimation approaches. We integrated this techniques into an automatic performance model generator (PMG). We evaluated all techniques in a standalone and in a distributed setup, Furthermore, we tested the techniques in a virtualized and a bare-metal environment for two levels of granularity: system entry point level and component operation level. The results in this paper show that all three techniques deliver good results for both granularity levels and in all environments. Hence, all of this techniques can be used to derive resource demands for performance

model generation. However, some techniques work better in certain scenarios. The estimation techniques delivers better results at system entry point level, but falls short behind direct measurements for the component operation level. Furthermore, measurements can extract resource demands on any level of detail. Estimation techniques require to calculate demands for the complete monitored system to distribute the measured utilization amongst its components. Estimation techniques can be applied to a broad variety of technologies as the requirements for data collection are easily fulfilled with standard system tools. Combining the approaches provides the best results. We demonstrated this by using a hybrid setup, where measurement approaches are used to extract resource demands for the User Interface (UI) and Web Service (WS) combined with estimations for the database (DB).

With **P3**, we demonstrated a full stack performance model considering all four major resource types. We demonstrate the generated model to be accurate for all resource types. Even though the resource utilization simulation is accurate, the response time error was quite high. Overhead tasks that were not part of the model, as well as a simple Garbage Collection (GC) approach leave room for improvement. The probabilistic GC model delivers promising results, however the average number of bytes could be eliminated as an input parameter for the GC operations. This requires, that the actual number of bytes ready to be freed is stored during simulation.

**P4** significantly improved the GC model and corresponding simulation. It demonstrated an accurate approach for predicting memory and garbage collection behavior. This work presented a meta-model extension for the architecture-level performance model Palladio Component Model (PCM), a monitoring approach for extracting memory including GC heuristics, and a model generation approach creating memory-aware models for Enterprise Applications (EAs). The results were validated in a series of controlled experiments using an on-premise and an industry cloud environment. The results approximate the actual memory behavior with high accuracy and show that this probabilistic approach is suitable for memory and GC behavior prediction. Predicting the impact of software changes, workload changes, or changes to the runtime do not require extensive performance tests. A short load test for the model generation results in an accurate memory model. This model compared to previous model versions can predict the impact on an applications memory profile or it can be used to predict the applications memory behavior in other environments. This allows accurate capacity planning for large-scale EA.

**P5** successfully connected PMGs and architecture optimizer for optimizing deployment topologies of EAs. This approach allows software architects to detect the current software architecture and resource demands of an EA and to automatically size and optimize for the target environments. This was demonstrated in an up-scaling scenario. Using this approach allows to reduce response times, increase resource utilization and, reduce costs depending on the optimization goal. Furthermore, this approach can be used to test optimization approaches against real world applications using the contained PMG. In contrast to previously introduced architecture optimizers, the presented approach uses simulations instead of solvers and covers scenarios with many users. The high accuracy of the simulations presented in this work justify this decision. However, this accuracy comes with higher computational costs. To reduce the effect on the optimization time we introduced parallel simulation in a dedicated model simulation cluster.



This cluster was further developed and presented in **P6**. This work showed a scalable simulation service called SaaS as part of the Performance Management Work (PMW) tool chain. SaaS can conduct PCM simulations as a distributed service that is resource-aware and automatically scales to necessary size when run in a *Kubernetes* instance. SaaS is offers a simple REST interface allowing developers to easily integrate SaaS into their tool chain. We demonstrated this with by integrating SaaS into two PMW tools: Deployment Unit Optimizer (DUO) and Load Test Selector (LTS). SaaS is able to support multiple versions of PCM using a Command Line Simulators (CLSs). This feature allows to run SaaS as a Software as a Service (SaaS) application for multiple institutes or to run tests of multiple PCM instances. The general architecture allows developers to integrate other performance simulations or solving techniques for performance models.

**P7** combines the above mentioned approaches into a memory-aware multi-objective deployment topology optimizer. This allows DevOps engineers to detect the current software architecture and demands of a distributed application (DA) and to size and optimize for on-premise, cloud, or hybrid environments. Furthermore, this work presented a flexible cost model as an extension for the PCM meta-model. This cost model can be extended with other billable items and supports tiered pricing models. We demonstrated the accuracy for the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) environment with a prediction error below 2%. In addition, this work demonstrated the scalability of the simulation cluster with up to 16 worker nodes. Compared to previous research, we improved the calculation speed by switching from the SimuCom to the EventSim simulation engine. This increased the amount of simulations that can be conducted by a single worker.

## 11.2 Limitations

The presented methods, models, and techniques have certain limitations, which we present in this section. The performance model generation approach, requires a transaction ID that is unique through the complete system. Such IDs are usually injected into the working thread of a user request and into the header information when network transfers, such as REST calls, occur. However, this works fine unless the technology stack does not allow that. This is the case for database management systems. We can still track database calls from an application server, but our monitoring an model generation is blind if any further cascades occur, like a secondary database system, a replication, or a shard. The resulting model would be inaccurate and with a higher error level.

Another limitation regarding the performance model generation is the amount of data and the expected input parameters. The amount of data handled by the EA should be representative, compared to the actual environment, when the model generation is conducted. Otherwise, higher or lower amounts of data might produce other resource demands and harm the accuracy of the resulting model. Same goes for input parameters. The resulting model can be inaccurate, if the the input parameters during the model generation and their distribution does not match the real environment. Assuming, that we want to generate a model for an address book application. This limitation also harms

No.	Key Results
P1	<ul style="list-style-type: none"> <li>• Monitoring-abstract layer for a PMG</li> <li>• Application of this abstraction layer to industry APM solutions</li> <li>• Model generation and comprehensive evaluation using the industry-standard benchmark SPECjEnterprise2010 with an integrated database</li> </ul>
P2	<ul style="list-style-type: none"> <li>• Comparison of resource demand measurements and resource demand estimation</li> <li>• Integration of LibReDE into a PMG</li> <li>• Hybrid of measurements and demand techniques</li> <li>• Model generation and comprehensive evaluation using the industry-standard benchmark SPECjEnterprise2010 with an external database</li> </ul>
P3	<ul style="list-style-type: none"> <li>• Initial probabilistic memory and garbage collection model</li> <li>• Integration of Central Processing Unit (CPU), Hard Disk Drive (HDD), network and memory into a single PMG</li> <li>• Full-stack evaluation of a performance model using all four resource types</li> <li>• Model generation and comprehensive evaluation using the industry-standard benchmark SPECjEnterpriseNEXT with internal databases</li> </ul>
P4	<ul style="list-style-type: none"> <li>• Mature memory meta-model for dynamic and automatic memory management simulation</li> <li>• Automatic extraction of memory demands and garbage collection heuristics</li> <li>• Demonstration of applicability in different environments, using different garbage collector algorithms, and different workloads</li> <li>• Model generation and comprehensive evaluation using the industry-standard benchmark SPECjEnterprise2010 and SPECjEnterpriseNEXT with internal databases</li> </ul>
P5	<ul style="list-style-type: none"> <li>• Single-objective deployment topology optimizer</li> <li>• Combination of architecture optimization and performance model generation</li> <li>• Model generation and comprehensive evaluation using the industry-standard benchmark SPECjEnterpriseNEXT using external databases</li> </ul>
P6	<ul style="list-style-type: none"> <li>• Scalable simulation service for PCM</li> <li>• Extension interface for other simulation engines and meta-models</li> <li>• Containerization of simulation service</li> </ul>
P7	<ul style="list-style-type: none"> <li>• Multi-objective deployment topology optimizer</li> <li>• Flexible cost-model integrated into PCM for cloud environments</li> <li>• Optimization of on-premise, cloud, and hybrid environments</li> <li>• Model generation and comprehensive evaluation using the industry-standard benchmark SPECjEnterpriseNEXT using external databases in multiple environments</li> </ul>

**Table 11.1:** *Key results of embedded publications*

load test approaches that use inaccurate or wrong data inputs to estimate the performance of a real system.

Even though, we use a combination of recombination and mutation for our deployment optimizer, it is still possible that not the best solution is found. The optimal solution might never be found as the search space can be very large. We try to conduct a broad search, by using recombinations in every generation. This might not lead to the best solution, but usually to a deployment topology that makes use of underutilized resources in a naive deployment.

The evaluation has been conducted using several technologies, amongst these are multiple databases (Derby and PostgreSQL), multiple application servers (Wildfly and Glassfish), and multiple EAs (SPECjEnterprise2010 and SPECjEnterpriseNEXT). However, the technological stack mostly relies on the Java Virtual Machine (JVM). The integration of industry APM and estimation techniques like LibReDE demonstrate that other stacks can work well with our PMG. Nevertheless, this has not yet been demonstrated yet. Therefore, other technologies might produce different results.

## 11.3 Contribution to Research

As a main contribution to research, this work combined two separated research fields: architecture optimization and performance model generation/extraction. This combinations allows to use real world applications and connect those to academic research regarding architecture optimization. It can be expected, that future research in this area is evaluated with real applications, as the barrier for generating such models is lower today.

Furthermore, academic research benefits from the meta-model approaches presented in this work. Especially, the probabilistic memory model allows to research in memory intensive areas such as in-memory databases. The performance model research so far disregarded memory, especially automatic memory management. With the contributions of this work, automatic and dynamic memory management is integrated in architecture-level performance models. The presented methods for generating a performance model including memory demands and garbage collection increases the quality of these models and their accuracy regarding real world performance.

Larger simulation-based experiments with a huge amount of variants become possible with the simulation service SaaS. This tool provides researches using PCM the opportunity to run multiple parallel simulations on a distributed simulation cluster. The integration into other tool-chains using a REST Application Programming Interface (API) contributes to automation in performance research. Other simulation engines are easily integrated, so that research in other fields as performance possibly benefits from this service.

## 11.4 Contribution to Practice

This work contributes to practice by providing a holistic tool that can derive performance models and optimize their deployment topology. Especially for cloud environments, this allows companies to save operation costs by providing similar quality of service (Rousel/Branson, 2017). Unused resources are utilized by the EA provider instead of the Infrastructure as a Service (IaaS) provider. Furthermore, this contributes to energy savings by optimizing the resource utilization of data centers and thus reducing the amount of necessary hosts.

The memory management model provides an easy to understand heuristic for profiling memory behavior of an application. Developers and architects can use these as metrics to understand, monitor, and improve the memory footprint of their applications. Instead of analyzing large amount of memory monitoring, the key heuristics of this model allow to analyze and simulate the memory behavior of an application.

The improvements of the PMGs presented in this work lead to more accurate performance models. These models cover more aspects of the runtime environment and of the application itself. This comprises memory and garbage collection behavior and the use of resource demand measurements and estimations. The latter increases the supported technologies for model generation the number of potential monitoring solutions is higher when combining both approaches. Tools for capacity planning and in continuous delivery pipelines that use PMG benefit from these improvements Brunnert/Kremer (2017); Düllmann et al. (2017).

## 11.5 Future Research

With the foundations of this research, analysis of industry EA regarding their performance was eased. The combination of of industry standard APM solutions with academic PMGs allows software engineers, architects, and operators to use latest performance model research in practical scenarios. However, applying this tools to other technology stacks, other than JVM based technologies, remains an open challenge. Differences in the runtime, typical application architectures, and design patterns might impact the generation approach. This can affect the resource demands, the workflow, and the memory/garbage collection behavior. Tweaks or changes might be necessary to generalize the approaches presented in this work.

Regarding our simulation cluster, future work mainly concerns integrating SiaaS into other tools and increase the stability of the service and its infrastructure components. Furthermore, the resource-awareness of SiaaS can be extended to consider CPU utilization instead of only memory consumption for scheduling jobs and spawning new simulation instances.

The memory-model and simulation techniques enables researchers to analyze several new technologies. Especially in-memory technologies, like in-memory databases, rely heavily on an accurate memory-model. Such a model is provided with this work and integrated into architecture-level performance models. This allows to generate, simulate, and predict the behavior of such memory-depended applications and runtimes. Future research in this area is eased by using the applied model and simulation technologies.

The deployment optimizer takes several hours of computation today. Future extensions of this optimizer can speed up the calculation by improving the simulator or by introducing a two-phase optimization. Using analytical solvers for a coarse-grained topology estimation before simulating the best candidates would allow to speed up the computation. This would allow to search for more topologies or evaluate the same amount of topologies in shorter period. A faster approach (e.g., solver, simplified model) could be used to calculate a good initial population instead of random topologies. This population would be the input for the optimization approach presented in this work. This concept could lead to faster or better results. Reducing the computation time by selecting smarter initial populations remains an open challenge that can be addressed with the results of this thesis.

Faster calculations would allow to use our concept for real-time decisions. Optimized topologies could be calculated on-the-fly, when continuously monitoring and analyzing the workload of an EA. Current runtime models tend to allocate new resources when spikes in the workload occur. A workload and software architecture aware can produce more resource efficient results and therefore reduce overhead and costs for distributed applications. Especially in cloud environments, cost savings by smart runtime decisions, can significantly reduce the operation costs (Roussel/Branson, 2017). The combination of our approach with runtime decision or autonomous computing frameworks is a very interesting topic for further research.

# References

- Alavi, M. (1984):** An Assessment of the Prototyping Approach to Information Systems Development. *Communications of the ACM*, vol. 27 no. 6, 556–563 (URL: <http://doi.acm.org/10.1145/358080.358095>) last accessed 2010-06-30, ISSN 0001–0782.
- Aleti, A.; Buhnova, B.; Grunske, L.; Koziolk, A.; Meedeniya, I. (2013):** Software Architecture Optimization Methods: A Systematic Literature Review. *Software Engineering, IEEE Transactions on*, vol. 39 no. 5, 658–683.
- Angel, D. J.; Kumorek, J. R.; Morshed, F.; Seidel, D. A. (2001):** Byte Code Instrumentation. Web <https://patents.google.com/patent/US6314558B1/en>, US Patent 6,314,558.
- Appel, A. W. (1989):** Simple Generational Garbage Collection and Fast Allocation. *Software: Practice and Experience*, vol. 19 no. 2, 171–183.
- Ardagna, D.; Gibilisco, G.; Ciavotta, M.; Lavrentev, A. (2014):** A Multi-model Optimization Framework for the Model Driven Design of Cloud Applications. In Search-Based Software Engineering. vol. 8636, Springer International Publishing, Switzerland (URL: [http://dx.doi.org/10.1007/978-3-319-09940-8\\_5](http://dx.doi.org/10.1007/978-3-319-09940-8_5)) last accessed 2010-06-30, ISBN 978–3–319–09939–2, 61–76.
- Balalaie, A.; Heydarnoori, A.; Jamshidi, P. (2016):** Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, vol. 33 no. 3, 42–52.
- Balsamo, Simonetta, M. A. (2007):** Queueing Networks. Venezia, Italy (URL: <http://www.sti.uniurb.it/events/sfm07pe/slides/Balsamo.pdf>) last accessed 2010-06-30.
- Becker, S.; Koziolk, H.; Reussner, R. (2009):** The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, vol. 82 no. 1, 3–22, Special Issue: Software Performance - Modeling and Analysis, ISSN 0164–1212.
- Blaschek, G.; Lengauer, P. (2015):** Time Matters: Minimizing Garbage Collection Overhead with Minimal Effort. In Proceedings of the 2015 Symposium on Software Performance (SSP 2015)..
- Boehm, H.-J.; Weiser, M. (1988):** Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience*, vol. 18 no. 9, 807–820.

- Brosig, F.; Gorsler, F.; Huber, N.; Kounev, S. (2013):** Evaluating Approaches for Performance Prediction in Virtualized Environments. In Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on. , ISSN 1526–7539, 404–408.
- Brosig, F.; Huber, N.; Kounev, S. (2014):** Architecture-Level Software Performance Abstractions for Online Performance Prediction. *Science of Computer Programming*, vol. 90, 71–92, ISSN 0167–6423.
- Brosig, F.; Kounev, S.; Krogmann, K. (2009):** Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In Proceedings of the 1st International Workshop on Run-time Models for Self-managing Systems and Applications (ROSSA 2009). ACM, New York, NY, USA.
- Brunnert, A.; Hoorn, A. van; Willnecker, F.; Danciu, A.; Hasselbring, W.; Heger, C.; Herbst, N.; Jamshidi, P.; Jung, R.; Kistowski, J. von; Koziol, A.; Kroß, J.; Spinner, S.; Vögele, C.; Walter, J.; Wert, A. (2015):** Performance-oriented DevOps: A Research Agenda. SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC) (SPEC-RG-2015-01)– technical report ⟨URL: [http://research.spec.org/fileadmin/user\\_upload/documents/wg\\_devops/endorsed\\_publications/SPEC-RG-2015-001-DevOpsPerformanceResearchAgenda.pdf](http://research.spec.org/fileadmin/user_upload/documents/wg_devops/endorsed_publications/SPEC-RG-2015-001-DevOpsPerformanceResearchAgenda.pdf)⟩ last accessed 2010-06-30.
- Brunnert, A.; Krcmar, H. (2017):** Continuous performance evaluation and capacity planning using resource profiles for enterprise applications. *Journal of Systems and Software*, vol. 123, 239 – 262 ⟨URL: <http://www.sciencedirect.com/science/article/pii/S0164121215001831>⟩ last accessed 2010-06-30, ISSN 0164–1212.
- Brunnert, A.; Neubig, S.; Krcmar, H. (2014):** Evaluating the Prediction Accuracy of Generated Performance Models in Up- and Downscaling Scenarios. In Symposium on Software Performance (SOSP) 2014., 113–130.
- Brunnert, A.; Vögele, C.; Danciu, A.; Pfaff, M.; Mayer, M.; Krcmar, H. (2014):** Performance Management Work. *Business & Information Systems Engineering*, vol. 6 no. 3, 177–179.
- Brunnert, A.; Vögele, C.; Krcmar, H. (2013):** Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications. In Computer Performance Engineering. vol. 8168, Springer, Berlin Heidelberg, ISBN 978–3–642–40724–6, 74–88.
- Brunnert, A.; Wischer, K.; Krcmar, H. (2014):** 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. ACM, Lille, France, QoSA '14 ⟨URL: <http://doi.acm.org/10.1145/2602576.2602587>⟩ last accessed 2010-06-30, ISBN 978–1–4503–2576–9, 53–62.
- Chen, F.; Grundy, J.; Schneider, J.-G.; Yang, Y.; He, Q. (2015):** StressCloud: A Tool for Analysing Performance and Energy Consumption of Cloud Applications. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 2., ISSN 0270–5257, 721–724.

- Coello, C. A. C.; Lamont, G. B.; Van Veldhuisen, D. A. (2007):** Evolutionary Algorithms for Solving Multi-objective Problems. Springer, New York, USA.
- Cooperation, O. (2016):** Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide. USA, Blog Article (URL: <https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html#sthref28>) last accessed 2010-06-30.
- Cortellessa, V.; Di Marco, A.; Inverardi, P. (2011):** Model-based software performance analysis. Springer Science & Business Media, Berlin, Germany.
- Danciu, A.; Kroß, J.; Brunnert, A.; Willnecker, F.; Vögele, C.; Kapadia, A.; Krcmar, H. (2015):** Landscaping Performance Research at the ICPE and Its Predecessors: A Systematic Literature Review. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. ACM, New York, NY, USA, ICPE 2015, ISBN 978-1-4503-3248-4, 91-96.
- Denaro, G.; Polini, A.; Emmerich, W. (2004):** Early Performance Testing of Distributed Software Applications. In ACM SIGSOFT Software Engineering Notes. vol. 29, ACM, 94-103.
- Denning, P. J. (1997):** A New Social Contract for Research. *Communications of the ACM*, vol. 40 no. 2, 132-134 (URL: <http://doi.acm.org/10.1145/253671.253755>) last accessed 2010-06-30, ISSN 0001-0782.
- Dijkstra, E. W.; Lamport, L.; Martin, A. J.; Scholten, C. S.; Steffens, E. F. (1978):** On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, vol. 21 no. 11, 966-975.
- Dlugi, M.; Brunnert, A.; Krcmar, H. (2015):** Model-based Performance Evaluations in Continuous Delivery Pipelines. In Proc. Qudos '15. ACM, 25-26.
- Düllmann, T.; Heinrich, R.; Hoorn, A. van; Pitakrat, T.; Walter, J.; Willnecker, F. (2017):** CASPA: A Platform for Comparability of Architecture-based Software Performance Engineering Approaches. In Proceedings of the IEEE International Conference on Software Architecture (ICSA 2017). IEEE, Gothenburg, Sweden, ICSA 2017.
- Dynatrace (2015):** Dynatrace Agent Timers. USA, <https://community.compuwareapm.com/community/display/DOCDT60/Agent+Timers>, Accessed: 14.05.2015.
- Ehrgott, M. (2006):** Multicriteria Optimization. Springer Science & Business Media, Berlin - Heidelberg, Germany.
- Fielding, R. T. (2000):** Architectural styles and the design of network-based software architectures. Dissertation, University of California, Irvine.
- Forouzan, B. A. (2013):** Foundations of Computer Science. 3. edition. Cengage Learning Emea, Cheriton House, North Way, Walworth Industrial Estate, Andover SP10 5BE, UK.



- Fowler, M.; Lewis, J. (2014):** Microservices - A Definition of this New Architectural Term. USA, Blog Article (URL: <http://martinfowler.com/articles/microservices.html>) last accessed 2010-06-30.
- Franks, G.; Hubbard, A.; Majumdar, S.; Neilson, J.; Petriu, D.; Rolia, J.; Woodside, M. (1995):** A Toolset for Performance Engineering and Software Design of Client-Server Systems. *Performance Evaluation*, vol. 24 no. 1-2, 117–136.
- Greifeneder, B. (2011):** Method and System for Processing Application Performance Data Outside of Monitored Applications to Limit Overhead Caused by Monitoring. USA, US Patent 7,957,934.
- Grinshpan, L.; Hanso, L. ed. (2012):** Solving Enterprise Applications Performance Puzzles: Queuing Models to the Rescue. 1. edition. John Wiley & Sons, USA, ISBN 1118061578, 9781118061572.
- Guo, S.; Bai, F.; Hu, X. (2011):** Simulation Software as a Service and Service-Oriented Simulation Experiment. In Proceedings of the 2011 IEEE International Conference on Information Reuse and Integration (IRI)., 113–116.
- Haight, C.; De Silva, F. (2016):** Magic Quadrant for Application Performance Monitoring Suites. Gartner – technical report (URL: <https://www.gartner.com/doc/3551918/magic-quadrant-application-performance-monitoring>) last accessed 2010-06-30.
- Harchol-Balter, M. (2013):** Performance Modeling and Design of Computer Systems. Cambridge University Press, New York, NY, USA.
- He, S.; Guo, L.; Guo, Y.; Wu, C.; Ghanem, M.; Han, R. (2012):** Elastic Application Container: A Lightweight Approach for Cloud Resource Provisioning. In 2012 IEEE 26th International Conference on Advanced Information Networking and Applications. , ISSN 1550–445X, 15–22.
- Henning, J. L. (2006):** SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, vol. 34 no. 4, 1–17 (URL: <http://doi.acm.org/10.1145/1186736.1186737>) last accessed 2010-06-30, ISSN 0163–5964.
- Herbst, N. R.; Huber, N.; Kounev, S.; Amrehn, E. (2014):** Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and computation: practice and experience*, vol. 26 no. 12, 2053–2078.
- Hevner, A.; March, S.; Park, J.; Ram, S. (2004):** Design Science in Information Systems Research. *MIS quarterly*, vol. 28 no. 1, 75–105.
- Hofer, P.; Hörschläger, F.; Mössenböck, H. (2015):** Sampling-based Steal Time Accounting Under Hardware Virtualization. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE 2015). ACM, New York, NY, USA, ISBN 978–1–4503–3248–4, 87–90.
- Honk, J. (2014):** How many Types Memory Areas Allocated by JVM. USA, Blog Article (URL: <http://javahonk.com/how-many-types-memory-areas-allocated-by-jvm/>) last accessed 2010-06-30.

- Hoorn, A. van; Waller, J.; Hasselbring, W. (2012):** Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. ACM, New York, NY, USA, ICPE '12 [⟨URL: http://doi.acm.org.eaccess.ub.tum.de/10.1145/2188286.2188326⟩](http://doi.acm.org.eaccess.ub.tum.de/10.1145/2188286.2188326) last accessed 2010-06-30, ISBN 978-1-4503-1202-8, 247–248.
- Hrischuk, C.; Woodside, M.; Rolia, J. (1999):** Trace-based Load Characterization for Generating Performance Software Models. *IEEE Transactions on Software Engineering*, vol. 25 no. 1, 122–135, ISSN 0098-5589.
- Huang, R.; Masanet, E. (2015):** Data Center IT Efficiency Measures. National Renewable Energy Laboratory (NREL), Golden, CO. – technical report.
- Huber, N.; Brosig, F.; Spinner, S.; Kounev, S.; Bahr, M. (2016):** Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language. *IEEE Transactions on Software Engineering*, vol. PP no. 99, 1–1, ISSN 0098-5589.
- Huber, N.; Quast, M. von; Hauck, M.; Kounev, S. (2011):** Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. In Proceedings of the 1st International Conference on Cloud Computing and Services Science (CLOSER 2011). SciTePress, Av. Dom Manuel i, 2910-582 Setúbal, Portugal, ISBN 978-989-8425-52-2, 563–573.
- Jones, R.; Hosking, A.; Moss, E. (2016):** The Garbage Collection Handbook: The Art of Automatic Memory Management. CRC Press, USA.
- Jones, R.; Lins, R. D. (1996):** Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, USA.
- Koch-Kemper, B. (2015):** Performance Model Generation of Distributed Java Enterprise Edition Applications considering CPU, Memory and I/O Resources. Master's thesis Technische Universität München.
- Kounev, S. (2005):** Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction. Dissertation, Technische Universität Darmstadt, Germany, Darmstadt, Germany.
- Kounev, S.; Brosig, F.; Huber, N. (2014):** The Descartes Modeling Language. *Department of Computer Science, University of Wuerzburg, Tech. Rep.*, 1–93.
- Kowall, J.; Cappelli, W. (2012):** Magic quadrant for Application Performance Monitoring. 56 Top Gallant Road, Stamford, CT, USA.
- Koziulek, A.; Koziulek, H.; Reussner, R. (2011):** PerOpteryx: Automated Application of Tactics in Multi-objective Software Architecture Optimization. In Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS. ACM, 33–42.

- Koziolok, H. (2010):** Performance Evaluation of Component-based Software Systems: A Survey. *Performance Evaluation*, vol. 67 no. 8, 634 – 658 [⟨URL: http://www.sciencedirect.com/science/article/pii/S016653160900100X⟩](http://www.sciencedirect.com/science/article/pii/S016653160900100X) last accessed 2010-06-30, Special Issue on Software and Performance, ISSN 0166–5316.
- Koziolok, H.; Becker, S.; Happe, J.; Tuma, P.; Gooijer, T. de (2014):** Towards Software Performance Engineering for Multicore and Manycore Systems. *ACM SIGMETRICS Performance Evaluation Review*, vol. 41 no. 3, 2–11.
- Kroß, J.; Willnecker, F.; Zwickl, T.; Krcmar, H. (2016):** PET: Continuous Performance Evaluation Tool. In Proceedings of the 2nd International Workshop on Quality-Aware DevOps. ACM, New York, NY, USA, QUDOS 2016, ISBN 978–1–4503–4411–1, 42–43.
- Kuperberg, M. (2010):** Quantifying and Predicting the Influence of Execution Platform on Software Component Performance. vol. 5, KIT Scientific Publishing, Karlsruhe, Germany, ISBN 3866447418.
- Levy, Y.; Ellis, T. J. (2006):** A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research. *Informing Science: International Journal of an Emerging Transdiscipline*, vol. 9 no. 1, 181–212.
- Libič, P.; Bulej, L.; Horky, V.; Tůma, P. (2015):** Estimating the Impact of Code Additions on Garbage Collection Overhead. In Computer Performance Engineering. vol. 9272, Springer International Publishing, Gewerbestrasse 11 CH-6330 Cham (ZG) Switzerland [⟨URL: http://dx.doi.org/10.1007/978-3-319-23267-6\\_9⟩](http://dx.doi.org/10.1007/978-3-319-23267-6_9) last accessed 2010-06-30, ISBN 978–3–319–23266–9, 130–145.
- Libič, P.; Bulej, L.; Horky, V.; Tůma, P. (2014):** On the Limits of Modeling Generational Garbage Collector Performance. In Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. ACM, New York, NY, USA, ICPE '14 [⟨URL: http://doi.acm.org/10.1145/2568088.2568097⟩](http://doi.acm.org/10.1145/2568088.2568097) last accessed 2010-06-30, ISBN 978–1–4503–2733–6, 15–26.
- Lukasiewicz, M.; Glaß, M.; Reimann, F.; Teich, J. (2011):** Opt4J - A Modular Framework for Meta-heuristic Optimization. In Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011). Dublin, Ireland, 1723–1730.
- McVoy, L. W.; Staelin, C. et al. (1996):** lmbench: Portable Tools for Performance Analysis. In USENIX annual technical conference. San Diego, CA, USA, 279–294.
- Menascé, D. A. (2008):** Computing Missing Service Demand Parameters for Performance Models. In Proceedings of the 2008 Computer Measurement Group Conference (CMG 2008). Las Vegas, NV, USA, 241–248.
- Mendel, T. (2013):** Vendor Selection Matric - Depp Application Transaction Management. Research in Action - Independent research and consulting – technical report [⟨URL: http://www.amasol.de/files/the\\_need\\_for\\_speed-research\\_in\\_action.pdf⟩](http://www.amasol.de/files/the_need_for_speed-research_in_action.pdf) last accessed 2010-06-30.
- Merkle, P.; Henss, J. (2011):** EventSim—An Event-driven Palladio Software Architecture Simulator. *Palladio Days*, 15–22.

- Okanović, D.; Hoorn, A. van; Heger, C.; Wert, A.; Siegl, S. (2016):** Towards Performance Tooling Interoperability: An Open Format for Representing Execution Traces. In **Fiems, D.; Paolieri, M.; Platis, A. N. eds.:** Computer Performance Engineering: 13th European Workshop, EPEW 2016, Chios, Greece, October 5-7, 2016, Proceedings. Springer International Publishing, Cham (URL: [http://dx.doi.org/10.1007/978-3-319-46433-6\\_7](http://dx.doi.org/10.1007/978-3-319-46433-6_7)) last accessed 2010-06-30, ISBN 978-3-319-46433-6, 94-108.
- Oracle Cooperation (2015):** Java Garbage Collection Basics. USA (URL: <http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>) last accessed 2010-06-30.
- Oransa, O. (2014):** Java EE 7 Performance Tuning and Optimization. Packt Publishing Ltd, Birmingham, UK.
- Pawlish, M.; Varde, A.; Robila, S. (2012):** Analyzing utilization rates in data centers for optimizing energy management. In Green Computing Conference (IGCC), 2012 International., 1-6.
- Peffers, K.; Tuunanen, T.; Rothenberger, M. A.; Chatterjee, S. (2007):** A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, vol. 24 no. 3, 45-77 (URL: <http://www.tandfonline.com/doi/abs/10.2753/MIS0742-1222240302>) last accessed 2010-06-30.
- Pitakrat, T.; Okanović, D.; Hoorn, A. van; Grunske, L. (2017):** Hora: Architecture-aware Online Failure Prediction. *Journal of Systems and Software*, -.
- Reussner, R.; Becker, S.; Burger, E.; Happe, J.; Hauck, M.; Koziolk, A.; Koziolk, H.; Krogmann, K.; Kuperberg, M. (2009):** The Palladio Component Model. *Journal of Systems and Software*, vol. 82 no. 1, 3 - 22.
- Reussner, R. H.; Becker, S.; Happe, J.; Heinrich, R.; Koziolk, A.; Koziolk, H.; Kramer, M.; Krogmann, K. (2016):** Modeling and Simulating Software Architectures: The Palladio Approach. MIT Press, Cambridge, MA, USA.
- Rolia, J. (1999):** Trace-based load characterization for generating performance software models. *Software Engineering, IEEE Transactions on*, vol. 25 no. 1, 122-135, ISSN 0098-5589.
- Rolia, J.; Vetland, V. (1995):** Parameter Estimation for Performance Models of Distributed Application Systems. In Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research (CASCON '95). IBM Press, USA, 54.
- Rolia, J. A.; Sevcik, K. C. (1995):** The Method of Layers. *IEEE Transactions on Software Engineering*, vol. 21 no. 8, 689-700.
- Roussel, A.; Branson, R. (2017):** The Million Dollar Engineering Problem. USA, Web <https://segment.com/blog/the-million-dollar-eng-problem/>.

- Rumbaugh, J.; Jacobson, I.; Booch, G. (2004):** The Unified Modeling Language Reference Manual. Addison-Wesley Longman, Amsterdam, Netherlands.
- Salehie, M.; Tahvildari, L. (2009):** Self-adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.* vol. 4 no. 2, 14:1–14:42 (URL: <http://doi.acm.org/10.1145/1516533.1516538>) last accessed 2010-06-30, ISSN 1556–4665.
- Saraswati, S.; Chatterjee, S.; Ramachandra, R. (2016):** Steal-A-GC: Framework to Trigger GC during Idle Periods in Distributed Systems. In 2016 IEEE 23rd International Conference on High Performance Computing (HiPC)., 392–400.
- Schildt, H. (2014):** Java: The Complete Reference. Ninth Edition edition. McGraw-Hill Education, New York, NY, USA.
- Simon, H. A. (1996):** The Sciences of the Artificial. MIT press, Boston, USA.
- Sjoeberg, D. I. K.; Hannay, J. E.; Hansen, O.; Kampenes, V. B.; Karahasanovic, A.; Liborg, N. K.; Rekdal, A. C. (2005):** A Survey of Controlled Experiments in Software Engineering. *IEEE Transactions on Software Engineering*, vol. 31 no. 9, 733–753, ISSN 0098–5589.
- Smith, C. U. (2007):** Introduction to Software Performance Engineering: Origins and Outstanding Problems. In Proceedings of the 7th International Conference on Dormal Methods for Performance Evaluation., 395–428.
- Speitkamp, B.; Bichler, M. (2010):** A Mathematical Programming Approach for Server Consolidation Problems in Virtualized Data Centers. *Services Computing, IEEE Transactions on*, vol. 3 no. 4, 266–278.
- Spinner, S. (2011):** Evaluating Approaches to Resource Demand estimation. Diplomarbeit, Karlsruhe Institute of Technology (KIT).
- Spinner, S.; Casale, G.; Brosig, F.; Kounev, S. (2015):** Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation*, vol. 92, 51 – 71 (URL: <http://www.sciencedirect.com/science/article/pii/S0166531615000711>) last accessed 2010-06-30, ISSN 0166–5316.
- Spinner, S.; Casale, G.; Zhu, X.; Kounev, S. (2014):** LibReDE: A Library for Resource Demand Estimation. In Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014). ACM, New York, NY, USA, ISBN 978–1–4503–2733–6, 227–228.
- Verma, A.; Pedrosa, L.; Korupolu, M.; Oppenheimer, D.; Tune, E.; Wilkes, J. (2015):** Large-scale Cluster Management at Google with Borg. In Proc. EuroSys '15. ACM, New York, NY, USA (URL: <http://doi.acm.org/10.1145/2741948.2741964>) last accessed 2010-06-30, ISBN 978–1–4503–3238–5, 18:1–18:17.
- Vögele, C.; Heinrich, R.; Heilein, R.; Hoorn, A. v.; Krcmar, H. (2015):** Modeling Complex User Behavior with the Palladio Component Model. In Proceedings of the Symposium on Software Performance (SSP) 2015. GI - Softwaretechnik-Trends, Siegen, Germany.

- Walter, J.; Stier, C.; Koziol, H.; Kounev, S. (2017):** An Expandable Extraction Framework for Architectural Performance Models. In Proceedings of the 2017 International Workshop on Quality-Aware DevOps (QUDOS'17) co-located with 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017). ACM.
- Wang, W.; Huang, X.; Qin, X.; Zhang, W.; Wei, J.; Zhong, H. (2012):** Application-Level CPU Consumption Estimation: Towards Performance Isolation of Multi-tenancy Web Applications. In Proceedings of the 5th International Conference on Cloud Computing (CLOUD). IEEE, Piscataway, New Jersey, USA, 439–446.
- Webster, J.; Watson, R. T. (2002):** Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Quarterly*, vol. 26 no. 2, xiii–xxiii (URL: <http://www.jstor.org/stable/4132319>) last accessed 2010-06-30, ISSN 02767783.
- Wieringa, R. J. (2014):** Design Science Methodology for Information Systems and Software Engineering. Springer-Verlag, Berlin Heidelberg.
- Wilnecker, F.; Brunnert, A.; Gottesheim, W.; Krcmar, H. (2015a):** Using Dynatrace Monitoring Data for Generating Performance Models of Java EE Applications. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE 2015). ACM, New York, NY, USA, ISBN 978-1-4503-3248-4, 103–104.
- Wilnecker, F.; Brunnert, A.; Koch-Kemper, B.; Krcmar, H. (2015b):** Full-Stack Performance Model Evaluation using Probabilistic Garbage Collection Simulation. In Proceedings of the 2015 Symposium on Software Performance (SSP 2015)..
- Wilnecker, F.; Dlugi, M.; Brunnert, A.; Spinner, S.; Kounev, S.; Gottesheim, W.; Krcmar, H. (2015c):** Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques. In Computer Performance Engineering. Springer International Publishing, Gewerbestrasse 11 CH-6330 Cham (ZG) Switzerland, Lecture Notes in Computer Science (URL: [http://dx.doi.org/10.1007/978-3-319-23267-6\\_8](http://dx.doi.org/10.1007/978-3-319-23267-6_8)) last accessed 2010-06-30, ISBN 978-3-319-23266-9, 115–129.
- Wilnecker, F.; Krcmar, H. (2016):** Optimization of Deployment Topologies for Distributed Enterprise Applications. In Proceedings of the 12th International ACM Sigsoft Conference QoSA 2016. ACM.
- Woodside, M.; Franks, G.; Petriu, D. C. (2007):** The Future of Software Performance Engineering. In Future of Software Engineering (FOSE). IEEE, Washington, DC, USA, 171–187.
- Wolf, B. (2009):** WebSphere SOA and JEE in Practice. Web (URL: [https://www.ibm.com/developerworks/community/blogs/woolf/entry/websphere\\_process\\_server\\_golden\\_topology?lang=en](https://www.ibm.com/developerworks/community/blogs/woolf/entry/websphere_process_server_golden_topology?lang=en)) last accessed 2010-06-30.
- Zheng, T.; Woodside, M.; Litoiu, M. (2008):** Performance Model Estimation and Tracking Using Optimal Filters. *IEEE Transactions on Software Engineering*, vol. 34 no. 3, 391–406.