Simon Spinner

Self-Aware Resource Management in Virtualized Data Centers

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2017
Erster Gutachter: Prof. Dr.-Ing. Samuel Kounev
Zweiter Gutachter: Prof. Dr. rer. nat. Kurt Geihs
Datum der mündlichen Prüfung: 28.07.2017

Abstract

Enterprise applications in virtualized data centers are often subject to time-varying workloads, i.e., the load intensity and request mix change over time, due to seasonal patterns and trends, or unpredictable bursts in user requests. Varying workloads result in frequently changing resource demands to the underlying hardware infrastructure. Virtualization technologies enable sharing and on-demand allocation of hardware resources between multiple applications. In this context, the resource allocations to virtualized applications should be continuously adapted in an *elastic* fashion, so that "at each point in time the available resources match the current demand as closely as possible" (Herbst et al., 2013). Autonomic approaches to resource management promise significant increases in resource efficiency while avoiding violations of performance and availability requirements during peak workloads.

Traditional approaches for autonomic resource management use *threshold-based* rules (e.g., Amazon EC2) that execute pre-defined reconfiguration actions when a metric reaches a certain threshold (e.g., high resource utilization or load imbalance). However, many business-critical applications are subject to Service-level Objectives (SLOs) defined on an application performance metric (e.g., response time or throughput). To determine thresholds so that the end-to-end application SLO is fulfilled poses a major challenge due to the complex relationship between the resource allocation to an application and the application performance. Furthermore, threshold-based approaches are inherently prone to an oscillating behavior resulting in unnecessary reconfigurations.

In order to overcome the deficiencies of threshold-based approaches and enable a fully automated approach to dynamically control the resource allocations of virtualized applications, *model-based* approaches are required that can predict the impact of a reconfiguration on the application performance in advance. However, existing model-based approaches are severely limited in their learning capabilities. They either require complete performance models of the application as input, or use a pre-identified model structure and only learn certain model parameters from empirical data at run-time. The former requires high manual efforts and deep system knowledge to create the performance models. The latter does not provide the flexibility to capture the specifics of complex and heterogeneous system architectures.

This thesis presents a *self-aware* approach to the resource management in virtualized data centers. In this context, self-aware means that it automatically learns performance models of the application and the virtualized infrastructure and reasons based on these models to autonomically adapt the resource allocations in accordance with given application SLOs. Learning a performance model requires the extraction of the model structure representing the system architecture as well as the estimation of model parameters, such as resource demands. The estimation of resource demands is a key challenge as they cannot be observed directly in most systems. The major scientific contributions of this thesis are:

- *A reference architecture for online model learning in virtualized systems.* Our reference architecture is based on a set of model extraction agents. Each agent focuses on specific tasks to automatically create and update model skeletons capturing its local knowledge of the system and collaborates with other agents to extract the structural parts of a global performance model of the system. We define different agent roles in the reference architecture and propose a model-based collaboration mechanism for the agents. The agents may be bundled within virtual appliances and may be tailored to include knowledge about the software stack deployed in a specific virtual appliance.

- *An online method for the statistical estimation of resource demands.* For a given request processed by an application, the resource time consumed for a specified resource within the system (e.g., CPU or I/O device), referred to as *resource demand*, is the total average time the resource is busy processing the request. A request could be any unit of work (e.g., web page request, database transaction, batch job) processed by the system. We provide a systematization of existing statistical approaches to resource demand estimation and conduct an extensive experimental comparison to evaluate the accuracy of these approaches. We propose a novel method to automatically select estimation approaches and demonstrate that it increases the robustness and accuracy of the estimated resource demands significantly.

- *Model-based controllers for autonomic vertical scaling of virtualized applications.* We design two controllers based on online model-based reasoning techniques in order to vertically scale applications at run-time in accordance with application SLOs. The controllers exploit the knowledge from the automatically extracted performance models when determining necessary reconfigurations. The first controller adds and removes virtual CPUs

to an application depending on the current demand. It uses a layered performance model to also consider the physical resource contention when determining the required resources. The second controller adapts the resource allocations proactively to ensure the availability of the application during workload peaks and avoid reconfiguration during phases of high workload.

We demonstrate the applicability of our approach in current virtualized environments and show its effectiveness leading to significant increases in resource efficiency and improvements of the application performance and availability under time-varying workloads. The evaluation of our approach is based on two case studies representative of widely used enterprise applications in virtualized data centers. In our case studies, we were able to reduce the amount of required CPU resources by up to 23% and the number of reconfigurations by up to 95% compared to a rule-based approach while ensuring full compliance with application SLOs. Furthermore, using workload forecasting techniques we were able to schedule expensive reconfigurations (e.g., changes to the memory size) during phases of load load and thus were able to reduce their impact on application availability by over 80% while significantly improving application performance compared to a reactive controller. The methods and techniques for resource demand estimation and vertical application scaling were developed and evaluated in close collaboration with VMware and Google.

Zusammenfassung

Unternehmensanwendungen in virtualisierten Rechenzentren unterliegen häufig zeitabhängigen Arbeitslasten, d.h. die Lastintensität und der Anfragemix ändern sich mit der Zeit wegen saisonalen Mustern und Trends, sowie unvorhergesehenen Lastspitzen bei den Nutzeranfragen. Variierende Arbeitslasten führen dazu, dass sich die Ressourcenanforderungen an die darunterliegende Hardware-Infrastruktur häufig ändern. Virtualisierungstechniken erlauben die gemeinsame Nutzung und bedarfsgesteuerte Zuteilung von Hardware-Ressourcen zwischen mehreren Anwendungen. In diesem Zusammenhang sollte die Zuteilung von Ressourcen an virtualisierte Anwendungen fortwährend in einer *elastischen* Art und Weise angepasst werden, um sicherzustellen, dass „zu jedem Zeitpunkt die verfügbaren Ressourcen dem derzeitigen Bedarf möglichst genau entsprechen" (Herbst u. a., 2013). Autonome Ansätze zur Ressourcenverwaltung versprechen eine deutliche Steigerung der Ressourceneffizienz wobei Verletzungen der Anforderungen hinsichtlich Performanz und Verfügbarkeit bei Lastspitzen vermieden werden.

Herkömmliche Ansätze zur autonomen Ressourcenverwaltung nutzen feste Regeln (z.B., Amazon EC2), die vordefinierte Rekonfigurationen durchführen sobald eine Metrik einen bestimmten Schwellwert erreicht (z.B., hohe Ressourcenauslastung oder ungleichmäßige Lastverteilung). Viele geschäftskritische Anwendungen unterliegen jedoch Zielvorgaben hinsichtlich der Dienstgüte (SLO, engl. Service Level Objectives), die auf Performanzmetriken der Anwendung definiert sind (z.B., Antwortzeit oder Durchsatz). Die Bestimmung von Schwellwerten, sodass die Ende-zu-Ende Anwendungs-SLOs erfüllt werden, stellt aufgrund des komplexen Zusammenspiels zwischen der Ressourcenzuteilung und der Performanz einer Anwendung eine bedeutende Herausforderung dar. Des Weiteren sind Ansätze basierend auf Schwellwerten inhärent anfällig für Oszillationen, die zu überflüssigen Rekonfigurationen führen können.

Um die Schwächen schwellwertbasierter Ansätze zu lösen und einen vollständig automatisierten Ansatz zur dynamischen Steuerung von Ressourcenzuteilungen virtualisierter Anwendungen zu ermöglichen, bedarf es *modellbasierter* Ansätze, die den Einfluss einer Rekonfiguration auf die Performanz einer Anwendung im Voraus vorhersagen können. Bestehende modellbasierte Ansätze sind jedoch stark eingeschränkt hinsichtlich ihrer Lernfähigkeiten. Sie erfor-

dern entweder vollständige Performanzmodelle der Anwendung als Eingabe oder nutzen vorbestimmte Modellstrukturen und lernen nur bestimmte Modellparameter auf Basis von empirischen Daten zur Laufzeit. Erstere erfordern hohe manuelle Aufwände und eine tiefe Systemkenntnis um die Performanzmodelle zu erstellen. Letztere bieten nur eingeschränkte Möglichkeiten um die Besonderheiten von komplexen und heterogenen Systemarchitekturen zu erfassen.

Diese Arbeit stellt einen *selbstwahrnehmenden* (engl. self-aware) Ansatz zur Ressourcenverwaltung in virtualisierten Rechenzentren vor. In diesem Zusammenhang bedeutet Selbstwahrnehmung, dass der Ansatz automatisch Performanzmodelle der Anwendung und der virtualisierten Infrastruktur lernt Basierend auf diesen Modellen entscheidet er autonom wie die Ressourcenzuteilungen angepasst werden, um die Anwendungs-SLOs zu erfüllen. Das Lernen von Performanzmodellen erfordert sowohl die Extraktion der Modellstruktur, die die Systemarchitektur abbildet, als auch die Schätzung von Modellparametern, wie zum Beispiel der Ressourcenverbräuche einzelner Funktionen. Die Schätzung der Ressourcenverbräuche stellt hier eine zentrale Herausforderung dar, da diese in den meisten Systemen nicht direkt gemessen werden können. Die wissenschaftlichen Hauptbeiträge dieser Arbeit sind wie folgt:

- *Eine Referenzarchitektur, die das Lernen von Modellen in virtualisierten Systemen während des Betriebs ermöglicht*. Unsere Referenzarchitektur basiert auf einer Menge von Modellextraktionsagenten. Jeder Agent fokussiert sich auf bestimmte Aufgaben um automatisch ein Modellskeleton, das sein lokales Wissen über das System erfasst, zu erstellen und zu aktualisieren. Jeder Agent arbeitet mit anderen Agenten zusammen um die strukturellen Teile eines globalen Performanzmodells des Systems zu extrahieren. Die Rereferenzarchitektur definiert unterschiedliche Agentenrollen und beinhaltet einen modellbasierten Mechanismus, der die Kooperation unterschiedlicher Agenten ermöglicht. Die Agenten können als Teil virtuellen Appliances gebündelt werden und können dabei maßgeschneidertes Wissen über die Software-Strukturen in dieser virtuellen Appliance beinhalten.

- *Eine Methode zur fortwährenden statistischen Schätzung von Ressourcenverbräuchen*. Der *Ressourcenverbrauch* (engl. resource demand) einer Anfrage, die von einer Anwendung verarbeitet wird, entspricht der Zeit, die an einer spezifischen Ressource im System (z.B., CPU oder I/O-Gerät) verbraucht wird. Eine Anfrage kann dabei eine beliebige Arbeitseinheit, die von einem System verarbeitet wird, darstellen (z.B. eine Webseitenanfra-

ge, eine Datenbanktransaktion, oder ein Stapelverarbeitungsauftrag). Die vorliegende Arbeit bietet eine Systematisierung existierender Ansätze zur statistischen Schätzung von Ressourcenverbräuchen und führt einen umfangreichen, auf Experimenten aufbauenden Vergleich zur Bewertung der Genauigkeit dieser Ansätze durch. Es wird eine neuartige Methode zur automatischen Auswahl eines Schätzverfahrens vorgeschlagen und gezeigt, dass diese die Robustheit und Genauigkeit der geschätzten Ressourcenverbräuche maßgeblich verbessert.

- *Modellbasierte Regler für das autonome, vertikale Skalieren von virtualisierten Anwendungen*. Es werden zwei Regler entworfen, die auf modellbasierten Entscheidungstechniken basieren, um Anwendungen zur Laufzeit vertikal in Übereinstimmung mit Anwendungs-SLOs zu skalieren. Die Regler nutzen das Wissen aus automatisch extrahierten Performanzmodellen bei der Bestimmung notwendiger Rekonfigurationen. Der erste Regler fügt virtuelle CPUs zu Anwendungen hinzu und entfernt sie wieder in Abhängigkeit vom aktuellen Bedarf. Er nutzt ein geschichtetes Performanzmodell, um bei der Bestimmung der benötigten Ressourcen die Konkurrenzsituation der physikalischen Ressourcen zu beachten. Der zweite Regler passt Ressourcenzuteilungen proaktiv an, um die Verfügbarkeit einer Anwendung während Lastspitzen sicherzustellen und Rekonfigurationen unter großer Last zu vermeiden.

Die Arbeit demonstriert die Anwendbarkeit unseres Ansatzes in aktuellen virtualisierten Umgebungen und zeigt seine Effektivität bei der Erhöhung der Ressourceneffizienz und der Verbesserung der Anwendungsperformanz und -verfügbarkeit unter zeitabhängigen Arbeitslasten. Die Evaluation des Ansatzes basiert auf zwei Fallstudien, die repräsentativ für gängige Unternehmensanwendungen in virtualisierten Rechenzentren sind. In den Fallstudien wurde eine Reduzierung der benötigten CPU-Ressourcen von bis zu 23% und der Anzahl der Rekonfigurationen von bis zu 95% im Vergleich zu regel-basierten Ansätzen erreicht, bei gleichzeitiger Erfüllung der Anwendungs-SLOs. Mit Hilfe von Vorhersagetechniken für die Arbeitslast konnten außerdem aufwändige Rekonfigurationen (z.B., Änderungen bei der Menge an zugewiesenem Arbeitsspeicher) so geplant werden, dass sie in Phasen geringer Last durchgeführt werden. Dadurch konnten deren Auswirkungen auf die Verfügbarkeit der Anwendung um mehr als 80% verringert werden bei gleichzeitiger Verbesserung der Anwendungsperformanz verglichen mit einem reaktiven Regler. Die Methoden und Techniken zur Schätzung von Ressourcenverbräuchen und zur

vertikalen Skalierung von Anwendungen wurden in enger Zusammenarbeit mit VMware und Google entwickelt und evaluiert.

Acknowledgments

The thesis would have been impossible without the aid and support of many people. First of all, I would like to thank my advisor Prof. Samuel Kounev. I first met him at the beginning of my master studies, and since then he always supported me with advice and encouragement on my journey in the academic world. He was a constant source of inspiration and motivation in all these years guiding my work on this thesis, on research papers, and on grant proposals. I would also like to thank Prof. Ralf Reussner for hosting the Descartes research group at SDQ and providing an enjoyable working environment at the KIT.

I would like to thank VMware Inc. for their support of my research work and especially Xiaoyun Zhu, Lei Lu, Rean Griffith, Anne Holler, and Mustafa Uysal for their valuable feedback and suggestions in many remote discussions, as well as their warm and friendly welcome during two internships in Palo Alto. Furthermore, I would also like to thank Google Inc. for their interest in the LibReDE tool and Arif Merchant for the fruitful discussions.

From the SE group at the University of Würzburg, I want to thank my current and former colleagues I had the pleasure to work with on many projects: Fabian Brosig, Prof. Jürgen Wolff von Gudenberg, Nikolas Herbst, Nikolaus Huber, Lukas Iffländer, Jóakim von Kistowski, Fritz Kleemann, Rouven Krebs, Aleksandar Milenkoski, Marco Nehmeier, Piotr Rygielski, Susanne Stenglin, and Jürgen Walter. Many thanks goes also to my former colleagues from the SDQ group at the KIT and the FZI.

Furthermore, I would like to thank Nadia Ahmed, André Bauer, Johannes Grohmann, Jürgen Walter and Can Alexander Yavuz for supporting me as students as part of their bachelor's or master's theses; and Mehran Saliminia, Frederik König, and Veronika Lesch for supporting me as research students in various projects.

Finally, I would like to thank my parents and my sister for their support and encouragement throughout the years, which made it all possible.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 State-of-the-Art . . . 4
- 1.4 Approach and Contributions . . . 6
  - 1.4.1 Contributions . . . 6
  - 1.4.2 Evaluation . . . 10
- 1.5 Thesis Outline . . . 11

I Foundations and Related Work 13

2 Foundations of Self-Aware Computing 15
- 2.1 Self-Aware Computing . . . 15
- 2.2 Model-based Knowledge Representation . . . 17
  - 2.2.1 Model-Driven Engineering . . . 18
  - 2.2.2 Descartes Modeling Language (DML) . . . 20
- 2.3 Statistical Techniques for Model Parameterization . . . 25
  - 2.3.1 Regression Analysis . . . 25
  - 2.3.2 Kalman Filter . . . 26
  - 2.3.3 Mathematical Optimization . . . 28
  - 2.3.4 Maximum Likelihood Estimation . . . 29
  - 2.3.5 Bayesian Inference . . . 29
- 2.4 Performance Prediction . . . 31
  - 2.4.1 Queueing Models . . . 31
  - 2.4.2 Layered Queueing Networks (LQNs) . . . 37
- 2.5 Server Virtualization . . . 39
  - 2.5.1 Virtualization Techniques . . . 39
  - 2.5.2 Open Virtualization Format (OVF). . . . 41
  - 2.5.3 Run-time Reconfiguration . . . 41

3 State-of-the-Art 45
3.1 Autonomic Resource Management in Virtualized Data Centers . 45
3.1.1 Capacity Planning 45
3.1.2 Local Resource Scheduling 46
3.1.3 Global Resource Scheduling 47
3.1.4 Application Workload Management 48
3.1.5 Application Scaling 48
3.1.6 Discussion 52
3.2 Performance Model Extraction 53
3.2.1 Monitoring, Instrumentation and Measurement-Based Analysis 53
3.2.2 Static Structure and Dynamic Behavior 55
3.2.3 Workload Characterization 57
3.2.4 Discussion 59

II Self-Aware Resource Management in Virtualized Systems 61

4 Online Model Learning in Virtualized Data Centers 63
4.1 Reference Architecture for Online Model Learning 66
4.1.1 Conceptual Overview 66
4.1.2 Agent Collaboration 72
4.1.3 Extraction Scopes 76
4.2 Model Extraction Agents 79
4.2.1 Data Center Scope 80
4.2.2 Usage Scope 83
4.2.3 Platform Scope 85
4.2.4 Application Scope 87
4.2.5 Model Variable Scopes 91
4.3 Performance Model Repository 94
4.3.1 Model Skeleton Composition 95
4.3.2 Merge Algorithm 97
4.4 Concluding Remarks 101

5 Online Statistical Estimation of Resource Demands 103
5.1 Systematization of Approaches 105
5.1.1 Estimation Approaches 106
5.1.2 Input Parameters 113
5.1.3 Output Metrics 116
5.1.4 Robustness 119

5.2 Experimental Comparison . . . 121
5.2.1 Experiment Setup . . . 122
5.2.2 Comparison of Estimation Approaches . . . 123
5.2.3 Results Summary . . . 136
5.3 Library for Resource Demand Estimation (LibReDE) . . . 138
5.3.1 Method Overview . . . 139
5.3.2 Derivation of Workload Description . . . 143
5.3.3 Derivation of Estimation Problems . . . 145
5.3.4 Cross-Validation and Approach Selection . . . 147
5.4 Concluding Remarks . . . 149

6 Model-based Vertical Scaling of Virtualized Applications 151
6.1 Short-term Vertical CPU Scaling . . . 154
6.1.1 Model-Adaptive Control Loop . . . 154
6.1.2 Modeling Approach . . . 156
6.1.3 Model Estimation . . . 160
6.1.4 Resource Control . . . 162
6.2 Mid- and Long-term Vertical Scaling . . . 167
6.2.1 Factors Limiting Application Elasticity . . . 168
6.2.2 Proactive Control Loop . . . 169
6.2.3 Evaluation of Forecast Accuracy . . . 173
6.3 Concluding Remarks . . . 176

III Validation and Conclusions 179

7 Validation 181
7.1 Evaluation Goals . . . 181
7.2 Integration into Existing Software Systems . . . 183
7.2.1 VMware vSphere . . . 183
7.2.2 Zimbra Collaboration Server . . . 185
7.2.3 Wildfly Application Server . . . 187
7.2.4 Discussion . . . 192
7.3 Use Cases of LibReDE . . . 193
7.3.1 Resource Usage Control in SAP HANA Cloud . . . 193
7.3.2 Offline Generation of Palladio Component Models . . . 195
7.3.3 Discussion . . . 196
7.4 Case Study: Distributed SPECjEnterprise2010 . . . 197
7.4.1 Degree of Automation . . . 200
7.4.2 Accuracy of Estimated Resource Demands . . . 202

7.4.3 Model Prediction Accuracy . . . 205
7.4.4 Discussion . . . 210
7.5 Case Study: Zimbra Collaboration Server . . . 211
7.5.1 Experiment Setup . . . 211
7.5.2 Application Scalability . . . 212
7.5.3 Physical Resource Contention . . . 215
7.5.4 Short-term CPU Scaling . . . 216
7.5.5 Mid- to Long-term Memory Scaling . . . 219
7.5.6 Discussion . . . 224

8 Conclusions 225
8.1 Summary . . . 225
8.2 Benefits . . . 228
8.3 Future Work . . . 229

Bibliography 241

Publications

Journals

Huber, N., F. Brosig, S. Spinner, S. Kounev, and M. Bähr (2017). "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language". In: *IEEE Transactions on Software Engineering (TSE)* 43.5 (see pages 6, 70, 185, 203, 225).

Spinner, S., G. Casale, F. Brosig, and S. Kounev (2015a). "Evaluating Approaches to Resource Demand Estimation". In: *Elsevier Performance Evaluation* 92, pp. 51–71. ISSN: 0166-5316 (see pages 9, 106, 121, 227).

Walter, J., S. Spinner, and S. Kounev (2015b). "Parallel Simulation of Queueing Petri Nets". In: *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems* 16.8.

Peer-Reviewed International Conference Contributions

Research Papers

Grohmann, J., N. Herbst, S. Spinner, and S. Kounev (2017). "Self-Tuning Resource Demand Estimation". (Short Paper). In: *Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017)* (see page 148).

Spinner, S., N. Herbst, S. Kounev, X. Zhu, L. Lu, M. Uysal, and R. Griffith (2015b). "Proactive Memory Scaling of Virtualized Applications". In: *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing (IEEE CLOUD 2015)*. Acceptance Rate: 15%. New York, NY, USA: IEEE, pp. 277–284 (see pages 10, 168, 219, 228).

Walter, J., S. Spinner, and S. Kounev (2015a). "Parallel Simulation of Queueing Petri Nets". In: *Proceedings of the Eighth EAI International Conference on Simulation Tools and Techniques (SIMUTools 2015)*. Athens, Greece.

Spinner, S., S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith (2014b). "Runtime Vertical Scaling of Virtualized Applications via Online Model Estimation". In: *Proceedings of the 2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. Acceptance Rate (Full Papers): 26%. London, UK: IEEE, pp. 157–166 (see pages 10, 154, 228).

Krebs, R., S. Spinner, N. Ahmed, and S. Kounev (2014a). "Resource Usage Control In Multi-Tenant Applications". In: *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014)*. Acceptance Rate: 19% (54/283). Chicago, IL, USA: IEEE/ACM, pp. 122–131 (see pages 9, 193, 195, 227).

Tutorial/Tool/Talk Extend Abstract Papers

Kounev, S., N. Huber, F. Brosig, S. Spinner, and M. Baehr (2017a). "Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language". (Talk Extended Abstract). In: *Jan Jürjens, Kurt Schneider (Hrsg.): Software Engineering 2017 (SE 2017), Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-24. Februar 2017, Hannover, Germany*. Lecture Notes in Informatics (LNI). Hannover, Germany: Gesellschaft für Informatik (GI).

Casale, G., S. Spinner, and W. Wang (2016). "Automated Parameterization of Performance Models from Measurements". (Tutorial Paper). In: *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*. Delft, the Netherlands.

Spinner, S., G. Casale, X. Zhu, and S. Kounev (2014a). "LibReDE: A Library for Resource Demand Estimation". (Demo Paper). In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. Dublin, Ireland: ACM Press, pp. 227–228 (see pages 9, 139, 227).

Spinner, S., S. Kounev, and P. Meier (2012). "Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0". (Tool Paper). In: *Proceedings of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2012)*. Ed. by S. Haddad and L. Pomello. Vol. 7347. Lecture Notes in Computer Science (LNCS). Hamburg, Germany: Springer-Verlag, pp. 388–397.

Kounev, S., S. Spinner, and P. Meier (2012a). "Introduction to Queueing Petri Nets: Modeling Formalism, Tool Support and Case Studies". (Tutorial Paper). In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ICPE*. Boston, Massachusetts, USA: ACM, pp. 9–18 (see page 145).

Peer-Reviewed International Workshop Contributions

Müller, C., P. Rygielski, S. Spinner, and S. Kounev (2016). "Enabling Fluid Analysis for Queueing Petri Nets via Model Transformation". In: *Electronic Notes in Theoretical Computer Science*. Vol. 327. The 8th International Workshop

on Practical Application of Stochastic Modeling, PASM 2016. Elsevier, pp. 71–91 (see page 163).

Spinner, S., J. Walter, and S. Kounev (2016). "A Reference Architecture for Online Performance Model Extraction in Virtualized Environments". In: *Proceedings of the 2016 Workshop on Challenges in Performance Methods for Software Development (WOSP-C'16) co-located with 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*. Delft, the Netherlands (see pages 7, 66, 226).

Willnecker, F., M. Dlugi, A. Brunnert, S. Spinner, S. Kounev, and H. Krcmar (2015). "Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques". In: *Computer Performance Engineering - Proceedings of the 12th European Workshop (EPEW 2015)*. Ed. by M. Beltrán, W. Knottenbelt, and J. Bradley. Vol. 9272. Lecture Notes in Computer Science. Madrid, Spain: Springer, pp. 115–129 (see pages 9, 195, 196, 227).

Spinner, S., S. Kounev, X. Zhu, and M. Uysal (2013). "Towards Online Performance Model Extraction in Virtualized Environments". (Position Paper). In: *Proceedings of the 8th Workshop on Models @ Run.time (MRT 2013) co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*. Ed. by N. Bencomo, R. France, S. Götz, and B. Rumpe. Miami, Florida, USA: CEUR-WS, pp. 89–95 (see pages 7, 66, 226).

Book Chapters

Spinner, S., A. Filieri, S. Kounev, M. Maggio, and A. Robertsson (2017). "Runtime Models for Online Performance and Resource Management in Data Centers". In: *Self-Aware Computing Systems*. Ed. by S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu. Berlin Heidelberg, Germany: Springer Verlag, pp. 485–505.

Kephart, J. O., M. Maggio, A. Diaconescu, S. Spinner, et al. (2017). "Reference Scenarios for Self-Aware Computing". In: *Self-Aware Computing Systems*. Ed. by S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu. Berlin Heidelberg, Germany: Springer Verlag, pp. 87–106.

Walter, J., A. D. Marco, S. Spinner, P. Inverardi, and S. Kounev (2017). "Online Learning of Run-time Models for Performance and Resource Management in Data Centers". In: *Self-Aware Computing Systems*. Ed. by S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu. Berlin Heidelberg, Germany: Springer Verlag, pp. 507–528.

Iosup, A., X. Zhu, A. Merchant, S. Spinner, et al. (2017). "Self-Awareness of Cloud Applications". In: *Self-Aware Computing Systems*. Ed. by S. Kounev, J. O.

Kephart, A. Milenkoski, and X. Zhu. Berlin Heidelberg, Germany: Springer Verlag, pp. 575–610.

Happe, J., B. Klatt, M. Küster, S. Spinner, et al. (2016). "Getting the Data". In: *Modeling and Simulating Software Architectures - The Palladio Approach*. Ed. by R. H. Reussner et al. In Press. Cambridge, MA: MIT Press.

Kounev, S., N. Huber, S. Spinner, and F. Brosig (2012b). "Model-based Techniques for Performance Engineering of Business Information Systems". In: *Business Modeling and Software Design*. Ed. by B. Shishkov. Vol. 0109. Lecture Notes in Business Information Processing (LNBIP). Berlin, Heidelberg: Springer-Verlag, pp. 19–37.

Kounev, S., S. Spinner, and P. Meier (2010). "QPME 2.0 - A Tool for Stochastic Modeling and Analysis Using Queueing Petri Nets". In: *From Active Data Management to Event-Based Systems and More*. Ed. by K. Sachs, I. Petrov, and P. Guerrero. Vol. 6462. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, pp. 293–311.

Industrial Conference Contributions

Spinner, S., X. Zhu, L. Lu, M. Uysal, R. Griffith, N. Herbst, and S. Kounev (2015c). "Proactive Memory Scaling of Virtualized Applications". In: *VMware R&D Innovation Offsite (RADIO)*.

Spinner, S. et al. (2014c). "Model-Based Performance Management for Virtualized Applications". In: *VMware R&D Innovation Offsite (RADIO)*.

Technical Reports

Brunnert, A. et al. (2015). *Performance-oriented DevOps: A Research Agenda*. Tech. rep. SPEC-RG-2015-01. SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC).

Chapter 1

Introduction

1.1 Motivation

IT services hosted in data centers, such as public internet services (e.g., Netflix, Facebook, or Google) as well as intranet services in corporate networks, are typically subject to time-varying workloads. Depending on the current number of users and their interactions with a service, the resource demands of applications providing these services change over time as well. At any point in time, the amount of resources allocated to an application needs to fulfill its current demand. Otherwise, its provided services may violate Service-level Agreements (SLAs) regarding their end-to-end performance[1] and availability. In conventional data centers, IT services are typically hosted on dedicated hardware with over-dimensioned capacity to ensure SLA fulfillment under varying workload conditions and load spikes. As a result, a growing number of underutilized servers causes increasing data center operating costs including system management and power consumption costs. Server virtualization techniques promise substantial reductions in the Total-Cost-of-Ownership (TCO) for IT services by enabling the consolidation of multiple servers on the same physical hardware and the on-demand provisioning of resources to applications. Server virtualization is a base technology for cloud computing and current market research (Gartner, Inc., 2015) shows that over 75% of all x86 servers in data-centers world-wide are virtualized.

However, the consolidation of servers also poses new challenges to the resource management of IT services in virtualized data centers. The higher utilization of physical resources makes IT services much more vulnerable to violations of SLAs resulting from unforeseen workload variations and load spikes. Resource management in virtualized data centers requires a high *elasticity*, i.e., the system must be "able to adapt to workload changes by provisioning

[1] In this thesis, the term *performance* is understood as the degree to which a software system meets its objectives for timeliness and the efficiency with which it achieves this (Kounev, 2008; Smith and Williams, 2002)

and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible" (Herbst et al., 2013). Without such continuous adaptations to the resource allocations of applications in a data center, the benefits of virtualization in terms of improved resource efficiency and lower operating costs are significantly limited since one would have to continue over-provisioning resources to ensure SLAs.

Modern virtualization platforms offer different knobs supporting the elastic allocation of resources to applications. For instance, they support horizontal scaling – i.e., add additional Virtual Machine (VM) instances – as well as vertical scaling – i.e., add resources to an existing VM instance – of applications without noticeable service interruptions. However, existing tools for resource management require system administrators to specify fixed, trigger-based rules in advance that determine when an application should be scaled. Unfortunately, trigger-based tools lack the ability to proactively determine the effects of changing resource allocations on the performance of hosted IT services, e.g., response time and throughput. This significantly limits the possibilities for optimizing resource allocations while ensuring SLA compliance and is currently a great hurdle to leverage the full potential of virtualization to significantly reduce costs for IT.

In order to *proactively* determine the effects of changing resource allocations, models are required that describe the relationship between resource allocation and the observed performance of hosted IT services. While a number of performance modeling and prediction techniques have been developed in the performance engineering community, existing techniques are either too coarse-grained (abstracting systems and applications at a high level), or if being fine-grained, they are typically designed for capacity planning in an offline setting. In the former case, important factors influencing the performance (e.g., software architecture) are missing, limiting the predictive power of the model. In the latter case, the high overhead of manual model creation and maintenance as well as compute-intensive prediction techniques have been a major hurdle for the adoption of such techniques for autonomic performance and resource management at system run-time.

1.2 Problem Statement

In the following, we assume a data center consists of a set of physical servers that are virtualized and that may host any number of VMs. Furthermore, a data center executes a set of applications that are deployed in one or multiple VMs. Each application provides services which are subject to certain SLAs. An SLA

specifies objectives on the performance and availability of services provided by an application in terms of service-level metrics, such as end-to-end response time or throughput. All applications share the hardware resources (e.g., CPU and memory) in a data center and each one is allocated a certain amount of these resources. The resource allocations may be changed at any point in time.

A resource management mechanism should continuously adapt the resource allocations to applications in an autonomic manner in order to minimize the resource usage while ensuring that applications can fulfill their SLAs under time-varying workloads. However, the relationship between the application performance and the resource allocation is highly non-linear and multi-dimensional as it depends on multiple factors including application architecture, system configuration, and resource demands. Therefore, we require models that allow us to predict the expected application performance for a given combination of resource allocation and workload. However, the usage of such models for resource management in virtualized data centers faces multiple challenges:

- *Multi-tier application architectures:* An application may comprise several tiers, each deployed in one or more VMs. The end-to-end application performance depends on the processing in each tier and the flow of requests between tiers. The processing of application requests requires access to different types of resources (e.g., CPU, memory, or I/O). The extent to which each resource contributes to the end-to-end latency may vary between different application tiers. Furthermore, asynchronous communication and limited software resources (e.g., thread pools or connection pools) also influence the achievable application performance.

- *Heterogeneous software stacks:* Inside a VM running in a virtualized data center, any type of software can be deployed. Given that virtualized data centers may be shared between different applications and also between different organizations, the software stacks running inside VMs are often heterogeneous and optimized for the requirements of a given application. As a result, the virtualization platform should not make any assumptions on the type of software running inside a VM.

- *Shared infrastructure:* Due to the shared nature of a virtualized infrastructure, the achievable performance of one application can be severely impacted by possible resource contention or interference from the co-hosted applications, a problem referred to as *noisy neighbors*. The data center owner may over-commit the physical resources resulting in additional wait times delaying application processing. A model needs to capture contention effects in virtualized data centers.

- *Frequent reconfigurations:* The deployment and configuration of applications may change frequently due to automatic or manual reconfigurations (e.g., deployment of new VMs, or migration of existing ones). As a result, the model needs to be continuously updated to reflect the current system configuration.

In consequence, we require sufficiently fine-grained models providing abstractions from the heterogeneous software stacks while describing the layered application architectures with sufficient details, including the performance influences of the shared infrastructure. As the creation and maintenance of such models can be time-consuming and requires deep knowledge of the performance behavior of a system, we cannot expect system administrators to provide them in advance. A *model-based* resource management mechanism in a virtualized data center needs to be able to learn models of the system automatically. Then it can use them for reasoning purposes in order to improve resource allocation decisions with regards to higher-level goals by proactively determining the effects of changes on the application performance.

1.3 State-of-the-Art

Current approaches in industry for automated performance and resource management in virtualized data centers follow a rule-based approach. A system administrator manually defines custom triggers that fire when a metric reaches a certain threshold (e.g., high resource utilization or load imbalance) and execute certain reconfiguration actions. Examples for such rule-based approaches are the *Amazon EC2 Auto Scaling* feature or the *VMware Distributed Resource Scheduler (DRS)* (Gulati et al., 2012). However, application-level performance metrics, such as response time, normally exhibit a highly non-linear behavior on system load. Therefore, it is not possible to determine general thresholds of when triggers should be fired, given that the appropriate triggering points are typically highly dependent on the architecture of the hosted services and their usage profiles, which can change frequently during operation. Furthermore, in case of contention at the physical resource layer, the performance of an individual application may be significantly influenced by applications running in other co-located VMs sharing the physical infrastructure. Thus, to be effective, triggers must also take into account the interactions between applications and workloads at the physical resource layer. The complexity of such interactions and the inability to predict how changes in application usage profiles propagate through the layers of the system architecture down to the physical resource layer render conventional rule-based approaches unable to

reliably enforce SLAs in an efficient and proactive fashion (i.e., allocating only as many resources as are actually needed and reconfiguring proactively before any SLA violations have occurred). Furthermore, current approaches used in industry are limited with respect to the possible reconfiguration actions. The focus is on the migration of VMs between physical hosts or the starting and stopping of VM instances. More fine-grained reconfiguration actions (e.g., adding or removing CPUs, or assigning additional memory, are typically not supported).

In order to overcome the limitations of rule-based approaches for resource management, model-based approaches have been developed in the research community over the past decade. A number of approaches take the system as a black box and use machine learning techniques (Tesauro et al., 2007; Yazdanov and Fetzer, 2013, e.g.), or feedback controllers (Kalyvianaki et al., 2014; Padala et al., 2009, e.g.) to determine the relationship between the application performance and the resource allocation. While machine learning techniques (e.g., reinforcement learning) can capture complex functions of arbitrary shapes, they typically require long training periods to do so. Given that applications in virtualized data centers are subject to time-varying workloads and dynamic changes to the system configuration, the time to learn and update a model is severely constrained. Feedback controllers are good at keeping a system stable around a target operating point, however, they require sufficiently fine-grained adaptation mechanisms. Most control knobs in virtualized systems have a discrete scale and are therefore not well suited for feedback controllers.

Stochastic performance models, such as, Queueing Networks (QNs) provide a powerful framework enabling a more fine-grained description of the performance behavior of an application including an explicit modeling of the workload. Several authors (Gandhi et al., 2011; Jung et al., 2010; Urgaonkar et al., 2008; Villela et al., 2004) propose resource management approaches based on Queueing Network (QN) models. These approaches expect pre-built performance models and, at most, update certain model parameters based on empirical observations. However, system administrators in virtualized data centers typically do not have the skills and time to manually create stochastic performance models of hosted applications. In this thesis, we built upon approaches for off-line extraction of performance models for architecture evaluation at design-time and integrate them into state-of-the-art virtualization platforms for continuous online model learning at system run-time. The automatically extracted performance models then serve as a basis for fine-grained resource management in virtualized data centers.

1.4 Approach and Contributions

In this thesis, we follow a *self-aware* approach to resource management in virtualized data-centers. In this context, self-aware means that the resource management approach is based on models describing the system with all its performance-relevant factors. The models are learned automatically and serve as a basis for advanced reasoning algorithms designed to adapt the resource allocations of applications in a data center according to higher-level goals regarding application performance and availability. Model learning and reasoning activities are performed continuously during system run-time while applications serve their production workloads.

Our approach is based on descriptive architecture-level performance models which allow for a fine-grained and explicit representation of the system architecture and its major performance-influencing factors. These models enable an integrated and consistent representation of a system, while supporting different mathematical solution algorithms for reasoning. Recent work (Huber et al., 2017) shows the benefits of using such models for online resource management enabling flexible reasoning techniques that automatically balance between prediction accuracy and time to result depending on the type of question to be answered.

1.4.1 Contributions

A Reference Architecture for Online Model Learning in Virtualized Systems

Many complex performance effects and influences in virtualized data centers are only observable during system operation when the system is running in the real production environment under real production workloads, as opposed to running in a controlled testing environment with artificial workloads or synthetic benchmarks. Our goal is to defer as much as possible of the model extraction to system run-time in order to deal with the challenge of capturing the highly non-linear and multidimensional performance influences and interactions between the system components and layers. We propose a novel agent-based reference architecture that enables the online model learning in virtualized data centers. The reference architecture provides a framework for the integration of different static and dynamic analysis techniques into state-of-the-art virtualization platforms in order to extract performance models of application and platform layers at system run-time. We make the following contributions as part of our reference architecture:

- We describe an extension to the definition of conventional Virtual Appliances (VAs) defining additional interfaces for the automatic extraction of models describing the performance behavior of the contained platform and application layers. The term VA refers to a set of pre-packaged VM images containing a software stack designed to run on a virtualization platform. Our extension enables the integration of model extraction logic into a VA in the form of agents that dynamically create and maintain sub-models of platform and application layers (so-called *model skeletons*) at system run-time. These agents may be specifically tailored to the software stack in a VA exploiting technology-specific knowledge during model extraction.

- We introduce additional components within virtualization platforms enabling the collaboration of multiple agents within different VAs in order to extract end-to-end models of a virtualized data center. We develop a performance model repository in which the model skeletons created by different model extraction agents are composed to a single, comprehensive performance model of the complete system covering all its application, platform and infrastructure layers. We describe an algorithm for composing multiple, independently created model skeletons into a single end-to-end architecture-level performance model.

The focus of our reference architecture lies on the integration of existing static and dynamic analysis techniques into an end-to-end model extraction process. Existing end-to-end approaches to online performance model extraction (e.g., Brosig et al., 2011; Brunnert et al., 2013) assume a specific technology stack, such as a Java Enterprise Edition (Java EE) server. Our reference architecture enables the integration of single extraction techniques (e.g., resource demand estimation, or control flow characterization) as well as such end-to-end approaches. It also supports the composition of sub-models representing applications running on heterogeneous technology stacks.

We have published the initial idea of a reference architecture for online performance model extraction in Spinner et al. (2013). In Spinner et al. (2016), we present a preliminary version of the elaborated specification of our reference architecture. The work on the reference architecture was partially supported by the German Research Foundation (DFG) under grant No. KO 3445/11-1.

An Online Method for the Statistical Estimation of Resource Demands

For a given request processed by an application, the resource amount consumed for a specified resource within the system (e.g., CPU or I/O device),

referred to as resource demand, is the total average time the resource is busy processing the request. Resource demands are a key parameter of stochastic performance models (e.g., QNs) as well as architecture-level ones. In order to extract complete performance models, we need to quantify the resource demand values in an online manner. However, existing systems usually do not provide the monitoring capabilities to measure the resource demands directly. Therefore, statistical techniques are required to estimate the resource demands from indirect measurements.

Many approaches to resource demand estimation using different statistical techniques have been proposed in the literature in the last decades. However, for performance engineers it is difficult to decide which one to use for a given system. Our goal is to provide guidelines helping to select the optimal approach and automate the selection process for use in autonomic systems. We make the following contributions:

- We provide a systematization of the state-of-the-art of statistical resource demand estimation. We categorize existing estimation approaches according to their required input parameters, their provided output metrics, and their measures to improve their robustness to anomalies in the empirical measurement data.

- We compare different approaches to resource demand estimation and evaluate their sensitivity to different factors (sampling interval, number of samples, number of workload classes, load level, collinear workload classes, background jobs, and delayed processing) on the estimation accuracy.

- We propose a method to resource demand estimation based on multiple statistical techniques combined with a feedback loop to improve the accuracy of the estimation iteratively by selecting suitable statistical techniques to be applied. We use a cross-validation scheme with an error metric based on the deviation between the observed response times and utilization, on the one hand, and the respective predicted metrics using the resource demand estimates, on the other hand.

- We provide the first publicly available library for resource demand estimation, called LibReDE, which can automatically choose between multiple statistical estimation techniques. It provides ready-to-use implementations of eight approaches to resource demand estimation and realizes our proposed method for resource demand estimation.

The systematization of the state-of-the-art and the experimental comparison have been published in Spinner et al. (2015a). The tool LibReDE has been described in Spinner et al. (2014a). Furthermore, we used LibReDE in collaborations together with the Fortiss research institute (Willnecker et al., 2015) and SAP (Krebs et al., 2014a). The development of LibReDE was partially supported by a VMware Academic Research Award and a Google Faculty Research Award.

Model-based Controllers for Autonomic Vertical Scaling of Virtualized Applications

Many existing model-based approaches to resource management in virtualized data centers are focused on horizontal scaling of applications (i.e., adding or removing replicated VM instances) to react to workload changes. However, starting new VM instances is a relatively expensive operation causing additional overheads and slowing down the reaction to workload changes. As an alternative, state-of-the-art virtualization platforms also provide mechanisms to hot-add virtual CPUs or memory to VMs without requiring a reboot of the guest operating system. These hot-add mechanisms promise a fast and inexpensive way to implement elasticity based on vertical scaling of virtualized applications.

Based on our contributions for online performance model extraction and resource demand estimation, we propose autonomic controllers for vertical scaling of virtualized applications at runtime to ensure that they meet their SLO regarding performance and availability:

- We design a *model-adaptive controller* that automatically determines and allocates the number of CPUs required by individual VMs of a virtualized application in order to fulfill the application SLO. The controller adapts to workload changes on a short-term basis (i.e., seconds to a few minutes). The controller uses a layered *performance model* based on queueing theory that describes the non-trivial relationship between the application performance and its resource allocation. The performance model explicitly captures scheduling delays in the hypervisor due to noisy neighbors. We describe a *learning-based approach* to automatically estimate this model at runtime based on available monitoring data including aggregate application performance and resource usage statistics. Given that the model is continuously updated in short time intervals (up to every 5 minutes), it can quickly capture changes in the workload or the configuration of the application.

- We design a *proactive controller* based on time series analysis techniques to automatically schedule costly memory reconfigurations on a mid- and long-term scale during phases of low load in order to ensure application SLOs regarding performance and availability. Our controller uses state-of-the-art time series *forecasting techniques* to predict seasonal patterns and trends in the incoming application workload for up to a complete day in advance. We incorporate additional meta-knowledge (e.g., calendar information) into the forecast models to better capture the seasonality patterns in the workloads. Automatic reconfigurations may include steps to adapt the application configuration (e.g., garbage collection settings) in order to fully utilize the additional main memory. It allows to schedule a complete restart of the application, if necessary, to be performed during a maintenance window.

Our short-term controller for vertical CPU scaling differs from related work (Dawoud et al., 2012; Shen et al., 2011; Yazdanov and Fetzer, 2012, 2013) in the following aspects: (a) it uses a layered queueing model to decide when to hot-add or -remove CPUs to ensure application SLOs, (b) it automatically estimates per-request resource demands, which are inputs to the performance model, and (c) it uses low-level scheduling statistics from the hypervisor to explicitly capture the effects of physical resource contention.

Our mid- and long-term controller for vertical memory scaling differs from related work (Kumar et al., 2009c; Lu et al., 2014; Shanmuganathan et al., 2013; Shen et al., 2011) in the following aspects: (a) it leverages memory hot-add mechanisms of VMware ESX, (b) it enables application reconfigurations necessary to exploit the additional memory resources, and (c) it uses mid-term workload forecasts (e.g., one day horizon) to automatically schedule reconfigurations during a pre-defined maintenance window.

The short-term controller for vertical CPU scaling has been published in Spinner et al. (2014b). The mid- and long-term controller for vertical memory scaling has been first described in Spinner et al. (2015b). The work on the proposed controllers was supported by a VMware Academic Research Grant.

1.4.2 Evaluation

The contributions are evaluated with regard to *(a)* their degree of automation, *(b)* their applicability to state-of-the-art software systems, *(c)* their effectiveness in improving the elasticity of virtualized applications. We provide proof-of-concept implementations of our approach for widely-used, state-of-the-art practical software systems, such as the VMware vSphere virtualization platform,

the Wildfly (former JBoss) Java EE application server and the Zimbra collaboration server. We demonstrate the feasibility of integrating model-learning techniques into these systems to enable the automatic creation and maintenance of fine-grained performance models. Based on these proof-of concept implementations we performed two case studies in collaboration with our industrial partners VMware and Google.

The first case study is based on the Wildfly application server using the industry-standard SPECjEnterprise2010 full system benchmark as a workload. We examine the degree of automation achieved for our approach to online model learning in a complex distributed architecture. We show that by integrating existing techniques for model extraction into our reference architecture, we are able to extract over 97% of all model elements automatically at system runtime. The resulting performance model provides a high prediction accuracy for a wide range of workload intensities with regards to resource utilization – absolute errors below 4% – and response time – relative errors below 21%. Furthermore, we demonstrate that our method to resource demand estimation scales well to large problem sizes (over 80 resource demands) and it is able to automatically choose the best estimation approach based on cross-validation results.

In the second case study, we demonstrate the effectiveness and efficiency of our proposed autonomic controllers for vertical scaling of virtualized applications. We consider the vertical scaling of CPU and memory resources under realistic workloads using the Zimbra collaboration server. We validate our approach to modeling CPU contention at the hypervisor level. We demonstrate that our model-adaptive controller can reduce the amount of CPU resources by up to 23% compared to a rule-based controller while ensuring full compliance with application SLOs. In the same experiment, it is also able to reduce the number of reconfigurations by up to 95%. Furthermore, using workload forecasting techniques we were able to schedule expensive reconfigurations (e.g., changes to the memory size) during phases of load load and thus were able to reduce their impact on application availability by over 80% while significantly improving application performance compared to a reactive controller.

1.5 Thesis Outline

The thesis is organized as described in the following.

- Chapter 2 introduces the theoretical and technical foundations of our approach. We define the term self-aware computing as it is used in the

context of our thesis. Then, we introduce different modeling formalisms for descriptive knowledge representation and for reasoning on the performance of a system. Furthermore, we describe the mathematical background of different statistical estimation techniques. Finally, we present the core concepts of server virtualization.

- Chapter 3 surveys the related work on autonomic resource management in virtualized data centers and on the automatic extraction of performance models.

- Chapter 4 presents our agent-based reference architecture for online model learning in virtualized systems. We propose extensions for state-of-the-art virtualization platforms to support deep integration of model extraction capabilities. We describe how agents that take on different roles collaborate at system run-time in order to extract models of the system. We propose an algorithm to compose sub-models from different agents into an end-to-end performance model.

- Chapter 5 focuses on the online estimation of resource demands. It systematizes the state-of-the-art on resource demand estimation and presents a comparison of different estimation approaches evaluating the influence of various factors on the estimation accuracy. Furthermore, we introduce a new method for resource demand estimation based on multiple statistical techniques in order to improve the robustness of resource demand estimation.

- Chapter 6 describes two autonomic controllers for the vertical scaling of virtualized applications. The first one is aimed at short-term reconfigurations of resources that are quickly adaptable (e.g., CPUs) without overheads. It uses a layered performance model that explicitly captures contention effects due to co-located applications. The second controller supports mid- and long-term reconfigurations of resources with high adaptation costs (e.g., memory) by proactively planning such reconfigurations in advance employing workload forecasting techniques.

- Chapter 7 contains the validation of our self-aware approach to resource management. We integrate our techniques for model extraction into state-of-the-art platforms and evaluate their accuracy and efficiency in case studies with real-world applications.

- Chapter 8 concludes this thesis and provides an overview of future work.

Part I

Foundations and Related Work

Chapter 2

Foundations of Self-Aware Computing

2.1 Self-Aware Computing

Self-aware computing is a research direction within the broader research field of autonomic systems. In 2001, IBM introduced the term *autonomic computing* envisioning computer systems "that can manage themselves given high-level objectives from administrators" (Kephart and Chess, 2003). They see autonomic computing systems as the only option to cope with the increasing size and complexity of today's computer systems (Kephart and Chess, 2003). An autonomic system consists of a set of agents with self-management capabilities, such as self-configuration, self-optimization, self-healing, and self-protection. Not only the self-management capabilities of the individual agent, but also the collaboration and interaction between agents are crucial for the fulfillment of the higher-level goals.

The autonomic computing initiative inspired numerous researchers around the world over the last 15 years. Kephart (2011) observes that most work uses machine-learning and feedback control techniques for self-optimization at the agent-level lacking support for the fulfillment of high-level goals at the system-level. In order to address these deficiencies of existing autonomic systems, leading researchers in the field introduced the idea of *self-aware computing systems* (Kounev et al., 2015). They define the term self-aware computing as following:

Definition 2.1 (Self-Aware Computing)**.** "Self-aware computing systems are computing systems that:

1. learn models capturing knowledge about themselves and their environment (such as their structure, design, state, possible actions, and run-time behavior) on an ongoing basis and
2. reason using the models (for example predict, analyze, consider, plan) enabling them to act based on their knowledge and reasoning (for example explore, explain, report, suggest, self-adapt, or impact their environment)

in accordance with higher-level goals, which may also be subject to change." (Kounev et al., 2015)

This definition highlights the central role of models in self-aware computer systems capturing knowledge about the system and its environment. It does allow for any type of model, requiring only the characteristics Stachowiak (1973) describes in his general model theory:

Definition 2.2 (Model)**.** Every model fulfills the following three main characteristics:

- *Mapping:* a model is always a representation of some entity (real-world, or a model itself).
- *Reduction:* a model always abstracts from the underlying entity.
- *Pragmatism:* a model is built for a certain purpose.

Models in self-aware computing systems may fulfill different purposes during learning and reasoning. Different types of models may be used to represent knowledge in a manner that suits its purpose. In general, the following three main types of models may be distinguished (Kounev et al., 2015):

- *Descriptive* models describe the actual state of a system at a given point in time.
- *Prescriptive* models represent planned states of a system in the future. The system needs to adapt to reach such a state.
- *Predictive* models enable the prediction of the impact of changes in the system or its environment on the system behavior. Predictive models are required for a system to become *proactive*, i.e., to react to changes in the environment in advance before violating high-level goals.

Figure 2.1 shows the conceptual model-based learning, reasoning, and acting loop (LRA-M) consisting of the main activities in a self-ware computing system and how they interact with the model-based knowledge. The system gets the high-level goals from a user and empirical observations from sensors as input. Based on the empirical observations it continuously learns models on the system behavior and structure, its environment and its goals. Using previously learned models and empirical observations, the system also executes reasoning activities to check the fulfillment of its goals. The reasoning process may trigger certain act processes in order to adapt the system in accordance with

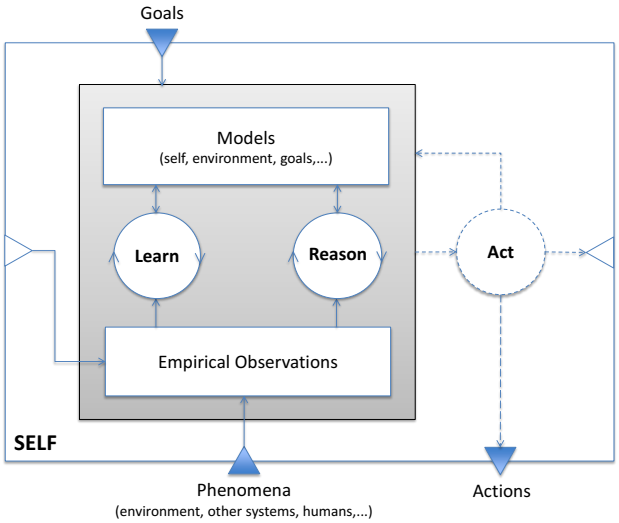


Figure 2.1: LRA-M loop for self-aware computing systems (excerpt from Kounev et al., 2017b).

the system goals. The act process is optional as self-adaptation capabilities are not a requirement for self-aware computing systems. A self-aware computing system may also assist a human in adapting a system.

The concept of self-aware computing is independent of a concrete domain. It may be realized for different types of systems (e.g., data-center performance and resource management, or cyber-physical systems). In the following sections, we describe existing techniques which are useful when building self-aware resource management systems for virtualized data-centers.

2.2 Model-based Knowledge Representation

In Section 2.1, we use the term "model" in a rather generic way ranging from machine-learning and statistical models (e.g., regression models, or artificial neural networks) over stochastic models (e.g., Markov or Queueing models) to domain-specific modeling languages (e.g., meta-models or ontologies). While the first two types of models provide a well-understood and mathematical rigorous foundation for analyzing and predicting the behavior of computer systems, they come with certain limitations:

- The models are domain-agnostic and therefore only have very limited descriptive capabilities. Additional semantic information on what a model

element represents in the actual system (e.g., a CPU, a component, or a request) cannot be explicitly included in the model.

- The models do not support a separation of concerns. The complete model is typically described in a single model. When changing a certain aspect, one needs to understand the complete model in order to avoid unwanted side-effects.

- The models support only certain types of analyses well. In order to support a broad set of analyses, a self-aware computing system may need to maintain a set of different models with potentially overlapping parts. This introduces additional complexity to its learning and reasoning processes.

Kounev et al. (2016) proposes an approach based on the principles of model-driven engineering (Schmidt, 2006) using a domain-specific language – called Descartes Modeling Language (DML) – for online performance and resource management in data centers. They specify a meta-model comprising all aspects of a data-center relevant for performance and resource management and describe model-to-model transformations to different types of prediction models in order to enable reasoning. In this thesis, we use DML as a basis to represent the knowledge in a system for self-aware resource management. We first give a short introduction to model-driven engineering in general in Section 2.2.1 and then provide an overview of DML in Section 2.2.2.

2.2.1 Model-Driven Engineering

Model-Driven Engineering (MDE), or Model-Driven Deveoplment (MDD), is a software engineering discipline based on domain-specific languages and model transformations in order to describe domain concepts effectively (Schmidt, 2006). In contrast to general programming languages (e.g., C++ or Java), domain-specific languages are specifically tailored to describe problems of a certain domain (e.g., financial services or industrial production systems). Domain-specific languages are focused on modeling the variable parts of a domain enabling more concise and abstract descriptions (Voelter et al., 2013).

Meta-models are a formal definition of the abstract syntax of a domain-specific language (i.e., the model entities and their relationships). Meta-metamodels are available to describe meta-models in a machine-processable way. The Object Management Group (OMG) standardization body specified such a meta-metamodel – called Meta-Object Facility (MOF) – for its Unified Modeling Language (UML) standard (OMG, 2015). OMG distinguishes between *complete*

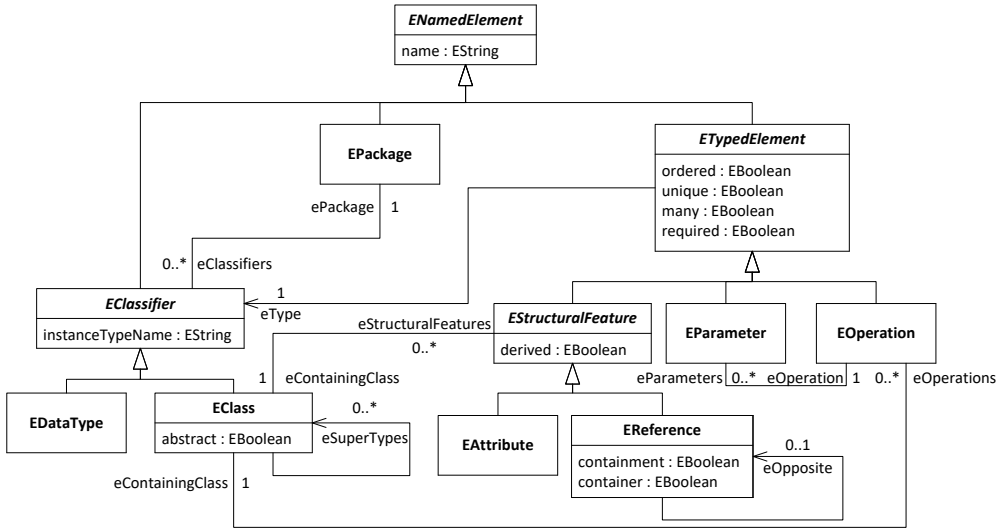


Figure 2.2: Ecore meta-metamodel.

and *essential* MOF. Essential MOF provides a simplified meta-metamodel and is most commonly used in practice.

The Eclipse Modeling Framework (EMF) (Steinberg et al., 2009) provides the Ecore meta-metamodel, which is aligned with the essential MOF standard. Figure 2.2 shows the main classes of the Ecore meta-metamodel. Ecore is self-describing, i.e., the meta-model can be used to describe its own abstract syntax. The root element `EPackage` in an EMF meta-model groups a set of `EClassifier` objects which belong semantically together. An `EClassifier` can be either an `EDataType`, representing a primitive value type, or a structured `EClass`.

Any EMF model can be considered as a directed graph. The nodes are objects conforming to meta-model classes which are instances of `EClass`. The edges are settings of `EReference` features defined in a meta-model class. A subset of the object graph of an EMF model forms a containment tree (i.e., references marked with `containment` or `container`). Each object has exactly one parent object (except for the model root).

In addition, each meta-model class may contain a set of `EAttribute` containing one or several values of an `EDataType`. Operations can be used to describe dynamic behavior within meta-models. Operations are not used in this thesis as we use meta-models as a basis for structured knowledge representation.

2.2.2 Descartes Modeling Language (DML)

DML is a modeling language for online performance and resource management in data centers (Kounev et al., 2016). It is designed as a descriptive meta-model. A set of model-to-model transformations exist to automatically derive prediction models (e.g., QNs or Queueing Petri Networks (QPNs)) from a DML instance. Depending on the type of analysis to be performed, a transformation can be chosen from this set. DML consists of a set of EMF-based meta-models. Figure 2.3 gives an overview of these meta-models the DML.

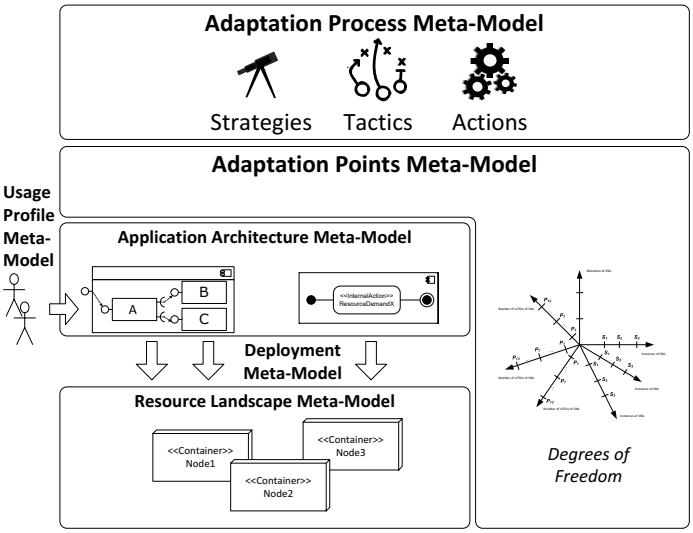


Figure 2.3: DML meta-model overview (excerpt from Kounev et al., 2014).

The *Resource Landscape* meta-model describes the infrastructure layers within a data-center (e.g., hypervisor, operating system, or middleware layers). Each layer may provide certain resources (e.g., physical or virtual CPUs, hard disks, or software resources). The *Application Architecture* meta-model provides a component-based description of the structure and behavior of applications (interfaces and software components, control flow between components, and performance-relevant behavior of the component implementation). The *Deployment* meta-model describes the mapping of application components to the resource landscape. The *Usage Profile* meta-model describes the external workload of an application (i.e., the order and intensity of requests arriving at the application). These four meta-models enable performance predictions under varying workload and configuration settings.

DML provides additional meta-models to describe adaptation processes within a system. The *Adaptation Process* meta-model is a decorator meta-model and describes the degrees of freedom of a system (i.e., the parts of the application architecture and the infrastructure layers that can be adapted at system run-time). The *Adaptation Process* meta-model provides a domain-specific language to description adaptation processes. These processes are based on the adaptation points and describe how the system is adapted at run-time in order to fulfill high-level goals. The following descriptions are focused on the meta-models used for performance predictions as they are the foundation of this thesis. We refer the reader to Huber (2014) for more details on the adaptation points and adaptation process meta-models.

Resource Landscape. The root element of the resource landscape meta-model is a `DistributedDataCenter`, which contains a set of `DataCenter` elements. A `DataCenter` describes the physical infrastructure of a data center containing sets of `ComputeNode` and `StorageNode` elements, which are both subclasses of `Container`. A `Container` represents a logical system entity that can execute software components of applications. Software infrastructure layers (e.g., hypervisor, operating system, or middleware system) can have a significant impact on the performance behavior of a system, for instance, see the evaluation of virtualization overheads in Huber et al. (2011). In order to support the explicit modeling of layered software infrastructures, DML supports the nesting of `Container` elements. For instance in a virtualized data-center, you typically have a hypervisor that hosts a set of VMs, each running its own guest Operating System (OS) and additional middleware services. `RuntimeEnvironment` elements, which are also a subclass of `Container`, can be used to describe each of these software infrastructure layers.

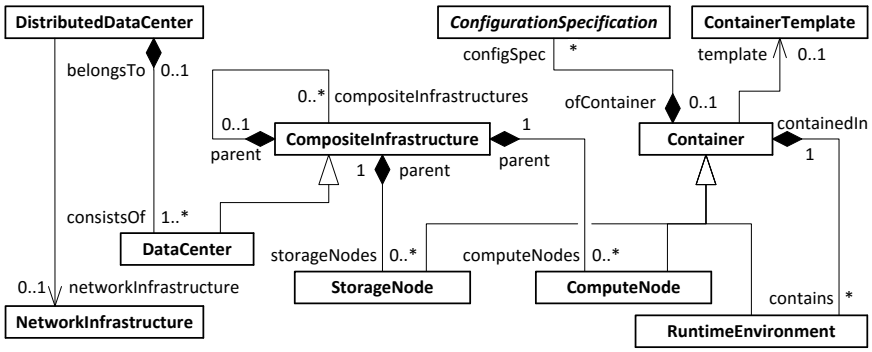


Figure 2.4: DML resource landscape (excerpt from Kounev et al., 2016).

Each Container references one or several ConfigurationSpecification elements that can be of one of the following subclasses:

- A ProcessingResourceSpecification describes an active resource offered by a container (e.g., physical and virtual CPUs, or hard disks). An active resource is characterized by a scheduling policy of the active resource, as well as a processing speed and level of parallism.
- A PassiveResourceSpecification represents software and hardware resources with limited capacity (e.g., main memory, thread or connection pools). A passive resource is characterized by its maximum capacity.
- A CustomConfigurationSpecification allows to include additional models describing the impact of the configuration on the performance of a system. For instance, Huber et al. (2011) describe an overhead model for the virtualization layer that may be referenced here.

In virtualized data centers, system entities are often replicated multiple times with the same configuration (e.g., to support load-balancing and failover). In order to reduce modeling efforts and enable reuse, it is possible to use the ContainerTemplate element to specify ConfigurationSpecification elements that are shared between multiple Container elements. These templates are stored in a separate ContainerRepository model.

A separate meta-model exists to describe the network infrastructure in a data-center, called Descartes Network Infrastructure (DNI) (Rygielski and Kounev, 2014). A complete DNI model may be referenced from the root DistributedDataCenter.

Application Architecture. The description of the application architecture is based on the principles of component-based software engineering. An application consists of a set of component definitions stored in a Repository root element. Figure 2.5 gives an overview of the main classes of the repository meta-model. We distinguish between different types of components derived from the abstract InterfaceProvidingRequiringEntity class. Each component may reference multiple InterfaceProvidingRole and InterfaceRequiringRole elements describing the component services provided or required by a component. Each role references an Interface definition in the Repository. An Interface contains a set of Signature elements representing individual operations. A component service is an implementation of a Signature at an InterfaceProvidingRole. DML distinguishes between the following three types of components:

- A `BasicComponent` is an atomic component containing direct implementations for each provided component service.
- A `CompositeComponent` may consist of several other components (composite or basic), which are always deployed together on the same container.
- A `SubSystem` may also consist of several other components (composite, basic, or subsystem). In contrast to `CompositeComponent`, the individual components may be deployed on separate containers.

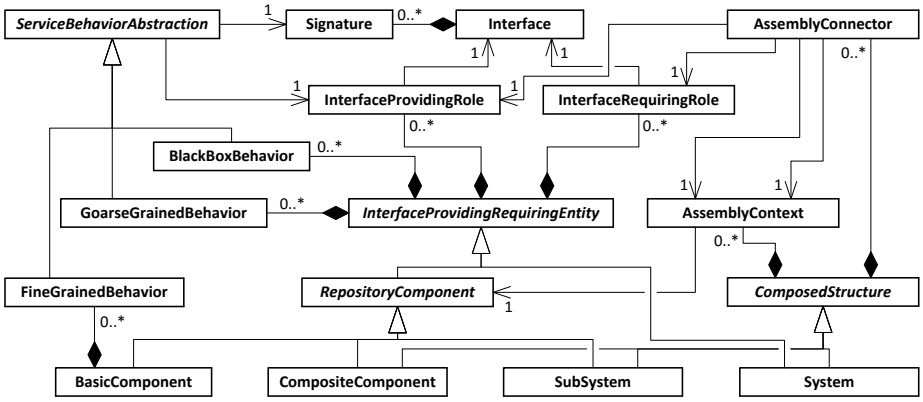


Figure 2.5: DML application architecture (simplified).

The `ServiceBehaviorAbstraction` element describes the performance behavior of a component service. It is possible to describe the service behavior at the following abstraction levels:

- A `BlackBoxBehavior` contains a `ResponseTime` element describing the response time behavior of the service as an arbitrary function.
- A `CoarseGrainedBehavior` specifies how many resources a component service consumes in total (`ResourceDemand`) and which external component services are called with what frequency (`ExternalCallFrequency`). The sequence of external calls and resource demands is not specified.
- A `FineGrainedBehavior` defines a sequence of performance-relevant actions (`AbstractAction`). Control flow actions (such as `BranchAction`, `LoopAction`, and `ForkAction`) are provided to describe different sequences

of external calls (represented by ExternalCallAction) and resource-demanding internal computations (represented by InternalAction). Furthermore, AcquireAction and ReleaseAction elements cn be used to describe synchronization behaviors (e.g., critical sections, or semaphores). Only basic components may contain FineGrainedBehavior descriptions.

It is possible to specify multiple service behaviors for the same component service. When predicting the performance of a system, one of the service behaviors is chosen depending on the requirements on prediction accuracy and solution time.

The AssemblyContext and the AssemblyConnector elements describe the control flow between components in a system. An AssemblyContext is the instantiation of a component defined in the Repository in a system. The same component may be instantiated multiple times, each instantiation is described by a separate AssemblyContext. The AssemblyConnector element describes the inter-component control flow connecting an InterfaceRequiringRole of a component in one AssemblyContext with an InterfaceProvidingRole of a component in another AssemblyContext.

Deployment. The resource landscape model and the application architecture are linked by the deployment model. It consists of a list of DeploymentContext elements, each mapping an AssemblyContext in the application architecture to a Container element in the resource landscape.

Usage Profile. A usage profile consists of one or several UsageScenario elements. Each UsageScenario contains a WorkloadType and a ScenarioBehavior element. The former specifies whether the workload is open or closed including a specification of the load intensity. The latter describes a sequence of UserAction elements. A UserAction element can be a SystemCallUserAction describing a single invocation of a system service, a DelayUserAction to represent user think times, or composite constructs (e.g., LoopUserAction, or BranchUserAction).

Model Variables. Service behavior descriptions contain random variables (called model variable in DML) that need to be characterized before a model can be solved. Examples are resource demands, response times (in case of black box behaviors), or control flow variables, such as branching or loop iteration counts. These parameters are represented by ModelVariable elements. Each model variable in a DML model may have either an EXPLICIT or an EMPIRICAL characterization. The former means, that the model variable comes with an explicit

description of its probability distribution function. The latter expresses that the probability distribution needs to be characterized based on empirical data observed at system run-time.

Parameter dependencies. The shape of a probability distribution function of a model variable may depend on the value of input parameters of the system. For instance, the resource demands in an online shop may depend on the number of items a user has in its shopping cart. Such parameter dependencies can be modeled explicitly in DML. A parameter dependency connects one or multiple input parameters defined in a component interface with one or multiple model variables.

2.3 Statistical Techniques for Model Parameterization

In this section, we give an overview of statistical techniques that can be used for the characterization of model parameters (e.g., resource demands) from empirical observations at a system.

2.3.1 Regression Analysis

Given a set of independent variables $x_1 \dots x_k$ and a dependent variable y, linear regression tries to describe the relationship between the dependent variable and the independent variables with the linear model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k + \epsilon. \tag{2.1}$$

In regression analysis, y is known as *response variable* and x_j with $1 \leq j \leq k$ as *control variables*. The goal is to determine the parameters β_j with $0 \leq j \leq k$ in such a way that the residuals ϵ are minimized regarding a specific measure. Examples for such measures are the sum of squared residuals used in Least Squares (LSQ) regression or the sum of absolute differences used for Least Absolute Differences (LAD) regression. To be able to determine a unique solution for the parameters β_j, at least n sets of known values $(y, x_1 \dots x_k)$ are required, where $n > k$. The above linear model can be written in matrix notation as (Chatterjee and Price, 1995, p. 96ff)

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{2.2}$$

with following matrices

$$\mathbf{X} = \begin{pmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,k} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,k} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,k} \end{pmatrix}, \mathbf{Y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix} \text{ and } \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \end{pmatrix}.$$

$\mathbf{X}$ is called control matrix and $\mathbf{Y}$ is the response vector. We assume that the vector of error residuals $\boldsymbol{\epsilon}$ is independent and identically distributed with mean $E[\boldsymbol{\epsilon}] = 0$ and a constant variance. Then we can conclude that $E[\mathbf{Y}] = \mathbf{X}\boldsymbol{\beta}$ (Chatterjee and Price, 1995, p. 97). The parameter vector $\boldsymbol{\beta}$ needs to be estimated.

LSQ regression estimates the parameter vector $\boldsymbol{\beta}$ by minimizing the sum of squared residuals. Hence, the following expression needs to be minimized

$$\boldsymbol{\epsilon}^T\boldsymbol{\epsilon} = (\mathbf{Y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}). \tag{2.3}$$

The value $\hat{\boldsymbol{\beta}}$ that minimizes the previous expression can be calculated with following formula (Chatterjee and Price, 1995, p. 97)

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}. \tag{2.4}$$

2.3.2 Kalman Filter

Statistical filtering is about the estimation of a hidden state of a dynamic system from known system inputs and incomplete and noisy measurements (Kumar et al., 2009a). In this context, the term *state* is defined as follows:

> "The states of a system are those variables that provide a complete representation of the internal condition or status at a given instant of time." (Simon, 2006, p. xxi)

The term *dynamic system* implies that the state of the system changes over time. Different statistical filtering methods have been proposed. We describe the Kalman filter here in more detail because it is often used to estimate resource demands.

Generally speaking, we can distinguish between discrete-time and continuous-time systems. Subsequently, we will focus on discrete-time Kalman filters. The notation used is based on the one used by Kumar et al. (2009a) and Simon (2006).

The system state $\mathbf{x}$ is a vector containing the variables that describe the internal state of a system. These variables cannot be directly observed at a

system. The Kalman filter estimates the vector $\mathbf{x}$ from a series of measurements $\mathbf{z}$. The system is described by two equations. The first equation describes how the system state evolves over time according to (Simon, 2006)

$$\mathbf{x}_k = \mathbf{F}_{k-1}\mathbf{x}_{k-1} + \mathbf{G}_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_{k-1}. \tag{2.5}$$

The time advances in discrete steps, which are denoted by index k. $\mathbf{x}_k$ is the system state at time step k, which is calculated from the previous system state $\mathbf{x}_{k-1}$ and the control vector $\mathbf{u}_{k-1}$ containing the inputs of the system. The matrices $\mathbf{F}$ and $\mathbf{G}$ are called state transition model and control-input model. The process noise $\mathbf{w}_{k-1}$ is assumed to be normally distributed with zero mean and covariance $\mathbf{Q}_k$.

The second equation describes the relationship between the system state $\mathbf{x}_k$ and the measurements $\mathbf{z}_k$ at time step k according to (Simon, 2006)

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k. \tag{2.6}$$

The matrix $\mathbf{H}_k$ is the observation model, which maps the state space to the observation space. $\mathbf{v}_k$ is the observation noise, which is assumed to be Gaussian white noise with zero mean and covariance $\mathbf{R}_k$.

If the relation $\mathbf{z} = h(\mathbf{x})$ between system state and measurements is non-linear, the Extended Kalman filter (EKF) can be used. The EKF approximates a linear model with the following output sensitivity matrix:

$$\mathbf{H}_k = \left[\frac{\partial h}{\partial x}\right]_{\hat{x}_{k|k-1}}. \tag{2.7}$$

The output sensitivity matrix is set to the Jacobian matrix of $h(\mathbf{x})$. The partial derivatives are evaluated with the current estimates of the system state. The vector $\hat{\mathbf{x}}_{n|m}$ represents the estimated system state $\hat{\mathbf{x}}$ at time step n given measurements $\mathbf{z}_1 \dots \mathbf{z}_m$.

The Kalman filter is a recursive estimator. It starts with an initial state and continuously updates its estimate as new measurements are obtained. At each time step k the calculations only depend on the previous estimate $\hat{\mathbf{x}}_{k-1|k-1}$ and the current measurements vector $\mathbf{z}_k$ (Kumar et al., 2009a). The internal state of the filter is represented by two variables:

- the state estimate $\hat{\mathbf{x}}_{k|k}$,
- and the error covariance matrix $\mathbf{P}_{k|k}$.

The error covariance matrix is a measure for the estimated accuracy of the state estimates (Kumar et al., 2009a). At the beginning the filter is initialized with given values for $\hat{\mathbf{x}}_{0|0}$ and $\mathbf{P}_{0|0}$.

The algorithm that calculates new state estimates consists of two phases (Kumar et al., 2009a): *Predict* and *Update*. In the predict phase a new state estimate $\hat{\mathbf{x}}_{k|k-1}$ is calculated with Equation 2.5. In the update phase the prediction error of $\hat{\mathbf{x}}_{k|k-1}$ is determined according to the current measurements $\mathbf{z}_k$. Then a corrected estimate $\hat{\mathbf{x}}_{k|k}$ is calculated. These two steps are carried out each time a new measurement sample vector gets available.

Assuming a linear relationship between the measurements and the system state, and uncorrelated and normally distributed noise with zero mean, the Kalman filter is an optimal estimator. Since most systems are inherently nonlinear, the EKF provides a linear approximation for cases with slightly nonlinear characteristics.

2.3.3 Mathematical Optimization

Generally speaking, an optimization problem is described by a *cost (objective) function* f with a domain $D \subseteq \mathbb{R}^n$ and a *constraint set* $\Omega \subseteq D$ (Dostl, 2009). The objective function can be either minimized or maximized. In the following, we assume that it should be minimized. Then the optimization problem is also called *minimization problem* and can be solved by finding a value $\bar{x} \in \Omega$, so that

$$f(\bar{x}) \leq f(x),\ x \in \Omega. \tag{2.8}$$

Solutions of a minimization problem are called *(global) minimizers* (Dostl, 2009). In contrast to global minimizers, there are also *local minimizers*. A local minimizer $\bar{x}$ satisfies the condition

$$f(\bar{x}) \leq f(x),\ x \in \Omega,\ \|x - \bar{x}\| \leq \delta \tag{2.9}$$

for $\delta > 0$ (Dostl, 2009).

Optimization problems can be classified into different categories. There are *constrained* and *unconstrained* optimization problems. In the case of unconstrained optimization problems, there are no additional constraints in the constraint set, i.e., $\Omega = D$. If additional equality and inequality constraints are given, we speak of constrained optimization. Depending on the degree of the objective function and the constraints the following types of optimization problems exist:

- *Linear programming* problems have a linear objective function and a set of linear equality and inequality constraints.

- *Quadratic programming* problems are optimization problems with a quadratic objective function and a set of linear equality and inequality constraints.
- *Non-linear programming* problems can have any kind of non-linear objective function and/or non-linear constraints.

Different solution algorithms exist for the different types of optimization problems. Descriptions of possible solution algorithms can be found in Dostl (2009) and Nemhauser (1989).

2.3.4 Maximum Likelihood Estimation

Suppose a collection of independent and identically distributed random variables $Y_1, Y_2, ..., Y_n$ where θ are the parameters of their probability distribution, and corresponding sample path $y_1, y_2, ..., y_n$, then the joint distribution is represented by the probability density function $f(y_1, y_2, ..., y_n|\theta)$. This function is also known as the likelihood $\mathbb{L}(\theta)$ stating the probability of observing a sample path $y_1, y_2, ..., y_n$ for a given θ. The joint distribution can be replaced by the product of the conditional probability of individual samples:

$$\mathbb{L}(\theta) = \prod_{i=1}^{n} f(y_i|\theta). \tag{2.10}$$

An equivalent representation that is often easier to solve is obtained by taking the logarithm of the likelihood function resulting in a sum of logarithms. This is known as the log-likelihood function.

The maximum likelihood estimate $\hat{\theta}$ is then defined as

$$\hat{\theta} = \max_{\theta} \mathbb{L}(\theta). \tag{2.11}$$

In case of complex likelihood functions, optimization algorithms (c.f. Section 2.3.3) are typically used to determine the maximum likelihood estimate.

2.3.5 Bayesian Inference

In the previous section, we described the maximum likelihood estimation method based on the frequentist interpretation of probability. In contrast, Bayesian inference introduces the concept of a prior distribution capturing assumptions and knowledge available before taking observations. Suppose a vector of parameters θ and vector $\mathbf{y}$ with observations, the posterior distribution $f(\theta|\mathbf{y})$ is

$$f(\theta|\mathbf{y}) = \frac{f(\mathbf{y}|\theta)f(\theta)}{f(\mathbf{y})}. \tag{2.12}$$

$f(\mathbf{y}|\theta)$ denotes the likelihood of observing $\mathbf{y}$ for a given θ. $f(\theta)$ is the prior distribution, and $f(\mathbf{y})$ is the marginal likelihood of the observations. Assuming fixed values for the observations $\mathbf{y}$, $f(\mathbf{y})$ can be seen as a normalization constant. The constant can be calculated using the following integral:

$$f(\mathbf{y}) = \int f(\mathbf{y}|\theta) f(\theta) d\theta. \tag{2.13}$$

However, the exact calculation of this integral required to determine the joint posterior distribution is intractable for most practical problems, especially in case of multi-variate posterior distributions. The Metropolis-Hastings algorithm (Hastings, 1970), which is a Markov-chain Monte Carlo (MCMC) algorithm for random sampling, allows us to approximate the posterior distribution without calculating the normalization constant. The algorithm only requires the availability of a function g that is proportional to a desired probability distribution. In case of Bayesian inference, the function g is the numerator in Equation 2.12.

The general idea of MCMC algorithms is to construct a Markov chain with an equilibrium distribution resembling the desired posterior distribution. Samples are generated by a random walk on this Markov chain. A sample is then the state of the Markov chain after a certain number of steps. Gibbs sampling (Geman and Geman, 1984) is a special case of the Metropolis-Hastings algorithm for highly multi-variate distributions. Suppose we want to obtain a sample $X = (x_1, ..., x_n)$ from the posterior distribution, then Gibbs sampling requires the availability of all conditional distributions $f(x_i|x_1, ..., x_{i-1}, x_{i+1}, x_n)$. The conditional distributions may either be calculated exactly, or we rely on other random sampling algorithms for single-dimensional distributions, such as adaptive rejection sampling. In order to determine the sample in step t, a Gibbs sampler iterates over each component of vector $X^{(t)}$ and determines its value by sampling from $f(x_i^{(t)}|x_1^{(t)}, ..., x_{i-1}^{(t)}, x_{i+1}^{(t-1)}, ..., x_n^{(t-1)})$. Given a large set of samples resulting from the Gibbs sampler, we can approximate the expected value of the posterior distribution by averaging over all samples. It should be noted that consecutive samples from a Gibbs sampler are typically auto-correlated. Therefore, only every n-th sample should be included. Furthermore, samples at the beginning should be discarded as long as the underlying Markov chain is not in its equilibrium state.

2.4 Performance Prediction

In contrast to the descriptive meta-models introduced in Section 2.2, we now describe model formalisms and algorithms that are used to create prediction models for a given system. In a self-aware system, prediction models enable to analyze the impact of adaptations on the system performance in advance. The prediction models are typically derived from a descriptive model, e.g., using the model-to-model transformations for DML proposed by Brosig (2014). In this section, we provide foundations of queueing theory used in Chapters 5 and 6 for resource demand estimation and performance prediction.

2.4.1 Queueing Models

Queueing theory is a discipline of stochastic theory and operations research. It provides general methods to analyze the queueing behavior at a service station and has been successfully applied to different domains in the last decades, e.g. to model manufacturing lines or call center operation. When analyzing the performance of a computer system, queueing models are used to describe the scheduling behavior at hardware resources, such as CPUs, hard disks and network devices (Bolch et al., 2006; Harchol-Balter, 2013; Lazowska et al., 1984). We will first introduce the single service center case including fundamental laws to determine performance measures of it. Then, we will describe QN with multiple service centers and discuss the challenges when analyzing such models.

2.4.1.1 Single Service Center

In general terms, a service center consists of a queue and one or several identical servers (see Figure 2.6). Incoming requests (or users, jobs, transactions) are processed at one of these servers. A server is either busy when serving jobs, or free otherwise. On arrival, requests are served immediately if at least one server is currently free. Otherwise, they have to wait in the queue until a server becomes free. After service completion the requests leave the queue.

A number of terms are commonly used when describing the timing behavior of a queue. Requests arrive at the queue at arbitrary points in time. The number of requests per time unit is called *arrival rate* λ. The average time between consecutive requests is called *interarrival time*. Each request requires a certain amount of processing at a server. The service rate μ determines the number of requests that can be processed per time unit at a single server. The mean *service time* is then defined as $S = \frac{1}{\mu}$ and specifies the time a server is

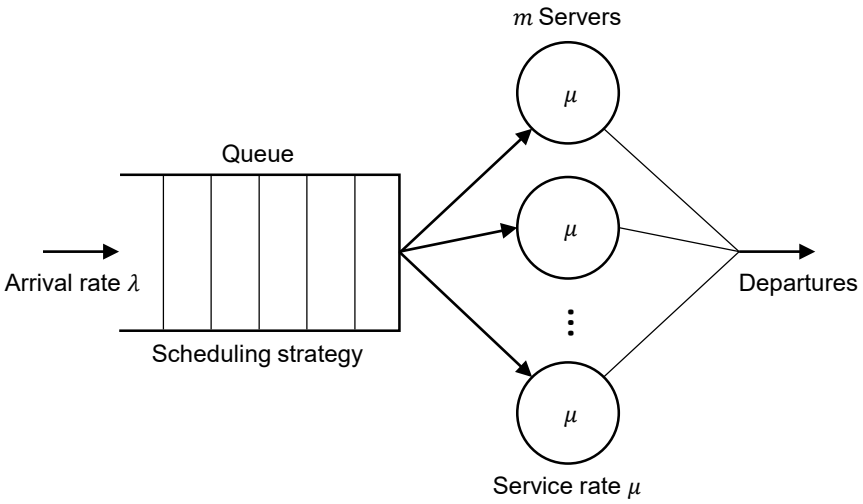


Figure 2.6: Single service center

occupied with processing a request on average. Based on the service time, the service demand is defined as following (Lazowska et al., 1984; Menascé et al., 2004):

Definition 2.3 (Service demand or resource demand)**.** The service demand D is the total average service time of a request over all visits at a queue. It can be defined as

$$D = V \cdot S, \tag{2.14}$$

where V is the mean number of visits and S the average service time of each request. The terms resource demand and service demand can be used interchangeably. In the following we will use the term resource demand.

When one request is completed, the next is selected from the requests in the queue according to a *scheduling strategy*. Typical scheduling strategies are First-Come-First-Serve (FCFS), where jobs are processed in the order of their arrival, Processor-Sharing (PS), where jobs are served concurrently each with an equal share of the total capacity (i.e., round-robin scheduling with infinitesimally small time slices), or Infinite-Server (IS), which represents queues with constant delays.

There is a standard notation for describing a queue, which is called *Kendall's notation* (Kendall, 1953), consisting of a six-tuple *A/S/m/B/K/SD*: A is the arrival process (i.e., the distribution of the interarrival times), S is the service process (i.e., the distribution of the service times), m is the number of servers, B is

the maximum number of jobs in the queue (by default ∞), K is the maximum number of jobs that can arrive at the queue (by default ∞), and SD is the scheduling strategy (by default FCFS). The distribution components are characterized using short-hand symbols for the type of distribution. Commonly used symbols are M for an exponential (Markovian) distribution, D for a deterministic distribution, E_k for an Erlang distribution with parameter k, and G or GI for general (independent) distributions.

In practice, many systems serve requests with different arrival and service characteristics (e.g., the service rate of read and write requests to a database may be different). In theory, it would be possible to use multi-modal distributions for such cases, however, this can complicate the parameterization and solution of queueing models (Harchol-Balter, 2013, Chapter 21). Instead, multi-class queueing models are used, where a *workload class* represents a set of jobs with similar characteristics. Each workload class has its own set of arrival rate and service rate parameters.

For a single queue, we may determine a number of performance measures for a *transient* point in time t or for the *steady state* (i.e., $t \to \infty$). In the following, we are interested in the steady-state solution of a queue. More details on the transient solution of a queue can be found in (Bolch et al., 2006). The performance measures of interest are typically the utilization U_i, the throughput X_c, the queue length Q_i, and the response time $R_{i,c}$. The utilization is the ratio of time a queue is busy serving requests. The throughput is the number of requests leaving the queue in a given time interval. If the maximum number of requests that arrive at a queue is unlimited, the relation $\lambda \leq \mu$ must hold, so that the queue is stable (i.e., a steady state solution exists). The queue length specifies the number of requests waiting for service (excluding those currently in service). The mean response time $R_{i,c}$ of queue i and workload class c is defined in general by the following equation:

$$R_{i,c} = W_{i,c} + D_{i,c} \tag{2.15}$$

$W_{i,c}$ is the mean time a request has to wait in the queue before being served, and $D_{i,c}$ is the mean resource demand of that request. The waiting time $W_{i,c}$ depends on a number of parameters, amongst other the scheduling strategy, and the arrival and service processes.

2.4.1.2 Queueing Network (QN)

A QN consists of at least two service centers which are connected together (Bolch et al., 2006, p. 282). The routing of requests between service centers is specified by a probability matrix. Requests of class c departing from service

station i will move to service station j with probability $P_{c,i,j}$ or leave the QN with probability $P_{c,i,out} = 1 - \sum_j P_{c,i,j}$ (Harchol-Balter, 2013, p. 297). Requests of class c can arrive from outside the QN at service station i with a rate $r_{c,i}$. In case of an *open* QN, requests of all workload classes can enter the network from outside as well as leave it (Bolch et al., 2006, p. 282). A QN is *closed* if the number of requests in the network is constant (i.e., no requests can enter or leave the network) (Bolch et al., 2006, p. 282). If some workload classes are open and others are closed, we speak of a *mixed* QN.

The solution of a QN with K service centers and C workload classes requires us to determine the steady state probabilities $\pi(\mathbf{N_1}, \mathbf{N_2}, ..., \mathbf{N_K})$ where $\mathbf{N_i} = (n_1, n_2, ..., n_C)$ is a vector of the number of requests of each workload class c at service center i. However, calculating the steady state probabilities for a general QN requires us to construct the complete state space. This is can be a compute and memory-intensive task and suffers from the problem of state space explosion with increasing numbers of service centers and requests (Bolch et al., 2006, p. 2). However, the construction of the complete state space is not required for *product-form* QNs. Product-form QNs have a special structure, so that the steady state probability of the QN can be easily computed from the ones of the individual service centers (Bolch et al., 2006, p. 281):

$$\pi(\mathbf{N_1}, \mathbf{N_2}, ..., \mathbf{N_K}) = \frac{1}{G} \left[\pi(\mathbf{N_1}) \cdot \pi(\mathbf{N_2}) \cdot \ldots \cdot \pi(\mathbf{N_K})\right] \tag{2.16}$$

G is a normalizing constant. In other words, we can calculate the steady state probabilities of each service center in a product-form QN independently.

Kelly showed that every QN with quasi-reversible service centers and Markovian routing has a product form (Kelly, 1975, 1976). Quasi-reversibility means "that the current state, the past departures and the future arrivals are mutually independent" (Balsamo, 2000). Markovian routing means that the routing of requests does not depend on the current state of the QN. The BCMP theorem (Baskett et al., 1975) showed that this property holds for the following types of service centers:

1. $M/M/m$ with FCFS scheduling assuming that the service rate does not depend on the workload class,
2. $M/G/1$ with PS scheduling,
3. $M/G/\infty$ with IS scheduling, and
4. $M/G/1$ with Last-Come-First-Serve (LCFS) scheduling with preemption.

The service rate distribution in cases 2, 3, and 4 are required to have rational Laplace transforms. This is in practice no limitation since any exponential, hyperexponential, or hypoexponential distribution fulfills this requirement and all other types of distribution can be approximated using a combination of the former (Cox, 1955).

Furthermore, Baskett et al. (1975) showed that the product-form property holds for these scheduling strategies even with certain forms of state-dependent service rates. Amongst others, the service rate may depend on the number of requests at a service center (Baskett et al., 1975). Thus, service centers with multiple servers are also allowed for PS and LCFS scheduling.

2.4.1.3 Operational Laws

The operational laws of queueing theory provide us a quick and simple way to determine certain average performance measures of a service center. These laws are independent of the arrival and service process, or the scheduling strategy. Therefore, they can be used generally to describe a single service center or a system of several service centers.

The most fundamental law in queueing theory, is Little's Law (Harchol-Balter, 2013, Chapter 6):

$$N_{i,c} = \lambda_c R_{i,c} \tag{2.17}$$

The number of requests $N_{i,c}$ of workload class c at service center i equals to the product of request arrival rate λ_c and the average time in the service center $R_{i,c}$ (i.e., response time).

The Utilization Law is defined as (Menascé et al., 2004, p. 64f):

$$U_{i,c} = \frac{X_{i,c}}{m_i} \cdot D_{i,c}. \tag{2.18}$$

where $U_{i,c}$ is the utilization at resource i due to requests of class c, m_i is the number of parallel servers, $D_{i,c}$ is the mean service demand and $X_{i,c}$ is the throughput.

The Service Demand Law is defined as (Menascé et al., 2004, p. 65f):

$$D_{i,c} = \frac{m_i \cdot U_{i,c}}{X_{0,c}}. \tag{2.19}$$

It relates the service demand $D_{i,c}$ of requests of class c at service center i to the utilization $U_{i,c}$ and the total system throughput $X_{0,c}$ of class c. Additional operation laws can be found in Menascé et al. (2004).

2.4.1.4 Response Time

The response time of a single service center is dependent on its parameters. In the following we show the exact results for different variants with Poisson arrival and service processes. If we have a $M/G/m$ queue with PS or preemptive LCFS scheduling, or a $M/M/m$ queue with class-independent service rates and FCFS scheduling, the response time $R_{i,c}$ is (Bolch et al., 2006, p. 251):

$$R_{i,c} = D_{i,c}\left(1 + \frac{1}{m_i} \cdot \frac{PB_i}{1 - U_i}\right) \tag{2.20}$$

PB_i is the probability that all m_i servers of the service center i are busy, and a new request has to wait in the queue. PB_i can be calculated using the Erlang-C formula (Bolch et al., 2006, p. 250):

$$\begin{aligned} PB_i &= \frac{(m_i U_i)^{m_i}}{m_i!(1 - U_i)} \cdot \pi_{i,0} \\ \text{with } \pi_{i,0} &= \left[\sum_{k=0}^{m_i - 1} \frac{(m_i U_i)^k}{k!} + \frac{(m_i U_i)^{m_i}}{m_i!} \frac{1}{1 - U_i}\right]^{-1} \end{aligned} \tag{2.21}$$

If a service center has IS scheduling, a request never has to wait for service and the response time simplifies to:

$$R_{i,c} = D_{i,c} \tag{2.22}$$

If the number of servers $m_i = 1$, the busy probability of service center i is $PB_i = U_i$. As a result, Equation 2.20 can be simplified to:

$$R_{i,c} = \frac{D_{i,c}}{1 - U_i} \tag{2.23}$$

However, the previous equations are not valid for M/M/m service stations with FCFS scheduling and service rates depending on the workload class. The response time of such a service station can only be approximated. Franks (1999) compared the accuracy of different approximations and proposes the following one:

$$R_{i,c} = D_{i,c} + \frac{PB_i}{m_i} \sum_{s=1}^{C} Q_{i,s} \cdot D_{i,s} \tag{2.24}$$

$Q_{i,c}$ is the queue-length of requests of workload class c at service center i. This is only an approximation, since the ordering of requests can also influence their response times for such service stations.

2.4.1.5 Solution Techniques

Different solution techniques for QNs have been developed in the last decades. They can be broadly classified into simulation and analytic techniques. Event-discrete simulation can be used to analyze arbitrarily complex QNs. However, it is often necessary to simulate a QN for a long time in order to obtain sufficiently accurate results.

Analytical techniques can provide exact solutions of a QN avoiding the overhead of simulation. There are state-space and non-state-space techniques (Bolch et al., 2006). State-space techniques rely on the generation of the complete underlying state space of a QN limiting their scalability to increasing numbers of requests, workload classes and service stations. If certain assumptions are fulfilled, non-state-space techniques can be used instead. Given a product-form queueing network with open workloads, we can apply the equations presented in Section 2.4.1.4 to directly calculate mean performance statistics for the individual queues. Given a product-form QN with closed workloads, the calculation of the normalizing constant G in Equation 2.16 is non-trivial. Mean Value Analysis (MVA) (Bolch et al., 2006) is a recursive algorithm to calculate the queue lengths in closed, product-form QNs, avoiding the direct determination of the normalizing constant G.

2.4.2 Layered Queueing Networks (LQNs)

A Layered Queueing Network (LQN) is an extended formalism based on QNs that enables the combined modeling and analysis of hardware and software contention in a model.

Formalism. Figure 2.7 shows an example LQN model. An LQN model consists of the following elements:

- *Processors* are the top-level element of an LQN and represent processing resources in a system (e.g., a CPU). Processors are depicted in Figure 2.7 as circles. Processors have a scheduling discipline and a multiplicity. The multiplicity specifies the level of parallelism of the processor.

- *Tasks* are executed on exactly one processor and represent logical system entities (e.g., nodes in a distributed system, or software components). Tasks are depicted in Figure 2.7 as parallelograms. A task may call other tasks in a system. Calls between tasks must ensure a strict layering, i.e, tasks may only call other tasks which are on a lower level. The top-most task implicitly represents users of a closed workload and are called

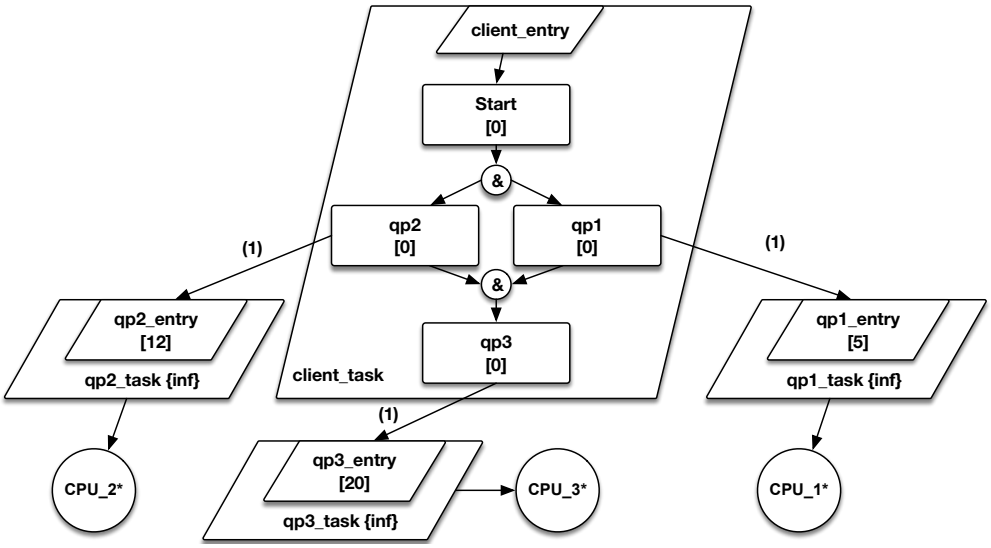


Figure 2.7: Example of an LQN model.

"reference tasks" (Franks et al., 2009). A task has a multiplicity specifying how many instances can execute in parallel.

- *Entries* are used to distinguish different types of requests that a task can accept (i.e., they correspond to workload classes in traditional QNs). Entries accept requests from other tasks in FCFS ordering. The multiplicity (in parentheses) of the call arrow specifies the number of requests sent per each invocation. In addition, synchronous and asynchronous calls are distinguished.

- *Activities* enable the fine-granular description of the behavior within a task. Activities are depicted as rounded rectangles in Figure 2.7 and form a graph. Activities can be either resource demands or calls to other tasks. In addition several predicates are supported such as fork/join or loops. Fork/join predicates can have either logical-and (&) or logical-or (+) semantics. Invocation probabilities can be attached to edges in the activity graph.

Solution Techniques. In order to solve an LQN analytically, a topological sort is performed on the tasks to derive the correct ordering of layers. Each layer l is represented by a closed QN submodel in which each service station corresponds to a task or processor in layer $l + 1$ (Franks et al., 2009). The resource demands

in layer l are the response times of the corresponding tasks in layer $l+1$. Using the method of layers (Rolia and Sevcik, 1995), the submodels are solved using MVA techniques from queueing theory. The response times and number of requests in each layer are iteratively updated with the results of the MVA and propagated across layers. The solution algorithm stops when the the resulting values of the response times and number of requests stabilize.

2.5 Server Virtualization

Virtualization was first introduced in the 1960s for mainframe systems to enable time-sharing of physical system resources between multiple logical VMs each running its own OS (Meyer and Seawright, 1970). Over the years, the concept of virtualization has been extended to different areas in computer science, such as, network virtualization (Rygielski and Kounev, 2013), or desktop virtualization (Miller and Pegah, 2007). In the following, we focus on server virtualization where different logical VMs share the same the physical server.

The adoption of server virtualization techniques increased significantly in the 2000s, when cheap commodity computer systems got powerful enough to run multiple VMs with decent performance. Today's data centers typically consist of large clusters of server nodes with relatively cheap hardware. Each server node is virtualized and application services are running inside VMs, which can be deployed on any of the server nodes. Through virtualization, the resources of individual server nodes in a data-center can be combined into one large logical resource pool and resources can be dynamically allocated from this pool to VMs depending on their current resource requirements. As a result, virtualization is a key technology to implement different Cloud Computing models.

2.5.1 Virtualization Techniques

The piece of software that executes a VM is called *hypervisor* or Virtual Machine Monitor (VMM). According to Popek and Goldberg (1974) a hypervisor has three main characteristics. First, the hypervisor provides an "essentially identical environment" (Popek and Goldberg, 1974), i.e., programs in a VM show the same functional behavior as if executed in a non-virtualized environment. Second, the hypervisor executes the majority of instructions directly without translation in order to minimize the virtualization overhead. Finally, the hypervisor "is in complete control of system resources" (Popek and Goldberg, 1974). Hardware emulation (Bellard, 2005) is not part of virtualization as it simulates

a different computer architecture and therefore requires the translation of each instruction from a guest OS incurring high overheads.

In order to fulfill the characteristics of a hypervisor, an implementation needs to intercept certain privileged instruction so that the isolation between VMs can be ensured. There are three major techniques to implement this:

- *Hardware virtualization* (Seawright and MacKinnon, 1979) means that a VM is executed directly on the physical hardware without the need to modify the guest OS. The hypervisor forwards the instructions from a VM to the physical resource without translation. If certain privileged instructions are executed by a guest OS, the hypervisor is triggered through a system trap to handle them. A computer architecture needs to fulfill certain requirements in order to be virtualizable (Popek and Goldberg, 1974). The classical x86 architecture does not fulfill these requirements, because certain privileged instructions do not result in system traps (Barham et al., 2003).

- *Binary translation* (Adams and Agesen, 2006) is a software technique that dynamically analyzes the binary machine code in the memory of a VM before execution. Privileged instructions are translated on demand according to certain rules. No modifications are required at the a guest OS level, as the translation directly works on the binary machine code. The performance overhead decreases as soon as the working set of a VM is analyzed and translated.

- *Paravirtualization* (Barham et al., 2003) requires specifically modified OSs in the VMs. The hypervisor can execute the majority of instructions directly without translation. The OS needs to be modified manually to use hypercalls instead of non-virtualizable instructions (e.g., the privileged instructions in the x86 architecture). Hypercalls are special instructions of the hypervisor that provide direct access to the underlying physical resources in a controlled manner so that the other VMs are not disturbed. As a result, the virtualization layer is not fully transparent to the OS as in the other cases. However, the modifications are limited to the OS kernel and do not influence the applications inside a VM. Paravirtualization can offer a significant performance improvement in cases where full virtualization is not an option (Barham et al., 2003).

In modern hypervisors, we can often see a mixture of these techniques for different resources. CPU virtualization is typically based on hardware virtualization using special extensions to the x86 architecture (e.g., Intel VT-x

or AMD-V). Other devices (e.g., network devices or storage controllers) are often paravirtualized requiring special drivers within the guest OS for optimal performance.

Typical examples of hypervisors for the x86 architecture are: the commercial VMware ESX, and Microsoft Hyper-V, as well as, the open-source Xen, and KVM. All of these hypervisors support hardware virtualization. VMware ESX also supports binary translation (Adams and Agesen, 2006) and Xen supports paravirtualization (Barham et al., 2003).

2.5.2 Open Virtualization Format (OVF).

VMs can be distributed as VAs to significantly reduce the effort for deploying a software application. The industry standard for creating VAs is the Open Virtualization Format (OVF) (DMTF, 2009) which defines a standard, vendor-independent format for virtual machine images. A standard-compliant VA can be deployed without adaptations on any hypervisor that supports OVF. All major hypervisors (including VMware ESX, Xen and KVM) provide tools to import OVF images. The OVF standard provides the following definition of a VA:

Definition 2.4 (Virtual Appliance)**.** A VA is "a service delivered as a complete software stack installed on one or more virtual machines" (DMTF, 2009, p. 9).

A OVF package consists of the the following parts (DMTF, 2009): a required descriptor, an optional manifest, an optional certificate, and any number of disk images and additional resource files. The descriptor is an Extensible Markup Language (XML) file containing meta-data of the virtual appliance (such as, the virtual machines, their hardware configuration, and additional settings). The manifest and certificate can be used to check the integrity of a VA. A disk image contains the contents of virtual hard disks in a standardized format. Additional resource files can be, for example, ISO images (DMTF, 2009).

The OVF standard is focused on the description of the virtual hardware environment. The actual software stack is contained within the unstructured disk images delivered as part of a VA. OVF does not provide any meta-data on the software stack or any introspection mechanisms to access the dynamic state of a VA.

2.5.3 Run-time Reconfiguration

Today's hypervisors provide different knobs to control the allocation of resources to VMs. In the following, we describe major knobs that can be adapted during system run-time without interrupting the applications in a VM.

Scheduling Priorities A hypervisor is in complete control of the physical system resources and typically implements a time-sharing of the resources between VMs. By default, all VMs on a host system can obtain equal shares of a resource enforced by a resource scheduler within the hypervisor. Most hypervisors support the setting of scheduling parameters of individual VMs in order to change its scheduling priority. These settings may be changed separately for each type of resource. The scheduling parameters that we can change usually depend on the type of hypervisor. In the following, we describe the scheduling parameters provided by VMware ESX in an exemplary manner. VMware ESX uses a preemptive, work-conserving scheduling strategy supporting the following parameters (Gulati et al., 2012):

- The *reservation* is the guaranteed amount of resources a VM can consume, even if all resources are demanded by other VMs. The sum of reservations of all VMs on a host system is required to be less or equal to the available resources.
- The *limit* is the maximum amount of resources a VM can consume, even if the resources are not requested by other VMs. The limit is always greater or equal to the reservation setting.
- The *share* is the priority of a VM with which it can get additional resources above its reservation from other VMs. The share is an absolute quantity and the actual priority depends on the share of the other VMs on the same host system.

The amount of resources is specified using resource-specific metrics. These are CPU time (in MHz), memory size (in MB), storage throughput (in IOPS) and network bandwidth (in Mbit/s).

In addition to the scheduling at the level of a host system, VMware vSphere provides a resource scheduler at the level of a cluster, called VMware Distributed Resource Scheduler (Gulati et al., 2012). A *resource pool* is a logical grouping of resources within a cluster and may contain a set of VMs. Each resource pool has associated reservation, limit and shares settings which apply to all containing VMs. The distributed resource scheduler ensures that the resource pool as a whole gets the demanded resource share, even if the individual VMs are distributed across different host systems. A distributed algorithm (Gulati et al., 2012) continuously adapts the scheduling priorities of the individual VM in a resource pool so that the settings of the whole resource pool are fulfilled. Resource pools can isolate groups of VMs from each other (e.g., test and production systems, or applications from different departments).

Resource Hot-Add and Hot-Remove In most recent years, hypervisors added the capability to add (or remove) virtual resources, such as virtual CPUs (vCPUs), memory, or I/O devices to running VMs. This is referred to as hot-add (or hot-remove). This way, for example, one can dynamically reconfigure a VM from a 2-vCPU, 8 GB memory configuration to an 8-vCPU, 32 GB memory configuration without restarting it. This is referred to as vertical scaling of a VM. Vertical scaling provides a running VM immediate access to more resources (bigger memory, more CPU instances). In order to leverage the additional resources, the guest operating system needs to support the hot-plug of resources, which many modern operating systems do.

VM Provisioning, Initial Placement and Admission Control New VM instances can be provisioned at any time during system operation, for instance, to scale an application horizontally by adding additional instances. When new VM instances are started in a virtualized cluster, a suitable host system needs to be identified with free resources. Virtualization allows over-commitment, i.e., the sum of all virtual resources may be greater than the available physical resources on a host system. A data center operator may allow a certain level of over-commitment in order to increase the resource efficiency in a virtualized cluster assuming that not all resources are requested at the same time. Admission control mechanisms are used to control the number of VMs deployed on a host in order to guarantee a minimum amount of resources in over-committed scenarios and reject new VMs instances if a host would be overloaded.

Live Migration and Load Balancing VMs can be migrated at run-time between hosts in a virtualized cluster without restarting the guest OS. When a live migration is triggered, a new clone of the VM image is created on the destination host and the current contents of the main memory are transferred through the network to the designated destination host. The applications in the VM can continue to execute, while the memory contents are transferred to the destination. All memory pages that are modified concurrent to the data transfer are marked and transferred again in a another iteration. This procedure is repeated until no modified pages are remaining or a maximum number of iterations is reached. Then the processing within the VM is interrupted, any remaining memory pages are copied to the destination host, the network connections are rerouted and the VM resumes processing on the destination host. The application is only unavailable during this last step. It typically takes less than one second to complete and is therefore mostly transparent to clients.

Live migration techniques can be used to balance the load in a virtualized cluster. If single hosts in a cluster are over-loaded, VMs can be dynamically moved to hosts with lower utilization until the load of all hosts in a cluster is balanced again. Furthermore, live migration techniques can be used to consolidate VMs during phases of low load on a reduced number of hosts. Then, unused hosts can be put into a stand-by mode in order to save energy.

Chapter 3

State-of-the-Art

3.1 Autonomic Resource Management in Virtualized Data Centers

In recent years, Galante and Bona (2012), Jennings and Stadler (2015), and Lorido-Botran et al. (2014) surveyed and classified approaches to autonomic resource management in virtualized data centers. Jennings and Stadler (2015) propose a conceptual framework for resource management based on eight functional areas. We adopt this conceptual framework here and present related work for five of these areas which are closely related to our approach.

3.1.1 Capacity Planning

Capacity planning subsumes activities to determine the changing resource requirements of applications on a longer time scale (e.g., the acquisition of additional physical servers, or deployment of new applications). Capacity planning can be performed at the application level (i.e., using request arrival rates) or at the infrastructure level (i.e., using resource usage statistics). Existing capacity planning tools, such as VMware vRealize Operations, enable the analysis of historic resource usage data and partly provide forecasting techniques based on time series analysis. However, the forecasting is usually limited to simple extrapolation of historic time series.

Academic researchers proposed more powerful forecasting techniques. The survey by Herbst et al. (2014) covers existing techniques for workload forecasting based on time series analysis and the authors propose an approach to automatically select the optimal technique using a decision tree. While time series models can capture temporal dependencies well, resource usage observations in data centers exhibit often also spatial dependencies between VMs. Xue et al. (2015) show that neural networks are well suited to incorporate spatial dependencies in forecasts. In Xue et al. (2016), the same authors add clustering and step-wise regression techniques to extract representative signature time

series from a large set of observations to reduce the training efforts of the neural network.

Stochastic performance models, such as QNs, have been widely used for off-line capacity planning (Lazowska et al., 1984; Menascé et al., 2004). For instance, QNs with a pre-defined structure are used in Urgaonkar et al. (2007) and Zhang et al. (2007) to predict the resource requirements of applications for a given workload. Both works also consider the estimation of model parameters based on empirical data. However, these models are only used in off-line settings for manual analysis of resource requirements.

3.1.2 Local Resource Scheduling

A hypervisor controls the allocation of resources to VMs running on the same physical machine. If resource allocations are dedicated, local resource scheduling needs to ensure performance isolation between VMs. If resources are over-committed, modern hypervisors offer capabilities to adjust the scheduling priority of individual VMs (see Section 2.5.3). While local resource scheduling for CPU and memory is well understood, recent research focuses on the optimization of the local scheduling of network and storage resources (see Jennings and Stadler, 2015).

Jennings and Stadler (2015) also identify several academic approaches (Padala et al., 2009; Rao et al., 2011; Xu et al., 2008) proposing autonomic control schemes to provide Quality of Service (QoS) differentiation for applications by adapting the scheduling priorities in over-committed scenarios. Padala et al. (2009) propose a multi-input multi-output, feedback controller based on a time-series Auto-Regressive-Moving-Average (ARMA) model. Xu et al. (2008) use a model based on fuzzy logic to capture the relationship between resource allocation and application performance. The fuzzy rules are learned automatically using clustering techniques on empirical monitoring data. Rao et al. (2011) describe a controller based on reinforcement learning techniques to predict the impact of a reconfiguration. Blagodurov et al. (2013) use a control loop with a linear model to determine the CPU requirements depending on the number of SLA violations in previous intervals. The control loop dynamically adjusts the scheduling priorities of VMs. All these approaches have in common that they abstract the application behavior as a black-box. Therefore, they depend on the availability of sufficient empirical data covering the possible reconfiguration space in order to be able to learn a function between resource allocation and application performance. An alternative approach based on a queueing model is proposed by Chandra et al. (2003). They represent the resource under control as a queueing station with Generalized Processor Sharing (GPS) scheduling

strategy and propose an optimization algorithm to determine scheduling priorities to minimize response time SLA violations. They automatically determine the model parameters from empirical data and adapt the system in regular intervals. However, they represent the application with a single queueing station, not considering the complex architecture of today's service-oriented applications.

3.1.3 Global Resource Scheduling

Global resource scheduling is responsible for the allocation of resources in a virtualized cluster consisting of multiple physical machines. This comprises the initial placement and admission control of newly arriving VMs as wells as the dynamic placement of existing VMs (e.g., using live migration techniques) to ensure the fulfillment of cluster-wide goals (e.g., load-balancing, locality of network traffic, or reduction in energy consumption). In general, global resource scheduling can be formulated as a static or dynamic bin-packing problem, which is known to be NP-hard. Therefore, research is mainly focused on finding good heuristics for global resource scheduling.

An industrial solution for global resource scheduling is the VMware Distributed Resource Scheduler (DRS) (Gulati et al., 2012) product. DRS realizes admission control and initial placement of VMs in a virtualized cluster in order to fulfill resource reservations. Furthermore, it autonomically schedules VMs on physical machines in order to balance the load in a cluster and to reduce energy consumption by putting physical machines with low utilization into stand-by mode. If workloads change, VMs are dynamically moved between physical hosts using live migration techniques. Zhu et al. (2009) propose a similar approach to global resource scheduling as part of HP's "1000 islands" framework.

Academic researchers have proposed numerous utility functions and solution heuristics (e.g., Gupta et al., 2008; Jung et al., 2008; Li et al., 2009) improving global resource scheduling (see survey in Jennings and Stadler, 2015). Most of these approaches do not consider the application SLAs in their scheduling decisions. Exceptions are Bennani and Menascé (2005), Jung et al. (2008) and Zhu et al. (2009). Bennani and Menascé (2005) assign servers to application optimizing the fulfillment of SLAs. A local controller predicts the performance of an application using QNs models. These models need to be provided beforehand. Jung et al. (2008) use an offline LQN model to generate adaptation policies by taking into account the predicted response times. The resulting fixed set of adaptation policies are then applied at run-time to optimize the resource allocation. Zhu et al. (2009) describe a multi-level architecture using

the local resource scheduling controller described in Padala et al. (2009) on the lowest level and building a global resource scheduler on top of it. Lu et al. (2014) extend the work of Padala et al. (2009) to also control the priority settings of logical resource pools in a virtualized data-center in accordance with application SLAs.

The approaches to global resource scheduling described above have in common that they are all reactive. Given that reconfigurations at the global level can be expensive in terms of additional overheads, proactive approaches are also considered in the research community. Shanmuganathan et al. (2013) describe how proactive resource management might be achieved at the granularity of an entire cluster of VMs. vManage (Kumar et al., 2009c) uses short range forecasts (15 minutes into the future) to optimize VM placement on physical hosts and avoid ping-pong of VM migrations.

3.1.4 Application Workload Management

Application workload management comprises admission control and load-balancing of requests at the application level. Classic approaches to load-balancing (Kremien and Kramer, 1992; Wang and Morris, 1985) can be used here. More recent work incorporates additional aspects (e.g., SLA compliance or energy consumption) into load-balancing algorithms (e.g., Liu et al., 2012; Tumanov et al., 2012; Zhang et al., 2011).

3.1.5 Application Scaling

Applications in virtualized data centers can be scaled in two different ways: *horizontal scaling* referring to automatically changing the number of servers (physical or virtual) hosting a multi-tier application, and *vertical scaling* referring to adjusting the effective "sizes" of individual servers. The main goal in both cases is to elastically allocate resources to ensure application SLAs under time-varying workloads while avoiding over-provisioning of resources.

Lorido-Botran et al. (2014) survey auto-scaling techniques for virtualized applications and propose the following categorization of approaches: static threshold-based rules, reinforcement learning, queueing theory, control theory and time-series analysis. We adopt these categories here and additionally consider scaling techniques based on meta-models as a separate category.

Threshold-based Rules. Commercial cloud offerings, such as Amazon EC2 or Microsoft Azure, or open-source cloud management solutions, such as OpenStack or Apache CloudStack, or third-party cloud resource management

tools, such as RightScale (2016), all follow a reactive approach based on static threshold-based rules. They are limited to horizontal scaling providing integrated load-balancing capabilities. Optimal values for the thresholds when to scale up or down are highly dependent on the application workload characteristics, which may change over time. In order to avoid oscillations in the controller, additional quiet times are required after a reconfiguration. RightScale uses a democratic voting system (i.e., the majority of the servers need to vote for scale up) to avoid accidental scaling actions (RightScale, 2016). Kupferman et al. (2009) show that RightScale is still very sensitive to the chosen threshold settings. Dawoud et al. (2012) compare vertical and horizontal scaling and use a simple threshold-based controller for adapting the number of vCPUs.

Reinforcement Learning. To overcome the limitations of static threshold-based rules, a number of approaches employ reinforcement learning to automatically determine good rules for application scaling without requiring a-priori knowledge of the application behavior. Lorido-Botran et al. (2014) identifies several approaches in this category: Barrett et al. (2013), Dutreilh et al. (2011), and Tesauro et al. (2007) for horizontal scaling and Rao et al. (2009) for vertical scaling. More recent work in this area are VScaler (Yazdanov and Fetzer, 2013) for vertical scaling and vScale (Padala et al., 2014) for horizontal scaling.

All these approaches have in common that they assume a state space S based on the current system configuration (e.g., number of replicated VMs, and current workload intensity), a set of possible reconfiguration actions A (e.g., add, remove, or maintain VM instances) and a reward function $R : S \times A \to \mathbb{R}$. The action with the highest reward is always applied in each control interval. The reward function is learned automatically at system run-time by observing the impact of reconfigurations on the application performance. The main challenges of reinforcement learning approaches are long training times and large state spaces (Lorido-Botran et al., 2014).

Hybrid approaches use additional offline models to speed up the training of reinforcement learning models. Tesauro et al. (2007) use a simple QN to perform initial training without the need to reconfigure the real system. They expect the QN as input and estimate the resource demands based on empirical data. Furthermore, neural networks have been used to predict the reward of unobserved states in case of large state spaces (Rao et al., 2009; Tesauro et al., 2007).

Feedback Control. Classic feedback controllers from control theory, such as, proportional integral derivative (PID) controllers, assume continuous actuators

(e.g., resource allocation mechanisms). In virtualized systems, this assumption is typically only given for the scheduling priorities used for local resource scheduling. However, application scaling typically relies on discrete adaptation mechanisms (e.g., add or remove CPUs or VMs). Lim et al. (2010) describe a proportional thresholding technique that can be used with a fixed-gain feedback controller in order to implement horizontal scaling of VMs. However, their approach requires a pre-identified model with manually determined parameter values.

Adaptive control schemes also learn and adapt parameters of a feedback controller at run-time. (Xu et al., 2008) propose a controller based on fuzzy-logic rules to automatically adapt the CPU limits (aka. caps) depending on the application workload. The fuzzy-logic rules are learned at system run-time to represent the relationship between the resource allocation and the application performance. Bodík et al. (2009) use statistical machine learning techniques based on time series analysis to automatically derive a transfer function based on smoothing splines for the application performance. They use the model for horizontal scaling of VMs in order to meet a target SLO for a multi-tiered web-application. Kalyvianaki et al. (2014) describe a Kalman filter design for tracking the CPU utilization in multi-tier applications and setting the CPU limit of VMs accordingly. They do not explicitly consider the application performance and instead assume a constant factor specifying a required headroom between the allocation and the actual utilization.

Time-series Analysis. Approaches based on time-series analysis typically use historic data about the resource usage of VMs to predict their future demands in order to adapt the resource allocations. PRESS (Gong et al., 2010) uses Fast Fourier Transforms (FFTs) to identify repeating patterns and fits a discrete-time Markov Chain for other cases. The predictions are used to adapt the limits settings of VMs. CloudScale (Shen et al., 2011) uses multiple statistical techniques, including FFTs to identify repeating patterns, discrete time *Markov Chains* to predict short-term demand, online adaptive padding and incremental (adaptive) over-provisioning to remedy and detect under-provisioning. This information is used for short-term optimization of CPU and memory limits.

AGILE (Nguyen et al., 2013) uses wavelets and curve fitting to predict resource demands and proactively adds additional VM instances if an application will be overloaded. The authors use resource pressure models to determine the amount of resources required by an application. Yazdanov and Fetzer (2012) use an auto-regressive prediction model to predict the resource requirements of a VM in order to dynamically hot-plug CPUs in Xen. In summary,

existing approaches based on time-series analysis support proactive reconfiguration, however, they work on low-level resource usage statistics and take the application performance into consideration.

Queueing Theory. Villela et al. (2004) use a QN model representing each server with a M/G/1/PS queue. The model needs to be created and maintained manually. It is used to dynamically adjust the number of servers depending on the current workload. They perform their evaluation in a simulated environment.

Urgaonkar et al. (2008) combine a proactive (covering hours and days) and a reactive (for short-term reconfigurations) controller for horizontal scaling of multi-tier applications. The controllers are based on a queueing model for multi-tier applications where each server is represented by a G/G/1 queue. The approach takes an unparameterized model as input and assumes the direct availability of measurements of service times and interarrival times including their distribution.

Jung et al. (2010) describe the Mistral controller which optimizes the resource allocation of VMs with regards to their performance and power consumption. The authors consider different types or reconfigurations: vertical and horizontal CPU scaling, VM migration and shutdown of physical hosts. LQN models are used to predict the impact of reconfigurations and workload changes on the application performance. These models are created and parameterized in an off-line step performing dedicated experiments with the application.

Gandhi et al. (2011) model a single-tier application with a M/G/1/PS queue and combine a proactive with a reactive controller to automatically adapt the number of servers to the incoming workload. The prediction model needs to be provided in advance and is not updated at run-time. In Gandhi et al. (2014), the authors use a Kalman filter to estimate the resource demands dynamically, however, they assume a fixed system architecture (three tiers and three workload classes).

Descriptive Meta-Models. Hoorn (2014) and Huber et al. (2014) both use descriptive, architecture-level performance models based on meta-models for online resource management. Hoorn (2014) uses the SLAstic meta-model to describe the system architecture at a coarse-grained abstraction level. However, the authors do not integrate predictive capabilities in their controller and work with fixed thresholds triggering reconfigurations. Huber et al. (2014) propose an approach for run-time reconfiguration of virtualized systems using the DML meta-model. They consider vertical as well as horizontal CPU scaling, and assume the availability of a complete model of the controlled system. The

impact of a reconfiguration is determined using a full event-discrete simulation of the DML model. Due to the overhead of the simulation, this approach is only feasible for planning reconfigurations on a longer time scale (e.g., hours or days). Furthermore, the approach expects a complete DML model as input, that needs to be created manually.

3.1.6 Discussion

While the enforcement of resource-level SLAs on a local and a global level in data-centers is a well understood topic and autonomic solutions are already used in practice, the efficient control of application SLAs is an open research challenge. In general, control theory distinguishes between feed-forward and feedback control. On the one hand, the use of feed-forward control based on static threshold-based rules is limited due to the manual effort required to determine good thresholds for a certain application and due to their inability to adapt to changing workload characteristics. While reinforcement learning has been successfully used to mitigate manual efforts when determining thresholds, it still requires long training times in case of changing workloads.

On the other hand, feedback control provides established techniques to automatically learn a transfer function between certain control knobs (i.e., resource allocation) and the system output (i.e., application performance). However, in practical systems many control knobs for resource allocation are discrete variables (e.g., vCPUs) with a strong influence on the system output, making feedback control hard to implement. As a result, we argue that feed-forward control with a model explicitly capturing the influence of the workload characteristics and the application architecture should be used here.

Several authors (Gandhi et al., 2011; Jung et al., 2010; Urgaonkar et al., 2008; Villela et al., 2004) proposed to use queueing theory models to predict the required resource allocation of an application under a given workload. Other authors (Hoorn, 2014; Huber et al., 2014) use descriptive meta-models to improve the model expressiveness. However, these approaches have in common that they expect the model (i.e., model structure and often also the model parameter values) as input, requiring deep performance modeling expertise from a user. A few of these approaches also consider the adaptive estimation of resource demands, however, only under strong assumptions on the model structure (e.g., number of queues, or scheduling strategies).

3.2 Performance Model Extraction

In the previous section, we described different approaches to autonomic resource management in virtualized data centers. While model-based approaches promise benefits for autonomic resource management with regards to the state-of-the-art, their limited model learning capabilities are an obstacle for a wider adoption. In this section, we review the state-of-the-art on performance model extraction in the software performance engineering community. In the past, this community proposed a number of meta-models for building *architecture-level* performance models of software systems have been proposed to support the performance analysis of software architectures at design time. Such models provide modeling constructs to capture the performance-relevant behavior of a system's software architecture as well as some aspects of its execution environment (Koziolek, 2010). The most prominent meta-models are the UML SPT and MARTE profiles (OMG, 2006), both of which are extensions of UML as the de-facto standard modeling language for software architectures. Further proposed meta-models include SPE-MM (Smith et al., 2005), CSM (Petriu and Woodside, 2007), KLAPER (Grassi et al., 2007) and PCM (Becker et al., 2009). Architecture-level performance models are built during system development and are used at design and deployment time to evaluate alternative system designs or predict the system performance for capacity planning. Over the past decade, with the increasing adoption of component-based software engineering (Crnkovic et al., 2005), the performance engineering community has focused on extending modeling approaches to support component-based systems.

3.2.1 Monitoring, Instrumentation and Measurement-Based Analysis

In industry, system management solutions (such as IBM Tivoli, HP OpenView, CA Unicenter, BMC Patrol) have been used since decades to manage the hardware and software inventory of enterprise IT infrastructures. These solutions typically include monitoring components that collect performance metrics of the infrastructure and the applications. There are also many open-source alternatives for systems monitoring (e.g., Hyperic or Zenoss) with similar functionality. The monitoring data is obtained using standard system interfaces or special agents deployed on each monitored system. Traditionally, the monitored metrics collected by these monitoring components are rather coarse-grained (e.g., average system response time, throughput, or resource utilization) and limited to standard applications.

In recent years, Application Performance Management (APM) tools, such as Dynatrace (Rometsch and Sauer, 2008), or AppDynamics, were developed providing fine-grained instrumentation and monitoring capabilities at the application level. APM tools can typically trace individual transactions in a production system through multiple application tiers and may provide application performance metrics on a per-function basis. This requires instrumentation probes at application run-time, although most tools come with transparent support for commonly used technologies (e.g., bytecode instrumentation in Java) so that no changes to the application source code are required. APM tools help to manually identify root causes of performance problems. However, they cannot provide models supporting analysis and reasoning on the system architecture.

In the research community, Carrera et al. (2003) and Rohr et al. (2008a) propose instrumentation platforms for the measurement-based performance analysis of Java applications. Carrera et al. (2003) supports fine-grained instrumentation at different levels (application, Java VM and operating system) in order to determine the resource accessing behavior of an application. Different visualizations are available supporting performance engineers to detect bottlenecks. Rohr et al. (2008a) developed Kieker, a monitoring framework for tracing individual requests in a system and observing their response time distributions. However, both instrumentation platforms are limited to the monitoring and the visualization of collected data.

Several researchers applied non-parametric regression techniques to interpolate measurements taking into account the influence of multiple system parameters. Courtois and Woodside (2000) uses Multivariate Adaptive Regression Splines (MARS) to fit resource functions to measurements. The authors are able to derive resource functions for the computational overhead of TCP/IP communications depending on given workload parameters. Westermann et al. (2012) describe a systematic approach to derive performance prediction functions describing the functional relationship between system performance and certain system parameters from empirical data. They propose techniques to optimize the experimental exploration of the parameter space based on given prediction goals. They support different types of statistical inference models (MARS, classification and regression trees, genetic programming, Kriging) to learn performance prediction functions. Noorshams et al. (2013) propose a black-box performance prediction approach for virtualized storage systems. They derive performance prediction functions for storage systems using automated experiments to obtain measurement data. The approach supports different regression techniques and uses cross-validation to automatically select

and optimize a regression technique for the best accuracy. While the experiment automation is specific to storage systems, the black-box performance prediction approach can be generalized to other systems.

An alternative approach to measurement-based analysis is based on clustering techniques. Zhang et al. (2013) use a density-based spatial clustering technique to predict the performance of services from historic observations. The authors introduce "service invocation patterns" describing the performance-influencing parameters of services and then build clusters based on these patterns. These clusters can then be used to predict the service performance by classifying new service invocation patterns. However, all measurement-based analysis techniques suffer from their limited extrapolation capabilities. While they do not require insight into the system architecture, they typically require large amounts of empirical measurements covering all possible system states in order to extract representative and accurate models. Measurement-based analysis techniques are therefore mainly applicable to the off-line analysis of systems running in a controlled experiment environment.

3.2.2 Static Structure and Dynamic Behavior

In order to create the structure of a performance model, it is necessary to derive the system architecture. We assume a component-based architecture, hence, the following aspects need to be covered: static component structure, dynamic control flow between and inside components, as well as the execution environment.

Static Component Structure. Existing work on reverse-engineering of component architectures mostly focuses on the extraction of design models. Ding and Medvidovic (2001) propose a reverse-engineering approach called "Focus" which derives a UML architecture model using static code analysis. ROMANTIC (Chardigny and Seriai, 2010) also uses static code analysis techniques and existing system documentation as input to reverse engineering. Kebir et al. (2012) propose two different approaches for component identification: explorative and requirement-driven. The former uses clustering techniques based on class definitions in the source code. The latter requires a software architecture to mark key classes or interfaces. Chouambe et al. (2008) describe the "ArchiRec" approach for component identification in order to reverse-engineer component models from existing source code. Components are identified using a set of heuristics defined on code metric. These code metrics are obtained through static analysis techniques.

While static code analysis techniques may also be applied at system runtime, they often produce suboptimal results since design-time and run-time architectures may be different. The approaches in Brosig et al. (2011), Brunnert et al. (2013), and Hoorn (2014) for performance model extraction at system runtime instead propose technology-specific mapping rules from implementation component models (e.g., Java EE).

Dynamic Control Flow. Due to well-known limitations of static code analysis techniques (e.g., with regards to polymorphic method invocations), the extraction of the control flow usually requires dynamic analysis techniques. The monitoring and instrumentation techniques described in Section 3.2.1 may be used to collect the required empirical data.

Reverse-engineering approaches for UML models have been proposed in the past. For instance, Briand et al. (2006) propose a reverse-engineering approach for UML sequence diagrams based on a dynamic analysis of distributed Java applications. The sequence of messages is extracted from traces obtained using a custom application instrumentation. However, when using models for performance prediction, it is important to find a suitable level of abstraction in order to limit the overhead for analysis such models. In the following, we focus on approaches specifically targeted at the extraction of performance models.

In Hrischuk et al. (1999), LQN models are generated automatically from traces recording the distributed message flow between operations in an application. The ordering of messages is based on the timestamps contained in the traces. Requests can be traced across several nodes in a distributed system. In order to correlate events from independent nodes in a distributed system, the approach assumes the availability of a so-called "angio dye id" request parameter (Hrischuk et al., 1999), which is appended to each request at the system boundary and propagated through the system. In order to mitigate the need for appending additional fields to requests, Israr et al. (2007) propose message correlation techniques based on a generic message-information field in the traces derived automatically from existing request parameters. Hrischuk et al. (1999) and Israr et al. (2007) first build up an interaction graph and then derive an LQN model using a set of rules. Both approaches are focused on extracting the dynamic control flow of the application and expect model parameters (e.g., resource demands) as an input.

Several researchers considered the extraction of architecture-level Palladio Component Model (PCM) models, which are typically used for performance predictions at design time. Krogmann et al. (2010) use genetic search techniques to determine parameterized behavioral models of individual compo-

nents. These models describe the performance-relevant control flow in a component, including control flow and data flow dependencies on input parameter. In order to determine dependencies on input parameter, it relies on dynamic analysis of the code using a bytecode instruction counting technique. The bytecode instruction counting introduces hight overheads of up to 250% (Krogmann et al., 2010) making it only applicable in test environments. Brosig et al. (2011) rely on proprietary instrumentation techniques provided by the Oracle WebLogic middleware platform to collect the required information on the request flow within an application. They automatically determine the control flow between and inside components. Brunnert et al. (2013) use traces collected by standard Java EE instrumentation techniques instead, and describe a tool, called PMW, that automatically generates a PCM instance from these traces. In Hoorn (2014), the author describes the SLAstic approach which supports the extraction of the dynamic software architecture based on traces collected by the Kieker framework (Rohr et al., 2008a). The author proposes an own meta-model to describe system architecture and extract the control flow between distributed software components including call frequencies. Furthermore, they describe an automatic transformation from their meta-model to PCM.

3.2.3 Workload Characterization

User Behavior. Menascé et al. (1999) introduce a model called Customer Behavior Model Graph (CBMG) for describing "the behavior of groups of customers who exhibit similar navigational patterns" (Menascé et al., 1999). They employ a K-means clustering algorithm to derive CBMGs automatically from session logs. Hoorn et al. (2015) introduces a domain-specific language, called WESSBAS, for describing workload models and use X-means clustering generate these models from session logs. Vögele et al. (2015) propose an automatic transformation from the WESSBAS domain-specific language to PCM usage profile models.

In (Kistowski et al., 2015), a meta-model-based language is introduced to describe the temporal course of the workload intensity of a system. The authors also propose an algorithm to extract these models from a time-series of arrival rates. The extraction method uses statistical methods to split the time-series into seasonal, trend, burst and noise components.

Resource Demands. Profiling tools (Graham et al., 1982; Hall, 1992), typically used during development to track down performance issues, provide information on call paths and execution times of individual functions. These profiling tools rely on either fine-grained code instrumentation or statistical sampling.

However, these tools typically incur high measurement overheads, severely limiting their usage during production, and leading to inaccurate or biased results. In order to avoid distorted measurements due to overheads, Kuperberg et al. (2008a,b) propose a two-step approach. In the first step, dynamic program analysis is used to determine the number and types of bytecode instructions executed by a function. In a second step, the individual bytecode instructions are benchmarked to determine their computational overhead. However, this approach is not applicable during system operatios and fails to capture interactions between individual bytecode instructions. APM tools, such as Dynatrace (Rometsch and Sauer, 2008) or AppDynamics, enable fine-grained monitoring of the control flow of an application, including timings of individual operations. These tools are optimized to be also applicable to production systems, however, they do not create models abstracting from the raw measurement data.

Modern operating systems provide facilities to track the consumed CPU time of individual threads. This information is, for example, also exposed by the Java runtime environment. This information can be exploited to measure the CPU resource consumption of processing individual requests as demonstrated for Java by Brunnert et al. (2013) and at the operating system level by Barham et al. (2004). This requires application instrumentation to track which threads are involved in the processing of a request. This can be difficult in heterogeneous environments using different middleware systems, database systems, and application frameworks. The accuracy of such an approach heavily depends on the accuracy of the CPU time accounting by the operating system and the extent to which request processing can be captured through instrumentation.

Over the years, a number of approaches to estimate the resource demands using statistical methods have been proposed. These approaches are typically based on a combination of aggregate resource usage statistics (e.g., CPU utilization) and coarse-grained application statistics (e.g., end-to-end application response times or throughput). These approaches do not depend on a fine-grained instrumentation of the application and are therefore widely applicable to different types of systems and applications incurring only insignificant overheads. Different approaches from queuing theory and statistical methods have been proposed, e.g., response time approximation (Brosig et al., 2009; Urgaonkar et al., 2007), least-squares regression (Bard and Shatzoff, 1978; Pacifici et al., 2008; Rolia and Vetland, 1995), robust regression techniques (Casale et al., 2008; Casale et al., 2007), cluster-wise regression (Cremonesi et al., 2010), Kalman Filter (Kumar et al., 2009a; Wang et al., 2012; Zheng et al., 2008), optimization techniques (Kumar et al., 2009b; Liu et al., 2006; Menascé, 2008; Zhang et al., 2002), Support Vector Machines (Kalbasi et al., 2011), Independent Com-

ponent Analysis (Sharma et al., 2008), Maximum Likelihood Estimation (Kraft et al., 2009; Pérez et al., 2013; Wang and Casale, 2013), and Gibbs Sampling (Pérez et al., 2013; Sutton and Jordan, 2011). These approaches differ in their required input measurement data, their underlying modeling assumptions, their output metrics, their robustness to anomalies in the input data, and their computational overhead.

3.2.4 Discussion

While monitoring tools in industry provide numerous statistics about the system state and fine-grained instrumentation capabilities are increasingly employed also in production system (e.g., in APM tools), techniques to automatically build abstract models from the raw monitoring data, e.g., for performance predictions, are not widely used in industry. In the last decade, a number of approaches have been proposed that show that it is feasible to automatically extract complete architecture-level performance models using a set of static and dynamic analyses. However, these approaches are typically designed for offline analysis: First, dynamic monitoring data is collected either in a controlled experiment or in a production system. Afterwards a user manually triggers the performance model extraction.

When using model extraction techniques to learn models for autonomic resource management in data centers, a model needs to be updated continuously to reflect changes in the workloads and system configuration. This requires a deep integration of the model extraction logic into the system itself in order to be able to react to changes in the system configuration. Furthermore, approaches need to continuously estimate and update the model parameters. A crucial parameter for resource management are the resource demands of application workloads. Robust and accurate techniques to determine these resource demands are a prerequisite to use fine-grained models for autonomic resource management ind data centers. While many for estimating resource demands from monitoring data have been proposed in the past, a systematization and an experimental comparison are missing providing guidelines when to use a certain technique to ensure highly accurate resource demand estimates.

Part II

Self-Aware Resource Management in Virtualized Systems

Chapter 4

Online Model Learning in Virtualized Data Centers

The definition of "Self-Aware Computing" (see Section 2.1) highlights the need to automatically learn models of a system and its environment at run-time in order to enable autonomic reasoning and adaptation according to higher-level goals (Kounev et al., 2015). When mapping this idea to the resource management in virtualized data centers, models help us to determine required system adaptations to fulfill goals regarding performance, availability, and resource efficiency. It is important to note here, that models covering the application architecture (including its static structure as well as dynamic behavior) are crucial given that the higher-level goals are usually specified on the application level (e.g., response time or throughput goals). Resource management techniques need to consider these application-level performance goals when changing the resource allocation to an application. As a consequence, they depend on the availability of performance models enabling the prediction of the expected impact of changes to the resource allocation on the application performance. These models need to capture the influence of different platform and application layers (e.g., hypervisor, middleware systems and application components) in order to provide reliable predictions.

Existing approaches to online resource management in virtualized environments are typically based on black-box or coarse-grained performance models abstracting systems at a high level severely limiting their prediction capabilities (see Section 3.1). Individual effects and complex interactions between the application workloads and the system layers are considered as static and viewed as a black box. This hinders fine-grained performance predictions that are necessary for efficient resource management (e.g., predicting the effect on the response time, if a virtual machine of an application tier is replicated or migrated). Recent approaches explore the usage of descriptive *architecture-level* performance models for online resource management. For instance, Brosig (2014) and Huber (2014) propose a meta-model for online resource management

called DML (see Section 2.2.2). It supports the explicit modeling of layered system architectures and provides abstractions for the fine-grained description of the performance-relevant behavior of a system. While Brosig (2014) and Huber (2014) show that architecture-level performance models provide significant benefits for online resource management in terms of more fine-grained and more flexible reasoning and prediction capabilities, the creation of such models is still a challenge. The goal of this chapter is the integration of model learning capabilities into virtualization platforms enabling the automated creation and maintenance of architecture-level performance models at system run-time.

Challenges An approach to automatic model learning needs to address a number of challenges to be applicable in virtualized data centers. We describe these challenges in the following.

- Given that model learning is performed during system operation, the system workload and configuration cannot be controlled. We rely on empirical observations while applications are serving production workloads. In order to avoid significant overheads on the performance of services in a data center, existing monitoring infrastructures and platform interfaces should be used to obtain the empirical information required for model learning.

- The integration of model learning capabilities into systems requires a pro-found understanding of the system architecture – including the application and any platform layers – and at the same time a deep knowledge of performance modeling techniques. However, system administrators often do not have sufficient skills to perform such tasks. Furthermore, it can be time-consuming and costly to design and implement model learning capabilities for a given system. Therefore, ways to enable the reuse and sharing of model learning capabilities between systems are necessary.

- Multiple applications with diverse technology stacks typically share the same underlying infrastructure in virtualized environments influencing each other. A performance model needs to represent the complete virtualized system (including the different applications) integrating information from heterogeneous data sources. However, the deployment of applications and their software stacks are often not known before system run-time (especially with the advancement of on-demand provisioning of VMs in cloud environments). As a result, the end-to-end

performance model of the system can only be dynamically composed a system run-time.

- The deployment and configuration of applications may change frequently due to automatic or manual reconfigurations (e.g., deployment of new VMs, or migration of existing ones). As a result, the overall performance model of the system needs to be continuously updated to always reflect the current system architecture and configuration.

A major field of research is the automatic extraction of performance models based on static and dynamic analysis of the system implementation and configuration (see Section 3.2). Existing work either describes holistic approaches to extract complete performance models, but assume a very specific technology stack (Brosig et al., 2011; Brunnert et al., 2013), or focuses on improving certain aspects of it, such as, resource demand estimation. However, these approaches do not provide solutions for the described challenges.

Research Questions In this chapter, we describe a novel agent-based reference architecture for online model learning in virtualized data center addressing the challenges described previously. In particular, we consider the following research questions.

- How to integrate automatic model learning capabilities into existing virtualization platforms? We propose to extend the notion of VAs to include model learning capabilities and introduce additional auxiliary components into virtualization platforms.

- What capabilities are required to learn an end-to-end performance model of a system? What are the dependencies between these model learning capabilities? Each agent in our reference architecture is responsible for maintaining a certain sub-model of the end-to-end performance model. We identify different roles an agent may take over during model learning and describe the required communication between agents in different roles.

- How to compose sub-models representing knowledge about the system architecture into a single end-to-end performance model? Which structural constraints do these sub-models have to fulfill in order to be composable? The sub-models created by each agent are dynamically composed in a central model repository. We describe a merging algorithm for sub-models and determine the rules they need to fulfill to avoid conflicts.

Chapter Outline Section 4.1 introduces our proposed reference architecture for integrating model learning capabilities into virtualization platforms. Section 4.2 provides an overview of the different types of agents and describes how the individual agents collaborate with each other. Section 4.3 contains a formal description of our merging algorithm for sub-models as implemented in the central model repository. Section 4.4 concludes this chapter.

4.1 Reference Architecture for Online Model Learning

Modern hypervisors (e.g., VMware ESX or Xen) and virtualization management software (e.g., VMware vCenter) - in the following we call the combination of both the *virtualization platform* - rely on standardized formats for VM images to support the deployment of new VMs. However, this image format is focused on the specification of the virtual hardware resources including their configuration and lacks meta-data describing the platform and application layers inside a VM. The program code of the platform and application layers as well as any additional data is contained in an unstructured binary image of the virtual hard disk.

Therefore, a virtualization platform is generally not aware what is contained inside a VM. Although, the virtualization platform may access all data in the main memory and hard disks of a VM, the data is hard to interpret given that no general assumptions can be made on their structure. An approach to model learning solely based on information available in the virtualization platform inevitably leads to performance models abstracting application and platform layers as a black-box. In contrast, an approach based on model learning inside a VM may provide detailed performance models of the platform and application layers running in the same VM. However, in the latter case access to the underlying infrastructure layers or co-located applications is prohibited.

In the following, we describe our reference architecture for model learning that bridges this gap. First, we give an overview of the components of the proposed reference architecture in Section 4.1.1. Section 4.1.2 describes the communication primitives for agent collaboration. Section 4.1.3 introduces the concept of extraction scopes. This section is based in parts on our publications in Spinner et al. (2013) and Spinner et al. (2016).

4.1.1 Conceptual Overview

We argue that model learning capabilities should be integrated deeply in both the virtualization platform and the hosted VMs enabling the extraction of end-

to-end performance models covering the virtual infrastructure, as well as any platform and application layers within VMs. We assume a virtualization platform that hosts a set of *Virtual Appliances (VAs)*. A VA is a set of pre-packaged VM images each containing a complete software stack ready to run on a virtualization platform (see also Definition 2.4 on page 41). VAs can significantly reduce the effort and knowledge required for deploying software systems. VAs are either provided directly by software companies or by individuals. For instance, there are VAs available providing a pre-configured Tomcat application server or a Zimbra collaboration server. These VAs are built by experts of the respective system and can then be shared with others (e.g., through online marketplaces, such as VMware Solution Exchange[1]). When deploying such a VA, only certain pre-defined settings may need to be customized (e.g., through a web interface provided by the VA) in order to adapt it to a target virtual environment (e.g., IP address settings, or passwords).

Our reference architecture is based on an extension of conventional VAs to include additional logic for learning performance models of the application as well as any contained platform layers (e.g., middleware systems or Java VMs) during system run-time. The model learning logic is encapsulated in specialized *agents* distributed as part of a VA. On instantiation of such a VA in a virtualized environment, the contained agent will start to monitor the application serving real production workloads and will automatically build a sub-model (so-called *model skeleton*) describing the observed performance behavior of the application and platform layers inside the VA. The agent continuously updates the model skeleton to reflect dynamic changes, for instance, in the configuration or the workload of an application. A virtualization platform may access the model skeletons extracted by the agents of a VA using a defined interface in order to obtain fine-grained performance models of an application.

However, model skeletons created within a VA do not contain information about the underlying infrastructure layers, or co-located VAs as such information is not visible inside guest VMs. Therefore, the virtualization platform itself needs to contain agents that extract model skeletons of the data center and the virtualization platform. The virtualization platform then composes the model skeletons from different VAs and underlying infrastructure layers into an end-to-end performance model.

A VA may contain a complete application (e.g., an SAP ERP system, or a Zimbra Collaboration server), or provide only certain platform layers (e.g., a Java EE application server) on which custom applications can be deployed after VA instantiation. In the former case, the deployment of the VA typically

[1] `https://solutionexchange.vmware.com/store`

involves only certain customizations of the configuration and the creator of the VA may be able to determine large parts of the model skeleton in advance. In the latter case, no prior knowledge about the static structure and dynamic behavior of applications running on the platform layers can be assumed and the extraction logic needs to create the model skeleton dynamically by analyzing the executed application.

With our reference architecture, it is possible to specifically design agents for a given software stack contained in a VA. Thus it is possible to incorporate technology-specific prior knowledge into the model learning logic. The model skeletons may be partially or completely created at run-time based on dynamic system information obtained through sensor or reflection interfaces. Sensors provide empirical observations of the dynamic behavior of a system. Reflection describes the ability of a software system to determine its own structure and state (e.g., based on configuration files, byte-code, etc.). Both sensors and mechanisms for reflection on the application level are typically very technology-specific, and therefore are part of the VA.

System administrators, who deploy a VA in a virtualized data center, do not need to be an expert in performance modeling. The model learning runs transparently in the background without disturbing the system operation. The resulting end-to-end performance model of the virtualized system can be used for online resource management in conjunction with advanced reasoning techniques exploiting knowledge of the system architecture (e.g., see Chapter 6).

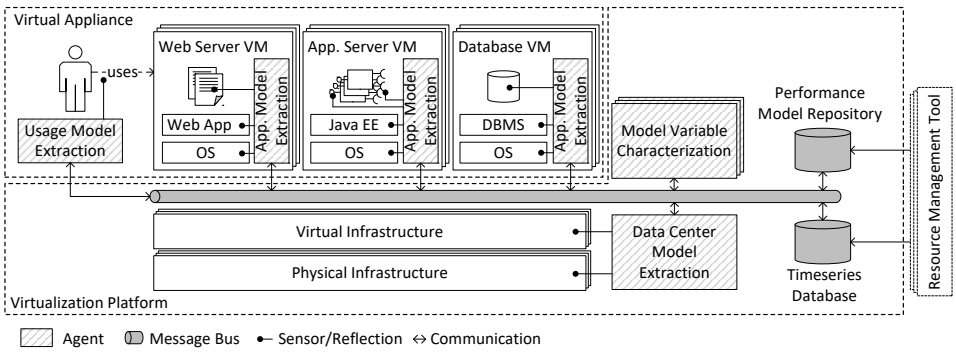


Figure 4.1: Conceptual overview of the reference architecture.

Components Figure 4.1 gives an overview of the main components of our reference architecture. Our reference architecture relies on specialized agents

focusing on learning models of certain aspects of a system. We see the following major roles for agents:

- Each VA contains one or several *application model extraction* agents creating models of the application architecture. This covers the components of an application including their behavior, their assembly and their deployment, as well as any platform layers running inside a VM. These agents only determine the model structure. They do not determine values for model variables (e.g., resource demands).

- The *usage model extraction* agent focuses on the behavior of external users of an application. It determines usage behaviors for different types of users and characterizes their load intensity. The agent needs to be able to observe incoming requests at the interface roles accessible from outside an application.

- The agents for *data center model* extraction are part of the virtualization platform, so that they have access to the virtual and physical infrastructure layers in a data center. The agents are specifically designed for the infrastructure technologies in a data center, and must not make any assumptions on the software stack within VAs.

- Agents for *model variable characterization* implement generic dynamic analysis techniques to determine the current value of model variables based on empirical data. The empirical data may be provided by the VAs (e.g., throughput or response time measurements) or the virtualization platform (e.g., resource utilization statistics). Model variable characterization agents may not make any assumptions on the software stack running in a VA. These agents need to derive the information they require from the model skeletons and monitoring data provided by the model extraction agents in the VAs and the virtualization platform.

For communication purposes, we assume that all agents have access to a shared network. The network connects them with the following central components:

- The *message bus* connects all components in our reference architecture and provides asynchronous messaging for communication purposes. We rely on publish/subscribe communication patterns in order to decouple the different agents from each other.

- The *performance model repository* contains the end-to-end performance model of the system. It is responsible of merging the model skeletons

coming from different agents in VAs and the virtualization platform into a consistent end-to-end model.

- The *time series databases* are used to collect and store historic monitoring data. Time series databases are optional, and only necessary if historic data needs to be persisted for longer periods of time.

Resource management tools may access the current model version for reasoning purposes. The resource management tools are not part of our reference architecture.

Meta-model The model skeletons and the end-to-end performance model share a common *meta-model* providing a formal definition of an abstract syntax. The meta-model enables us to create technology-independent descriptions of the system architecture. Furthermore, a common meta-model helps to enforce consistent syntactic and semantic constraints between model skeletons and simplifies their composition into an end-to-end performance model avoiding the need for model transformations.

Our reference architecture is based on an existing meta-model for online resource management, called Descartes Modeling Language (DML) (Kounev et al., 2016). Section 2.2.2 provides an overview of its meta-model. Compared to low-level prediction models (e.g., QNs), a descriptive meta-model provides the advantage of greater expressiveness to include additional information on the static architecture and dynamic behavior of a system. Compared to other descriptive architecture-level performance models, such as PCM (Becker et al., 2009) and SLAstic (Hoorn, 2014), DML provides a number of benefits:

- DML provides explicit modeling elements to describe the layering and configuration of the system environment. This is important to capture the structure of the underlying virtualized infrastructure.

- DML supports different levels of granularity to describe the behavior of services (i.e., black-box, coarse-grained, and fine-grained). The different granularity levels increase the flexibility when solving a model to trade-off between prediction time and accuracy (Huber et al., 2017, see).

- Model variables (e.g., resource demands or branching probabilities) and parameter dependencies can be marked as explicit or empirical in DML. Explicit model variables are assumed to have a fixed value (or a stochastic expression calculating a value based on input parameters). The value of

empirical model variables are determined at run-time based on monitoring data from the system. We exploit this modeling construct for our model skeletons.

Degrees of Freedom The design of model extraction agents for VAs allows for several degrees of freedom. In the following paragraph, we discuss the design decisions that need to be taken into account.

Functionality: In the simplest case, an agent just delivers a prepackaged model skeleton when the agent is started (e.g., capturing prior knowledge about an application). However, in many cases the model skeleton depends on the configuration of the operating system, middleware system (e.g., which components are deployed in a runtime container) or application (e.g., customizations in the application settings). An agent may use different static and dynamic analysis techniques to construct such a model skeleton at runtime.

Granularity: An agent may take on more than one role in the model extraction (see Section 4.2). While this increases the implementation complexity of the agent, it may be beneficial when integrating existing model extraction approaches (e.g., Brosig et al., 2011; Brunnert et al., 2013) into the reference architecture. The existing tool may be reused as a whole only requiring a transformation from its output format to a model skeleton (based on DML).

Genericity: It is the agent designer's decision how generic an agent is designed (i.e., how much technology-specific knowledge is included in it). Model extraction agents may often be specifically designed for a certain technology (e.g., certain JEE application server product) in order to be able to fully exploit proprietary instrumentation and reflection capabilities. Furthermore, a deeper understanding of the underlying technology also may be required for mapping them to concepts in DML (e.g., what is a component, or what is a container?). On the other hand, model variable characterization agents will typically be generic as they directly work on the DML model and empirical observations.

Distribution: The distribution of the agents in Figure 4.1 is just exemplary, and not prescribed by the reference architecture. For instance, model extraction agents may not be required for each VM of an application, if an existing system or application monitoring solution is used, such as Dynatrace (Rometsch and Sauer, 2008) or Kieker (Rohr et al., 2008b), that provides all required information for creating the model skeleton in a central place.

Deployment: The agents may be deployed in the same VM as the application itself or in dedicated VMs. For application model extraction agents, a deployment directly alongside the monitoring tool or the application itself may be beneficial (e.g., easier access to reflection interfaces or log files). Model variable

characterization agents may depend on computationally expensive algorithms (e.g., for resource demand estimation) and a deployment in a dedicated VM avoids negative impacts on the application performance.

Notification: Reconfigurations in the environment or changes in the workload may require updates to the performance model. The agents may either work in push or pull mode. In push mode, the agent exploits special notification mechanisms of the infrastructure or application software in order to be informed of changes. In pull mode, the agents check for changes in regular intervals.

4.1.2 Agent Collaboration

Agents in a VA may interact with the message bus and the model repository in the virtualization platform. Figure 4.2 gives an overview of the different types of interactions.

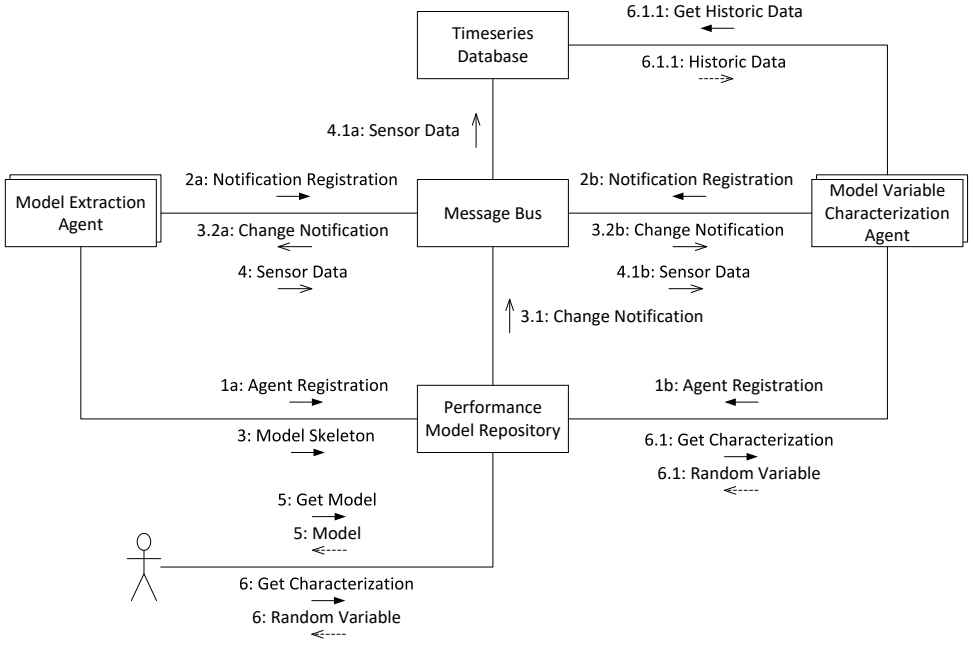


Figure 4.2: Communication diagram of agent interactions.

1. A newly started agent first registers itself with the model repository (messages $1a$ and $1b$), which maintains a list of all active agents. Each agent starts independently and no order is prescribed.

2. Each agent may register for certain model notifications at the message bus (messages $2a$ and $2b$). The registrations may be changed dynamically over the life-time of an agent.

3. An agent may send a model skeleton to the model repository at any time after its registration (message 3). The model repository updates the performance model with the model skeleton. It sends out model notifications for new, changed, or removed model objects informing all agents that have registered for notifications of that type (messages $3.2a$ and $3.2b$). Section 4.2 describes the different types of model notifications.

4. Agents send out monitoring data in regular intervals (message 4). The monitoring data is distributed through the message bus. Model variable characterization agents may listen for new monitoring data (message $4.1a$). Additionally, a timeseries database may store the monitoring data to provide historic data to model variable characterization agents (message $4.1b$).

5. A user may request the current performance model at any time (message 5). The returned performance model does not contain explicit characterizations of its model variables.

6. A user may request the current characterization of a model variable. The model repository should implement the interface introduced by Brosig (2014, p. 65), so that DML solvers can directly lookup the characterization of model variables (message 6). The performance model repository may then ask each model variable characterization agent for the current values of model variables (message 6.1). If the model parameterization agent does not have a recent value in cache, it needs to determine it. It may ask the timeseries database for historic data, if required (message $6.1.1$).

The following paragraphs describe the types of messages that are exchanged between agents and model repository in greater details. The messages are defined on a logical level. An implementation of our reference architecture needs to define how these messages are mapped to a technical protocol.

Agent Registration On start up, an agent registers itself at the model repository. An agent always needs to specify its extraction scope on registration (see Section 4.1.3). The scope then determines which parts of the model repository are accessible to an agent. System administrators define these scopes in the model repository in advance, before any agents can register with it. When

defining a new scope, the system administrator may also specify a token that agents need to provide during registration for authorization purposes.

Model skeletons A model skeleton represents the local view of an agent on the virtualized system. Different agents may be responsible for different parts of the system. For instance, an agent at the virtualization layer can determine the physical hosts and the VMs running on each host, but cannot see what is running inside a VM. This information needs to be provided by other agents that have access to the applications inside a VM.

Definition 4.1 (Model skeleton)**.** A model skeleton is a valid, self-contained and unparameterized model instance of DML. Valid means that a model skeleton needs to conform to all constraints specified by the DML meta-model. Self-contained means that it does not contain external references to objects outside of the model skeleton. Unparameterized means that all model variables in the model skeleton are marked empirical, i.e., their values are determined later based on monitoring data.

While a single model skeleton describing the complete system may be possible, we expect that in practice the end-to-end performance model of the complete system is the result of composing a set of model skeletons coming from different agents. We include the validity and self-containedness properties in the definition in order to facilitate the composition of model skeletons.

Model skeletons are assumed to be subject to discrete changes at irregular intervals (e.g., due to system reconfigurations). In contrast model variables (e.g., resource demands or branching probabilities) may change continuously over time. Therefore, we treat model variables separately in our reference architecture. In the model skeletons, model variables are always marked as empirical. The actual value of a model variable is determined on-demand, i.e., we determine the current value of a model variable, when the current model is requested, for instance, for performance predictions.

A model skeleton is described by a Meta Object Facility (MOF) compliant meta-model. Figure 4.3 gives an overview of this meta-model. A `ModelSkeleton` references six sub-models of the DML meta-model (see Kounev et al., 2016): `Repository`, `UsageProfile`, `System`, `ContainerRepository`, `DistributedDataCenter`, `Deployment`, and `AdaptationPointDescriptions`. These sub-models however contain only elements which are part of the local view of the agent, i.e., they are not a complete representation of the system. Therefore, all elements of a model skeleton are optional. The `Container` elements describes the resource layers within one or several VMs (e.g., middleware resources).

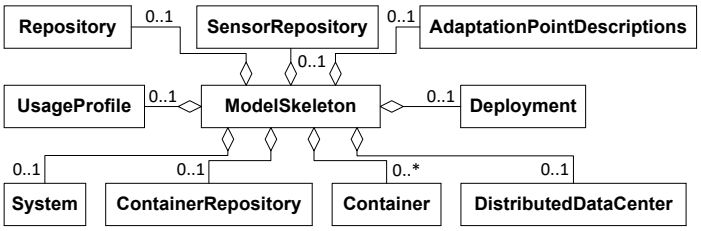


Figure 4.3: Model skeleton meta-model.

The `SensorRepository` contains information about the sensors in the system. Each agent needs to provide a description of the sensors for which it collects monitoring data. Figure 4.4 shows the corresponding meta-model (not part of DML).

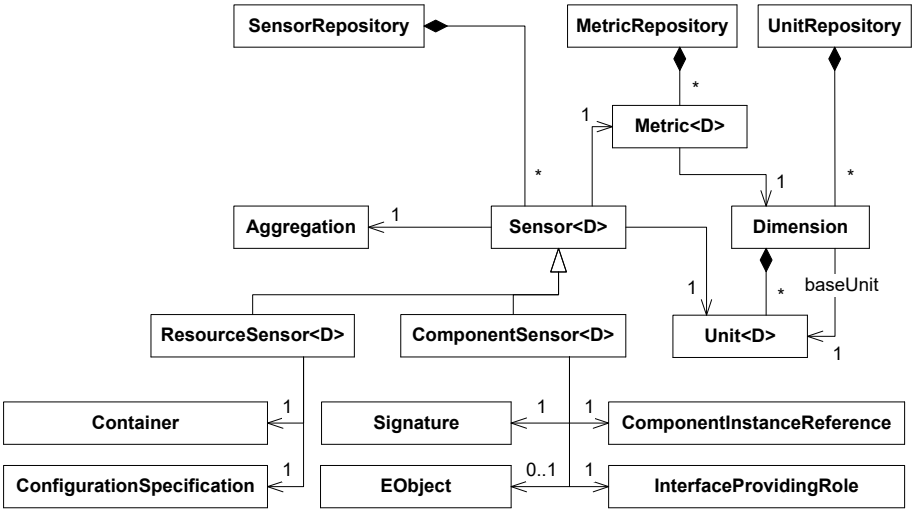


Figure 4.4: Sensor meta-model.

The `SensorRepository` contains a list of `Sensor` defintions. Each `Sensor` represents one instrumentation point in the real system, where monitoring data is collected. In general, a `Sensor` references a `Metric` (e.g., response time, throughput, or utilization), a `Unit` (e.g., seconds), and an `Aggregation` (e.g., mean, minimum, or sum). `Sensor`, `Metric` and `Unit` are generic classes param-

eterized with a sub-class of `Dimension` (e.g., `Time`). The sub-classes are not depicted for reasons of conciseness. `MetricRepository` and `UnitRepository` are not part of the model skeleton. These are global registries for standard metrics and units. Furthermore, the subclasses `ResourceSensor` and `ComponentSensor` specify where in the system architecture the data is collected (referencing corresponding model entities in the DML model).

Change Notification Change notifications are sent out if certain model elements are created, updated or deleted. In this section, we describe the general message structure of change notifications. See Section 4.2 for details on the different types of change notifications intended by our reference architecture. A change notification is an asynchronous message containing key-value data. We define the following two required keys: `TYPE` and `ENTITY_ID`. `TYPE` specifies the change operation that triggered the notification (`CREATE`, `UPDATE`, or `DELETE`). `ENTITY_ID` contains the identifier to a model element in the model repository that triggered the change notification. Additional keys may be added by an implementation of our reference architecture.

Sensor Data Agents send asynchronous messages to transmit current monitoring data to other agents or to a time-series database. A sensor data message contains key-value data. We define three required keys: `SENSOR_ID` contains the identifier of the sensor that produced the monitoring data. `TIMESTAMPS` contains a comma-separated list of floating-point values representing timestamps in milliseconds. `VALUES` contains a comma-separated list of floating-point values containing the observed values. The number of timestamp values must match the number of observed values.

4.1.3 Extraction Scopes

A single agent for model extraction typically does not have a global overview and insight of the services running inside a data center as well as the underlying infrastructure. The part of the system that is visible to a model extraction agent is defined by its extraction scope.

Definition 4.2 (Extraction Scope)**.** An extraction scope is defined by a tuple $scope = (ms, A)$. $ms \subseteq ms_{global}$ is the set of model elements contained in the scope (a subset of the end-to-end model ms_{global}) and A is a set of model extraction agents. Each agent belongs to exactly one extraction scope. Agents can access all model elements in their scope. Access to model entities outside the agent scope is restricted.

The extraction scopes in this thesis are a generalization of the model variable scopes proposed for DML by Brosig (2014). These scopes are used to specify the validity of empirical characterizations of a model variable (e.g., resource demands, or branching probabilities). We extend the notion of scopes in this thesis to also apply to structural model entities (e.g., components, or containers) as well. For instance, a component that is extracted in one application may not match with another component with the same name and interface roles in another application, because the two applications use different implementations of a component.

Types of Extraction Scopes Extraction scopes are typically prescribed by the system architecture. Certain aspects define the local view of model extraction agents and limit the validity of extracted model skeletons. We see the following aspects that typically define the extraction scope of agents:

- *Layered architectures* are used to abstract certain low-level layers from higher layers in order to hide complexity and increase portability. Communication between layers is only allowed through well-defined interfaces. Model extraction agents need to be aligned according to the existing layering and may not breach the layering. For instance, an agent that can access and obtain information from the hypervisor should not make any assumptions on the applications running inside the VMs so that it can be used for any type of VM. At the same time, an agent extracting the application architecture, should not depend on the underlying hypervisor technology. Agents at different layers in the system should only communicate through well-defined and technology-neutral interfaces.

- *Shared infrastructures* may host different applications from separate organizations or organizational units on the same hardware system. In such a setting, the isolation between these applications needs to be ensured, i.e., no information flow is allowed between these applications except through their public interfaces. As a result, agents from separate applications should be isolated as well. If an agent could access the architectural performance model of another application, this would violate the isolation between applications.

- *Heterogeneous infrastructures* limit the generality of certain parts of the architecture performance model. For instance, empirical resource demands are only representative in systems with the same hardware and software configuration. In case of significantly different configurations, resource demands need to be estimated separately for each configuration.

In order to cover these aspects, we distinguish between the following general types of extraction scopes:

- The *data center scope* represents the global view on a data center. It is concerned with the extraction of the hardware infrastructure level as well as the high-level topology of applications running in a data center. There is typically only one data center scope per data center.
- A *platform scope* comprises a platform layer (e.g., virtualization, operating system or middleware layers) on top of the hardware infrastructure level. We explicitly support the layering of platform scopes. This scope is responsible with the extraction of the configuration of the platform layer, its impact on the performance of applications, as well as the identification of directly contained platform layers. A platform scope may be either part of the virtualization platform or of a VA.
- A *usage scope* covers the extraction of the behavior of external clients of an application. External clients of the same application may be separated into different usage scopes.
- An *application scope* covers the extraction of the architecture of a single application. Each application scope corresponds to a `SubSystem` element in the global model.
- A *model variable scope* is responsible for the empirical characterization of model variables in the application architecture. Each model variable scope is bound to an application scope. Model variable scopes are separated as they are part of the virtualization platform and may have access to platform scopes (e.g., to obtain platform monitoring statistics).

Scope Delegation Given two agent scopes $scope_s$ and $scope_t$, where the former is called source scope and the latter target scope. The intersection $ms_s \cap ms_t$ is called the set of delegated model elements. These model elements are accessible by agents in both scopes. Changes to delegated model elements are subject to the following restrictions.

An agent in source scope $scope_s$ may explicitly delegate a model element $m \in ms_s$ to a target scope $scope_t$ so that $ms'_t = ms_t \cup C_m$. C_m is a set containing all model elements which are descendants of m in the containment tree. Now agents in $scope_t$ may access and update elements in C_m, however, they are not allowed to delete them. The same applies to agents in $scope_s$. If an agent in $scope_s$ wants to delete any of the model elements in C_m, it must explicitly cancel

the delegation to $scope_t$ first. Agents in $scope_t$ must listen for cancellations of delegated model objects and acknowledge it, before an agent in $scope_s$ can delete the delegated model object. If an agent in $scope_s$ wants to change a model element in C_m, it must send a model notification to $scope_t$.

The source scope does not know the target scope. The message bus is responsible for routing a delegated model element to a target scope. The routing rules are either statically provided by a system administrator, or the virtualization platform may dynamically derive them based on the current configuration.

Access Restrictions By default, an agent can only access model elements within its own scope. However, depending on the type of extraction scope, we may relax this rule to allow for read-only access to neighboring scopes:

- If a platform scope is part of the virtualization platform, it may access the data center scope, as well as any platform layer below itself. For instance, a platform scope covering the hypervisor layer, may obtain information on the hardware infrastructure.

- A model variable scope has access to the associated application scope, the data center scope, and any platform scopes the application depends on. For instance, the characterization of resource demands requires information on the physical and logical resources the application depends on.

4.2 Model Extraction Agents

In our reference architecture, model extraction agents may take on different roles. An agent role defines the aspects of the performance model an agent takes care of. In order to extract a complete model, all required roles need to be implemented by agents. However, there are certain dependencies between agent roles requiring coordination between agents. In this section, we define a set of standard agent roles and the communication channels between them.

We analyzed the DML meta-model identifying clusters of classes in the meta-model with a high cohesion. Such clusters should be covered by a single agent role in order to reduce the required communication between agent roles. In the following, we describe each agent role and its interfaces to other roles. The agent roles are grouped by extraction scopes. Furthermore, we also discuss how existing dynamic and static analysis techniques can be used to implement the agent roles.

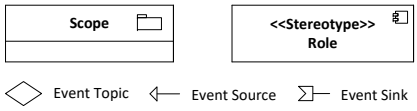


Figure 4.5: Notation.

Notation The communication between agents in our reference architecture is based on asynchronous, event-based communication. In the following, we adopt the UML component diagram notation with the profile for event-based communication used by Rathfelder (2013). Figure 4.5 gives an overview of the notation. Each agent role is represented by a component. Composite components are used to group them into extraction scopes. A component may have any number of event sources and event sinks specifying the types of events it may send and receive. Event topics connect event sources with event sinks and allow for publish/subscribe communication. Events are either change notifications (as described in Section 4.1.2), or scope delegations, if the event source is marked with the stereotype `«delegates»` (as described in Section 4.1.3).

4.2.1 Data Center Scope

Agents in the data center scope create and maintain a resource landscape model of the physical hardware infrastructure as well as a high-level system model of the applications running in a data center. The agents in this scope treat the physical hardware nodes and applications as black-boxes without any knowledge of their internal structure.

Agent Roles Figure 4.6 gives an overview of the agent roles in the data center scope. We distinguish six agent roles:

- *D1 (Data Center Structure).* The agent discovers the global structure of the data center. It identifies compute and storage nodes representing physical computers and storage systems in a data center. This agent role is focused on the static structure.
- *D2 (Compute Node Configuration)* The agent enriches compute nodes with information on their configuration (e.g., number of CPUs and their speed). Furthermore, it determines the directly contained runtime environment (e.g., a hypervisor, or a native operating system). We assume that each

physical compute node contains only one direct child container. This is a safe assumption for real systems, given that only a single hypervisor or operating system can run natively on a compute node.

- *D3 (Storage Node Configuration, optional)* The agent extracts the configuration of storage nodes. The current version of DML does not provide meta-models to describe storage nodes (Huber, 2014). The preliminary work of Noorshams (2015) for storage performance modeling may be integrated as part of future work.
- *D4 (Network Infrastructure, optional)* The agent extracts the physical network infrastructure connecting compute and storage nodes in a data center. The current version of DML does not provide meta-models to describe network infrastructures (Huber, 2014). The work of Rygielski and Kounev (2014) for network performance modeling and extraction may be integrated here as part of future work.
- *D5 (System Interface Providing Roles)* Applications may provide services to users outside the data center. An agent with this role is responsible to identify these interface roles including their operations and their input and output parameters.
- *D6 (Application Assembly).* The agent determines the different applications running inside a data center including their interface providing and requiring roles. The interface providing roles of an application describe the services which are publicly visible to other applications in the same data center or to external users. The interface requiring roles of an application may be connected to services provided by other applications in the same data center. Interface providing roles of applications may be delegated to the system level if they are visible from outside the data center.

Scope Delegations Figure 4.6 shows the delegation interfaces to other agent scopes. We define the following three delegation interfaces in the data center scope:

- The role *D2* determines the direct child runtime environment (e.g., a hypervisor, or an operating system) of a compute node. However, the extraction of the internal structure and behavior requires deeper insights into the runtime environment. Therefore, the runtime environment may be delegated to a platform scope. The target platform scope may be determined dynamically (e.g., depending on the type of hypervisor). If

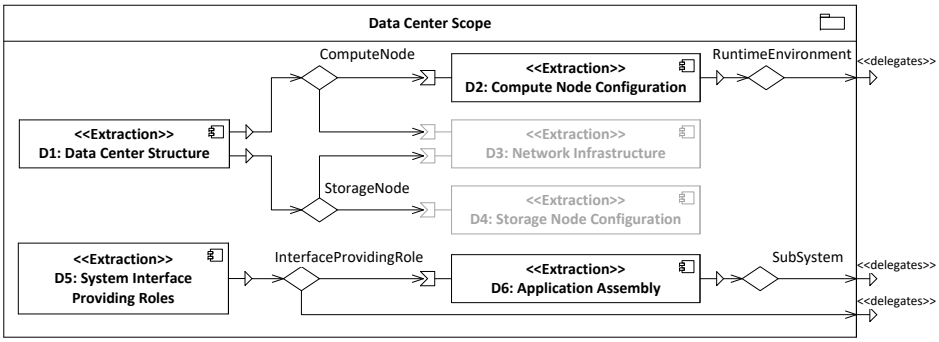


Figure 4.6: Overview of the data center scope.

applications run natively on the operating system, the runtime environment may be also delegated directly to an application scope. Each runtime environment requires a name that uniquely identifies it within a data center. For instance, a unique IP address or host name may be used as name.

- Interface providing roles identified by role *D5* can be delegated to usage scopes. Agents in the usage scope then determine how external users are accessing the interface providing role. We suggest the use of protocol-specific addresses, such as Uniform Resource Locators (URLs), as unique names for interface providing roles.

- The applications discovered by role *D6* are represented as subsystems. These subsystems are delegated to application scopes for further extraction of the application architecture. The name of the subsystem determines the target application scope.

Change Notifications Figure 4.6 shows the flow of notifications within the data center scope. *D1* notifies *D2*, *D3*, and *D4* when new compute nodes or storage nodes are discovered, or old ones are deleted. *D5* informs *D6* if a new interface providing role is created or if the set of operations provided by an existing one changes.

Existing Approaches Traditional system or network management software, such as IBM Tivoli or Hyperic, and virtualization management software, such as VMware vCenter, are commonly used to manage data centers. Such software

typically maintains an inventory of the systems and applications as well as their topology in a data center. Proprietary or standardized management interfaces, such as Simple Network Management Protocol (SNMP), or Common Information Model (CIM), are available to access that information.

4.2.2 Usage Scope

The usage model captures the external requests of an application. In our context, external requests come from outside the data center. Load from other applications in the same data center is covered by agent role *D6*. Compared to the workload characterization step in classic performance modeling, we only cover the arrival process in the usage model and do no consider the mapping to resources as part of the usage scope.

In order to extract usage models we require empirical information on the type and frequency of requests. In case of session-based workloads, we also need a session identifier for correlating requests in the same session. Furthermore, if call parameters are considered for parameter dependencies, the monitoring needs to log the values of input parameters of individual requests. We assume that agents in the usage scope have access to session logs containing the required information.

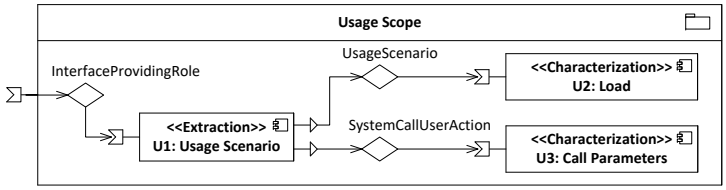


Figure 4.7: Overview of the usage scope.

Agent Roles Figure 4.7 gives an overview of the agent roles in the usage scope. We distinguish three agent roles:

- *U1 (Usage Scenario).* Users of a system may differ in their usage behavior (i.e., number and types of system calls) and the load they cause on the system (i.e., load intensity and open vs. closed workloads). Usage scenarios group users with similar behavior and load characteristics. The agents categorizes observed user sessions into different usage scenarios and determines the usage behavior (i.e., the sequence of requests) for

each usage scenario. The agent also determines probabilities for each request.

- *U2 (Load).* Performance models distinguish between open and closed workloads. Open workloads are characterized by the arrival rate, while closed ones are defined by the number of concurrent users and an optional think time. The load intensity of a system is often time-dependent with seasonal patterns, trends and bursts. The agent needs to characterize the workload type as well as the load intensity over time.
- *U3 (Call Parameters, optional).* The performance behavior of an application may depend on the value of input parameters. In order to extract such parameter dependencies, we need to characterize the values of such parameters for individual requests. Given that parameters may have many different values, techniques to determine groups of parameter values with similar performance impact are required.

Scope Delegations *U1* agents should listen for incoming interface providing role events (see Figure 4.7). These events are delegated from a data center scope when new services are deployed in a data center.

Change Notifications Figure 4.7 shows the flow of notifications in a usage scope. When *U1* identifies a new usage scenario, it informs *U2* of it. *U2* agents then can start observing the load intensity of that scenario. If the behavior definition of a usage scenario changes, *U1* triggers corresponding update notifications for *U2* agents. Each new or removed system call user action identified by a *U1* agent triggers a notification for *U3* agents. *U1* should also trigger an update notification if the call parameters of an operation associated with a system call action changes.

Existing Approaches Table 4.1 gives an overview of existing approaches to usage model extraction. Session logs can be collected either on the server or on the client. On the server side, access logs (e.g., on a web server) are often available containing the required information. In recent years, user monitoring on the client side becomes increasingly popular (e.g., using javascript instrumentation of web pages, such as Google Analytics). Numerous approaches to web mining (see Liu and Keselj, 2007) have been proposed in the literature. However, such techniques are focused on the general analysis of user behavior.

Sharma et al. (2008) uses Independent Component Analysis (ICA) to classify user requests according to their resource needs. However, the approach does

Approach	U1	U2	U3
Web mining techniques (e.g. Liu and Keselj, 2007)	✗		
Sharma et al. (2008)	(✗)		
CBMG (Menascé et al., 1999)	✗		
WESSBAS (Hoorn et al., 2015)	✗		
LIMBO (Kistowski et al., 2015)		✗	
Brosig et al. (2011)			✗

Table 4.1: Existing approaches to usage model extraction.

not consider sessions consisting of several requests. Thus it only covers agent role *U1* partially. Hoorn et al. (2014) and Menascé et al. (1999) consider the extraction of usage models for performance prediction. Menascé et al. (1999) proposes a modeling formalism called CBMG to describe user behaviors. They employ clustering techniques to determine different types of user sessions and reconstruct the usage behavior (see *U1*) by analyzing the the sequence and timings of observed sessions in a session log. Hoorn et al. (2014) published a tool, called WESSBAS, based on similar techniques to extract usage behaviors for load testing.

Kistowski et al. (2015) proposes an approach, called LIMBO, to extract models describing the temporal development of load intensities. The approach employs signal processing techniques (e.g., Fourier transformations) to identify seasonal patterns, trends, bursts and noise. Brosig (2014) considers supervised learning techniques to group values of input parameters according to their performance impact.

4.2.3 Platform Scope

On top of the physical hardware layer (represented by a data center scope), data centers typically have one or several platform layers representing the hypervisor and optional middleware layers required by applications. Each platform layer may host agents that extract models describing its structure and behavior.

Agent Roles Figure 4.8 gives an overview of the agent roles in a platform scope. We distinguish three agent roles:

- *P1 (Platform Configuration).* The agent analyzes a run-time environment (e.g., a hypervisor, or a middleware system) and determines its logical software resources (e.g, logical CPUs, thread and connection pools, or

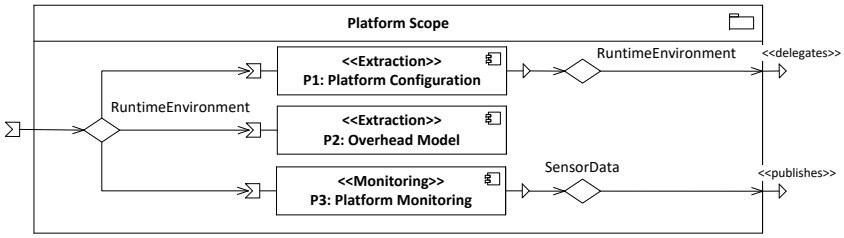


Figure 4.8: Overview of the platform scope.

asynchronous message queues), as well as any contained run-time environments (e.g., VMs). The agent extracts the current configuration of software resources of the run-time environment (e.g., the resource capacity, or scheduling priorities). This role is focused on structural aspects.

- *P2 (Overhead Model, optional).* The agent determines the dynamic behavior of a run-time environment, i.e., its performance impact on higher layers in a system. For instance, hypervisors introduce certain overheads slowing down the processing within a VM (Huber et al., 2011). In heterogeneous data centers with different types of physical nodes or varying hypervisor configurations, we need models describing the overhead if we want to predict the expected performance after a horizontal scale-out or a VM migration to another physical node. Furthermore, resource-intensive reconfigurations may impact the performance of VMs in a data-center. We also consider that as a type of overhead of which explicit models may be extracted.

- *P3 (Platform Monitoring, optional).* Platform layers often provide numerous monitoring statistics at system run-time covering the state of physical and logical resources (e.g., the current resource usage). The agent exposes this information in a technology-independent way.

Scope Delegations *P1*, *P2*, and *P3* agents react to incoming runtime environments which are delegated either by a data center scope or an underlying platform scope. We support an arbitrary layering of platform scopes to reflect the layered architecture of many data centers (e.g., different virtualization, operating system, and middleware layers). If a *D1* discovers a child runtime environment for which it cannot determine its internal structure, it delegates it to the next platform scope. For instance, a platform scope covering the hy-

pervisor may delegate run-time environments representing VMs for further processing to a platform scope knowing the middleware layers inside a VM. The highest platform scope delegates runtime environments to an application scope.

Existing Approaches The extraction of platform layers is highly technology-specific. We limit our discussion to hypervisors. Virtualization management software (e.g., VMware vCenter) provide access to the hypervisor configuration to well-documented interfaces. These interfaces are useful to implement *P1* agents. In addition, they typically also provide comprehensive monitoring capabilities supporting *P3* agents. However, they contain purely descriptive models without support for predictive analyses.

The extraction of hypervisor overheads (see role *P2*) is an active research field. Huber et al. (2011) uses micro-benchmarks to determine the performance impact of certain hypervisor configurations. Lu et al. (2011) employ directed factor graphs with regression analysis techniques in order to map the resource usage statistics observed within a VM to corresponding statistics at the hypervisor level including hypervisor overheads.

4.2.4 Application Scope

The application model extraction covers the static structure and dynamic behavior of an application. We assume a component-based software architecture. The agents in these roles focus on determining all possible control flow paths in a component and do not determine the value of control flow variables (e.g., branching probabilities, or external call frequencies) and resource demands. These model variables are characterized by downstream agents in model variable scopes using empirical observations.

Agent Roles Figure 4.9 gives an overview of the agent roles in the application scope. We distinguish eight agent roles:

- *A1 (Component Boundary).* The agent discovers the software components an application consists of. This includes the interface providing and requiring roles of the components as well as interface definitions (i.e., signatures and parameters). The components are on a type level, they may be used in different assembly contexts within an application. Only one component definition is created even if it is used in different assembly contexts. By default, the agent creates a black-box service behavior

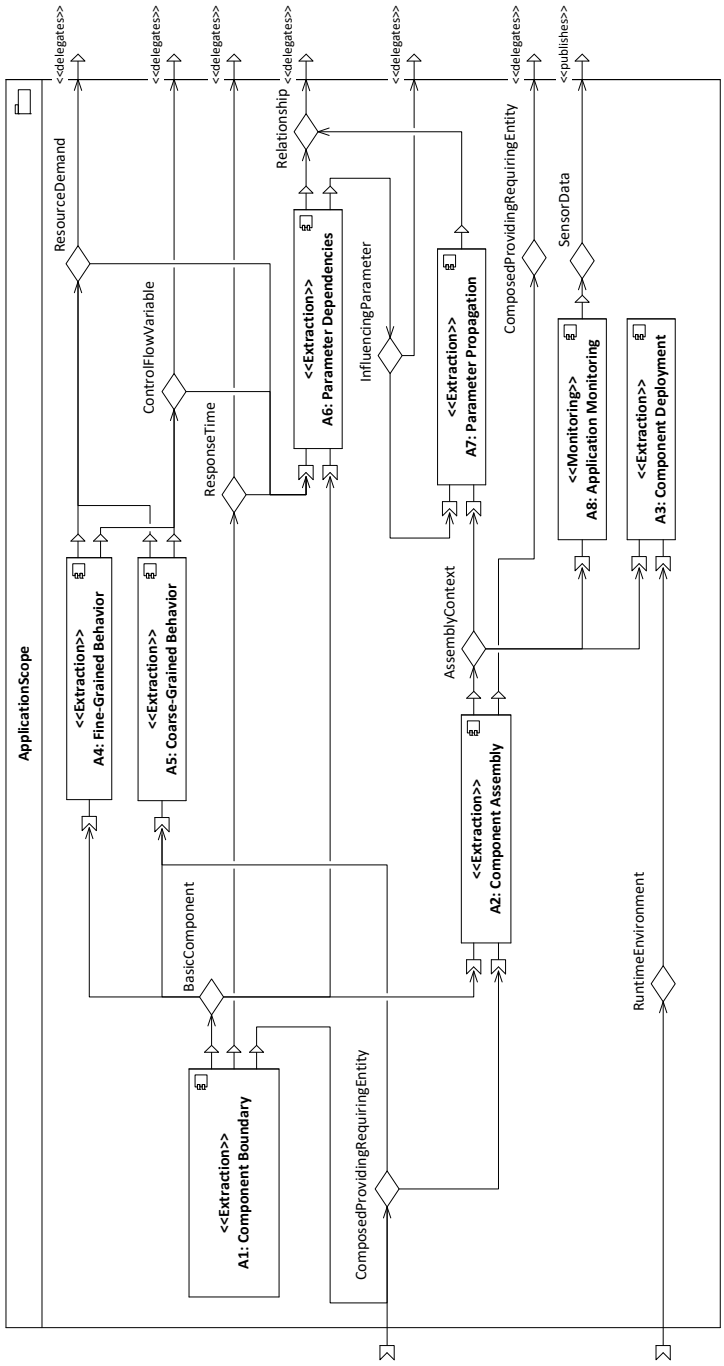


Figure 4.9: Overview of the application scope.

description for each component service. In addition, *A4* or *A5* agents may add more detailed descriptions of the service behavior later.

- *A2 (Component Assembly).* The component assembly describes the composition of components in an application. The agent discovers all component instances in an application and determines the control flow between them.
- *A3 (Component Deployment).* The agent determines the deployment of component instances on runtime environments.
- *A4 (Fine-Grained Behavior, optional).* The agent determines individual actions the component-internal control flow consists of (e.g., internal actions, forks, loops, and branches). It also determines their exact execution order.
- *A5 (Coarse-Grained Behavior, optional).* The agent determines which resources are accessed and which other components are called within a component service irrespective of the order.
- *A6 (Parameter Dependencies, optional).* Certain model variables (e.g., resource demands or branching probabilities) may depend on the value of an input parameter. The agent identifies possible parameter dependencies within a single component (e.g., which input parameters have an influence on the resource demand of a component).
- *A7 (Parameter Propagation, optional).* The agent determines the data flow of input parameters across components in an application.
- *A8 (Application Monitoring).* The agent collects performance statistics (e.g., response times and throughput of services) using application instrumentation techniques. The availability and access to this monitoring data usually depends heavily on the application and the employed implementation technologies.

Scope Delegations The application scope has two incoming event ports for composed providing requiring entities and runtime environments. Composed providing requiring entities may be either a subsystem or composite component. A source of subsystems is the agent role *D6* in a data center scope. Then a subsystem represents an application in the data center. If required, it is possible to nest application scopes (e.g., to encapsulate tiers into own application scopes). Then the outer application scope may delegate subsystems or composite components to inner application scopes (see the corresponding outgoing

port), so that the inner one extracts the internal structure of them. Runtime environments may come from data center or platform scopes and are required by role *A3* to determine the deployment target of components.

Each new model variable (i.e., resource demand, control flow variable, response time, relationship, or influencing parameter) created by an agent in the application scope is passed to a model variable scope. The model variable scope is then responsible to provide an empirical characterization of this model variable at run-time. The application scope also publishes sensor data. Model variable scopes may exploit this data for the characterization of a model variable.

Change Notifications *A1* agents notify *A2*, *A4*, *A5*, and *A6* agents of new basic components as well as any changes to component services of existing components (i.e., new or removed interface roles, signatures, and call parameters). If available, *A4* and *A5* then start to extract service behaviors for the new or updated component services. If *A4* or *A5* agents create new resource demands or control flow variables in a service behavior, they also inform *A6* agents of the new model variables. Furthermore, *A1* agents by default create a black-box behavior for each new component service and notify *A6* agents of the corresponding response time variable.

A6 agents uses the information on new or updated components, and on new model variables to keep track of potential influencing parameters and their influence. If the *A6* discovers a new influencing parameter, it informs the *A7* agents, so that they are aware of any new candidate targets for a propagation dependency. If an *A1* agent detect a new, updated or removed composite component or subsystem, it notifies *A2* and *A5* agents, so that they can extract the inner component assembly or the coarse-grained service behavior description. If an *A2* agent finds a new, updated or removed assembly contexts, it notifies *A3*, *A7* and *A8* agents. *A8* needs to be aware of new component instances as it might need to adapt the application instrumentation accordingly.

Existing Approaches A broad set of static or dynamic analysis techniques are available for application model extraction. Table 4.2 gives an overview of existing approaches to application model extraction. Awad and Menascé (2014) and Israr et al. (2007) extract QNs and respectively, LQNs models. These models are not component-based, therefore they do not fulfill roles *A1*, *A2*, and *A3*. Awad and Menascé (2014) only determine call frequencies, whereas Israr et al. (2007) use traces of individual transactions to determine more fine-grained control flows.

Approach	A1	A2	A3	A4	A5	A6	A7
Awad and Menascé (2014)					✗		
Israr et al. (2007)				✗			
SoMoX (Krogmann, 2010)	✗	✗		✗		✗	
Brosig et al. (2011)	✗	✗	✗	✗			
PMW (Brunnert et al., 2013)	✗	✗	✗	✗			
SLAstic (Hoorn, 2014)	✗	✗	✗		✗		
PMX (Walter, 2015)	✗	✗	✗	✗			

Table 4.2: Existing approaches to application model extraction

SoMoX (Krogmann, 2010) uses a combination of static and dynamic analysis techniques to extract PCM instances. It requires access to the source code of an application, and uses clustering techniques on code metrics to identify components as well as the component assembly. In order to determine parameter dependencies, the application is executed in a dedicated test environment and genetic search techniques are employed. Given that the approach is targeted at model extraction at design time, it lacks the information where the application will be deployed at run-time.

Brosig et al. (2011), PMW (Brunnert et al., 2013), SLAstic (Hoorn, 2014) and PMX (Walter, 2015) are exclusively based on dynamic analysis techniques of applications. Brosig et al. (2011), PMW and PMX are focused on the extraction of PCM instances, while SLAstic uses its own meta-model (although a transformation to PCM exists). The main difference between the four approaches is the monitoring tools used for obtaining the input for the dynamic analysis. Brosig et al. (2011) is based on the proprietary instrumentation techniques provided by the Oracle WebLogic middleware platform. PMW uses standardized Java EE filters to intercept incoming requests or alternatively can exploit session data from the Dynatrace APM tool[2]. SLAstic and PMX are based on the Kieker application monitoring framework (Rohr et al., 2008a).

4.2.5 Model Variable Scopes

The techniques to characterize model variables are typically generic, however, they may require access to information in the data center scope and the platform scopes. Therefore, agents for the characterization of model variables run in scopes separate to the application scope. Such agents may also use statistical techniques based on empirical monitoring with higher computational complexity and may be deployed on isolated machines.

[2] `http://www.dynatrace.com/de/index.html`

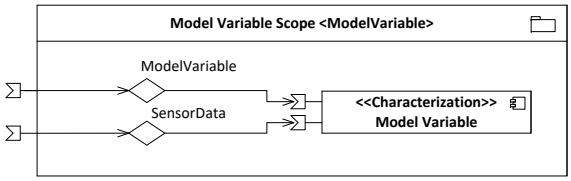


Figure 4.10: Overview of the model variable scope.

Agent Roles Figure 4.10 gives an overview of the model variable scope. The scope definition is a generic template where the `ModelVariable` parameter determines the actual variable type: `ResponseTime`, `ControlFlowVariable`, `ResourceDemand`, `InfluencingParameter`, or `Relationship`. We distinguish between five agent roles accordingly:

- *M1 (Response Time).* Black-box service behaviors contain a function describing the response time of a component service depending on input parameters. The agent derives a function describing the response time depending on the values of input parameters.

- *M2 (Resource Demand).* Resource demands are required for coarse-grained and fine-grained service behaviors. The agent determines a value for resource demands (including their stochastic distribution). Optionally, a resource demand may have a dependency on the value of an input parameter.

- *M3 (Control Flow Variable).* The agent determines values for control flow variables, such as loop iteration counts or branching probabilities in fine-grained service behaviors and external call frequencies in coarse-grained ones. The value of control flow variables may depend on values of input parameters.

- *M4 (Parameter Characterization).* In order to enable the characterization of parameter dependencies, the distribution of values of input parameters to a component service needs to be determined.

- *M5 (Relationship)* The agent determines the data flow of input parameter values between components in an application and provides empirical distributions of these relationships.

Scope Delegations The incoming model variable port may be connected to any model variable port of an application scope. The incoming sensor data port may be connected to any application scope and any platform scope. This port may connected to multiple scopes.

Approach	M1	M2	M3	M4	M5
Courtois and Woodside (2000)	✗				
Westermann et al. (2012)	✗				
LibReDE		E			
Wang et al. (2015)		E			
Israr et al. (2007)			✗		
SoMoX (Krogmann, 2010)			✗	✗	✗
ByCounter (Kuperberg et al., 2008a)		M			
Brosig et al. (2011)		E	✗	✗	✗
PMW + LibReDE (Brunnert et al., 2013)		M/E	✗		
SLAstic (Hoorn, 2014)			✗		
PMX + LibReDE (Walter, 2015)		E	✗		

Table 4.3: Existing approaches to model parameterization (E stands for estimation and M for measurement).

Existing Approaches Table 4.3 shows major existing approaches to model parameterization that may be used to implement such agents. Courtois and Woodside (2000) and Westermann et al. (2012) propose regression techniques to determine functions on the observed response time. LibReDE is our tool for resource demand estimation presented in Section 5.3, which has also been integrated with the PMW (Brunnert et al., 2013) and the PMX (Walter, 2015) tools. Brunnert et al. (2013) also implement a measurement-based approach for resource demands using application instrumentation. Further approaches using resource demand estimation techniques are Wang et al. (2015) and Brosig et al. (2011). The ByCounter (Kuperberg et al., 2008a) approach uses fine-grained instrumentation to count bytecode instructions and micro-benchmarks to measure the resource demand of individual instructions.

Israr et al. (2007), Brosig et al. (2011), PMW (Brunnert et al., 2013), PMX (Walter, 2015) and SLAstic (Hoorn, 2014) are all using dynamic analysis techniques to determine the control flow of applications, including a characterization of control flow variables. In contrast, SoMoX (Krogmann, 2010) uses static analysis techniques to reach that goal. Combined with dynamic analysis techniques, SoMoX can also characterize parameter dependencies using explicit stochastic expressions. Brosig et al. (2011) characterizes parameter dependencies using empirical distributions.

4.3 Performance Model Repository

The model repository is the central place where the extracted DML model instance of the complete system is maintained and persisted. The model instance is the result of merging all model skeletons coming from different agents into a single model instance. Model skeletons from different agents may overlap, i.e., they may contain model objects referring to the same physical entity. To avoid duplication of model objects, a merging of the model skeletons is required resolving duplicate model elements to a single element in the performance model repository.

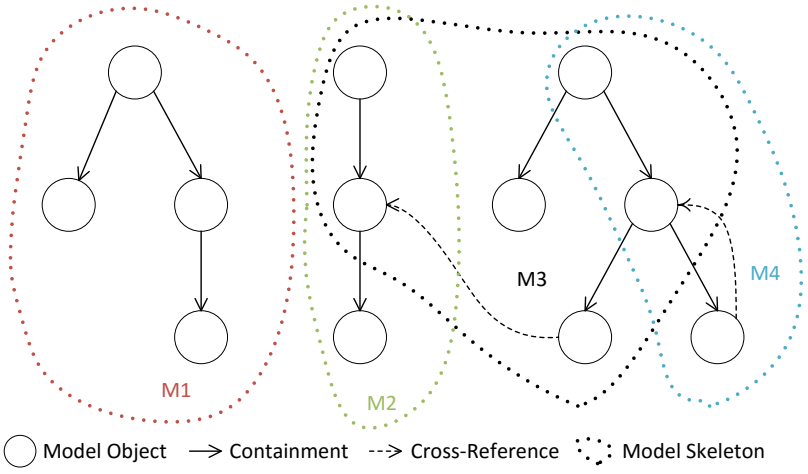


Figure 4.11: Example of overlapping model skeletons.

Figure 4.11 gives a schematic example how a complete DML model may be distributed across different model skeletons. In general, an EMF model may be seen as a forest of containment trees. Each model object has exactly one parent object (or zero in case of root objects). Cross-references may connect objects in different positions in the same containment tree or between trees. Model skeletons always start at root objects of DML and contain a subset of their descendant model objects. Given the self-containment property of model skeleton, all ancestors of a model object need to be contained in the same model skeleton. Furthermore, all targets of cross-references are part of the same model skeleton.

Given that model skeletons are created independently from each other, conflicts and inconsistencies need to be detected and resolved when merging them

into a single model instance in the performance model repository. In the following sections, we describe how model skeletons can be merged to an end-to-end performance model.

4.3.1 Model Skeleton Composition

Existing work on merging EMF models is focused on use cases where different users work concurrently on the same model instance. Förtsch and Westfechtel (2007) provide an overview on the state-of-the-art in this area. Westfechtel (2014) distinguishes three types of merge algorithms for EMF models: raw, two-way, or three-way. Given two versions of a model changed independently, raw merging takes one of the two versions discarding the changes in the other one. Two-way merging compares the different versions to detect differences between them. The merge algorithm itself cannot handle these differences automatically and typically requires manual intervention by a user. Three-way merge also takes a common base version into consideration. A base version is a common ancestor version from which the different model versions were derived. By exploiting the additional knowledge of the base version, three-way merging can automate the resolution of certain types of conflicts. However, there exists no domain-independent merging algorithm for models that can resolve all types of conflicts automatically (Förtsch and Westfechtel, 2007).

In our use case, we want to merge a model skeleton into the global model instance in our model repository. Given that the model skeleton is created and maintained independently from the global model, we cannot assume a common base version. Therefore, we have to resort to a two-way merging algorithm.

Formal Definitions

For the following descriptions, we adopt the formalization of EMF models used in Westfechtel (2014). We give a short repetition of this formalization here. We refer the reader to Section 2.2.1 for an introduction to EMF.

Definition 4.3 (Meta-model)**.** A meta-model "is a tuple $em = (C, D, F, P)$. The components of which are sets of classes, data types, features and properties, respectively. F is partitioned into sets of attributes A and references R ($F = A \cup R, A \cap R = \emptyset$)"(Westfechtel, 2014). P consists of attribute functions describing the properties of classes, data types and features (such as $superTypes$, $domain$, $range$, $many$, $ordered$, $unique$, $containment$, and $opposite$).

A model consists of a set of objects which are instantiations of classes in a meta-model. Each object contains values for the features defined in its class.

Attributes have literal values defined by their data type. References link to other objects in the model.

Definition 4.4 (Model)**.** "Let $em = (C, D, F, P)$ denote a meta-model. A model instantiated from em is a tuple $m = (O, class, FV)$. O is a set of objects. $class$: $O \rightarrow C$ assigns to each object the class from which it was instantiated. For each feature $f \in F$ there is one and only one feature value function $fv \in FV$."(Westfechtel, 2014)

Algorithm Overview Suppose a model skeleton m_s and the global model m_p in our model repository, the goal is to merge the contents of m_s into the existing structure of m_p. We use a state-based, two-way merging algorithm. State-based means that it does not require a complete change history; the merging is done solely based on the current state of m_s and m_p. We start with an empty m_p. Each new or updated model skeleton sent by an agent to the model repository is then merged into the current state of m_p in an atomic transaction. The merge algorithm needs to address the following two steps:

- *Differencing:* Find the model objects that are contained in both models. Given that the model skeletons are created independently by different agents, we need to consider strategies for *matching* the same objects in m_s and m_p.
- *Merging:* If a model object is only contained in m_s it can be simply copied to m_p. Otherwise, we need to merge the two objects to integrate any changes of m_s into m_p.

Compared to other merging algorithms (Westfechtel, 2014, e.g.) that assume the two model versions come from a common base version, our algorithm differs as the model skeleton m_s only represents a subset of m_p. As a result, it is not possible to determine model elements deleted from m_s solely on the current state of m_s and m_p. We need to be able to reconstruct the last version of m_s merged into m_p. Therefore, the model repository maintains for each model object in m_p a list of model skeletons that contain m_p.

Object Matching DML defines the abstract base class `Identifier` with a string attribute `id`. The `id` is used to uniquely identify a model object on a global level in a DML model instance. This class is implemented by many classes in the meta-model. DML does not apply any restrictions on the format of the `id` attribute, and it is typically generated automatically. However, the `id` attribute is not suitable for matching model objects due to the following reasons:

- *Traceability:* Model objects referring to the same physical entity may have different values for the `id` attribute if created by different model extraction agents. Often, the `id` attibute is just a randomly generated Universally Unique Identifier (UUID).

- *Coverage:* Not all classes in DML are subclasses of `Identifier`. Only model objects that are destinations of cross-references are subclasses of `Identifier`.

Instead of the `id` attribute, we use matching rules specific to the DML meta-model. These rules specify a set of identifying features (i.e., attributes or references) for each class in the DML meta-model. The values of these features need to be unique only on a local level (i.e, between siblings in the containment tree). The values of the `id` attribute in a model skeleton are always ignored when merging them into the model repository.

Conflict Prevention Conflicts may occur if two different model skeletons contain the same elements. Two models em' and em'' are conflicting, if they contain a model object $o \in (O' \cap O'')$ with a feature f for which the feature value functions fv'_f and fv''_f assign different values. The feature may be an attribute, a containment or a cross-reference. The equality of feature values depends on the feature attributes. For single-value features, the two values are compared directly. For unordered many-value features, we check for set equality. For ordered multi-value features, we also compare the sequence of values.

Given two versions of a model, which were changed independently of each other, there exists no domain-independent merging algorithm for models that can resolve all types of conflicts automatically (Förtsch and Westfechtel, 2007). Therefore, we need to introduce additional constraints to prevent conflicts. The basic idea is to allow sharing of model objects between different model skeletons only if we can be sure that any conflicting changes to these objects can be resolved automatically. To enforce this constraint, we rely on the agent roles introduced in Section 4.2. In Section 4.3.2, we describe the conditions that need to be fulfilled to allow automatic merging of model objects.

4.3.2 Merge Algorithm

We now derive a formal description of the state of the model repository and the model skeletons. These descriptions are focused on model merging and are not complete formalizations. They assume the availability of a common meta-model $em = (C, D, F, P)$, which is DML in our case.

Definition 4.5 (Model Skeleton)**.** A model skeleton $m_s = (s, b, id, owns)$ contains a model $s = (O_s, class, FV)$ conforming to the meta-model em. The element $b \in B$ specifies the agent that created the model skeleton. In addition, it provides a function $id : C \rightarrow \text{SET}(F)$ that returns a set of identity features used for matching elements. The function $owns : O_s \rightarrow boolean$ specifies whether an object is owned ($true$) or only referenced ($false$) in the model skeleton.

Several model skeletons are merged into a central model repository. The state of a model repository is defined as:

Definition 4.6 (Model Repository)**.** The state of a model repository is defined by a tuple $m_p = (p, B, shared, refs, owner)$ containing a model instance $p = (O_p, class, FV)$ conforming to the meta-model em. The set B contains all currently connected agents. The function $shared : O_p \rightarrow boolean$ specifies whether multiple owners are allowed for an object. The function $refs : O_p \rightarrow SET(B)$ returns a set of agents that reference a model object. The function $owners : O_p \rightarrow B$ determines the set of agents that own a model object.

The merging is based on a two-ways merging algorithm: the model versions are s and p. It is important to note that compared to traditional merging algorithms, s is only a subset of p.

Differencing. The first step, is the differencing to determine the set of changes in the model skeleton m_s that need to be merged into the model repository m_p. We define a helper function $matches : O_s \times O_p \rightarrow boolean$. The function evaluates to true if the following condition for two objects $o_1 \in O_s$ and $o_2 \in O_p$ with $c = class_s(o_1) = class_p(o_2)$ is fulfilled: $\forall i \in id(c) : fv_i(o_1) = fv_i(o_2)$. The results of the differencing step are the following three sets (b is the agent that created model skeleton m_s):

- The new objects set contains all objects in the model skeleton which have no matching counterparts in the repository.

$$\Delta_{new} = \{o_s \in O_s \mid \forall o_p \in O_p : \neg matches(o_s, o_p)\}$$

- The existing objects set contains all objects newly added to a model skeleton which already have matching counterparts in the repository. This may be the case if another agent has already created the same object in the repository.

$$\Delta_{exists} = \{o_s \in O_s \mid \exists o_p \in O_p : (matches(o_s, o_p) \wedge b \notin refs(o_p))\}$$

- The removed objects set contains all objects which were contained in a previous version of a model skeleton and now have been removed by the agent. We perform reference counting to ensure that no objects are deleted in the model repository which are still referenced in any of the model skeletons.

$$\Delta_{remove} = \{o_p \in O_p \mid b \in refs(o_p) \wedge (\forall o_s \in O_s : \neg matches(o_s, o_p))\}$$

Merging. The merging step uses the sets Δ_{new}, Δ_{exists}, and Δ_{remove} from the differencing step and merges the model skeleton m_s into the model repository m_p. The result is a new version m'_p of the model repository. The merging uses the following four primitives:

- $create(o_s)$: Creates a new object o_p in the model repository m'_p that matches the object o_s in the input model skeleton. The post-condition of this function is:

$$\exists o_p \in O'_p : (matches(o_s, o_p) \wedge b \in refs'(o_p) \\ \wedge (owns(o_s) \implies b \in owners'(o_p)))$$

  b is the agent which created the model skeleton. We update the $refs$ and $owner$ attribute functions of the model repository accordingly.

- $link(o_s, o_p)$: Similar to the $create$ function, except that a matching counterpart o_p of the object o_s in the model skeleton already exists in the model repository. The pre-condition of this function is:

$$(owns(o_s) \wedge (owners(o_p) \setminus \{b\} \neq \emptyset)) \implies shared(o_p)$$

  The pre-condition ensures that objects which cannot be shared between mutiple agents can only be owned by a single agent. If the pre-condition is not fulfilled, the merging aborts with a conflict state. The $link$ function updates the $refs'$ and $owner'$ in the model repository m'_p accordingly. This is enforced by the following post-condition:

$$b \in refs'(o_p) \wedge (owns(o_s) \Leftrightarrow b \in owners'(o_p))$$

- $remove(o_p)$: This function removes the agent b from the list of referencing agents and removes the object from the model repository if the number of references is zero. Its post-condition is:

$$b \notin owners'(o_p) \wedge b \notin refs'(o_p) \wedge (refs'(o_p) = \emptyset \Leftrightarrow o_p \notin O'_p)$$

- $merge(o_s, o_p)$: This function synchronizes the contents of the object o_s in the model skeleton and its counterpart o_p in the model repository. We define $\Omega_{o_p} = \{f \in F \mid domain(F) \in (class(o_p) \cup superTypes(class(o_p)))\}$ that contains all features of the class of o_p as well as all its super classes. Then the pre-condition of this function is:

  $$\forall f \in \Omega_{o_p} : (shared(o_p) \wedge f \notin id(class(o_p))) \Rightarrow (many(f) \wedge \neg ordered(f))$$

  Single-valued or ordered multi-valued features are not permitted for objects which are shared between agents, as we may overwrite changes of other agents. Identifier features used for matching are excluded given that they are ensured to always have the same value in o_s and o_p. The function has two post-conditions depending on whether it is a single-valued or multi-valued feature:

  $$\exists o'_p \in O'_p, \forall f \in \Omega_{o_p} : many(f) \Rightarrow fv_f(o'_p) = fv_f(o_p) \cup fv_f(o_s)$$

  $$\exists o'_p \in O'_p, \forall f \in \Omega_{o_p} : \neg many(f) \vee ordered(f) \Rightarrow fv_f(o'_p) = fv_f(o_s)$$

  In case of multi-valued, unordered features, we create the union of all its values in o_s and o_p. In case of single-value or an ordered multi-value features, we overwrite the value of the feature in the model repository with the one in the model skeleton.

The merging step calls the functions in the following order: a) $create$ for each object in set Δ_{new}, b) $link$ for each object in set Δ_{exists}, c) $remove$ for each object in set Δ_{remove}, and d) $merge$ for the set of objects $\Delta_{owned} = \{o_s \in O_s \mid owns(o_s)\}$.

Conflicts. If any of the pre-conditions of the merging primitives above were violated, the merging would fail in a conflict state. Given that we do not assume that a user may manually help to resolve the conflicts, we need to ensure that conflicts may not happen in a correctly set up system. The following invariants need to hold to ensure a conflict-free model repository:

- If a model object is allowed to be shared between agents, its non-identifying features may only be non-ordered and multi-valued.

  $$\forall o_p \in O_p, \forall f \in \Omega_{o_p} : shared(o_p) \wedge f \notin id(class(o_p)) \implies many(f) \wedge \neg ordered(f)$$

- Each non-shared model object may be only owned by a single agent:

$$\forall o_p \in O_p : \neg shared(o_p) \implies |owners(o_p)| \leq 1$$

The first invariant can be enforced through a deliberate formulation of the $shared$ and id functions. The second invariant needs to be checked at system run-time. Our idea to avoid conflicts at run-time is to utilize the agent roles introduced in Section 4.2. We require each agent to specify on start up, which roles it fulfills. The agent is only permitted access to the model repository if in the same extraction scope no other agent with any of these roles is registered or if a role explicitly allows multiple agents.

We now discuss, which agent roles may allow multiple agents of the same role in the same extraction scope. Given agent role t, the subset $C_t \subseteq C$ of meta-model classes specifies which objects an agent in t may own in its model skeleton. It is important to note, that the agent still may reference objects from the full set C. If for all classes in C_t holds that all non-identifying features are non-ordered and multi-valued, multiple agents of role t may be allowed in the same extraction scope.

We determined the functions $shared$ and id for the DML meta-model and identified the following agent roles that allow for sharing of model objects: *D1* (Data Center Structure), *D5* (System Interface Providing Roles), *D6* (Application Assembly), *A10* (Component Boundary), *A2* (Component Assembly), *A3* (Component Deployment), and *A5* (Coarse-Grained Behavior).

4.4 Concluding Remarks

In this chapter, we presented a reference architecture for online model learning in virtualized environments. The reference architecture is based on *agents* which are responsible of extracting model skeletons of certain aspects of a system. Agents may employ different *static* and *dynamic* analysis techniques to create model skeletons at run-time. We expect a *deep integration* of agents into existing technologies and platforms in order to exploit domain-specific knowledge for model learning. The model skeletons are *dynamically composed* into a comprehensive performance model of a system. Our reference architecture is based on a common meta-model in order to ease the composition of model skeletons from different agents.

In order to implement our reference architecture, virtualization platforms need to be extended with additional components supporting the online model learning. However, these components are supplementary and do not require changes in the existing parts of a virtualization platform, as we will demonstrate

later in Section 7.2.1 for the VMware vSphere platform. We provide a reference implementation of the core components (i.e., model repository and message bus) of our architecture employing state-of-the-art MDD technologies.

Leveraging our reference architecture, future work may provide VAs containing model extraction agents focused on specific aspects of model learning. A performance engineer, who has expertise in performance modeling, can specifically design the extraction logic to exploit a-priori knowledge about a technology and leverage proprietary interfaces. For a given technology, this needs to be done only once and the resulting VA can be reused in different deployments. We describe exemplary implementations for the WildFly Java EE application server in Section 7.2.3 and the Zimbra collaboration server in Section 7.2.2. Furthermore, we describe the LibReDE tool in Section 5.3, which can be used as a generic agent for characterizing resource demand variables.

Chapter 5

Online Statistical Estimation of Resource Demands

The reference architecture in Chapter 4 enables the deep integration of model learning capabilities into virtualization platforms. It is able to provide us an overall DML model representing the current architecture of a system at any given point in time. However, before using this model for reasoning purposes, concrete values for various model variables (e.g., branching probabilities and resource demands) in this model need to be characterized. We rely on empirical data to determine values for model variables. Whereas most model variables can be directly observed at a system, the accurate quantification of resource demands poses a major challenge. In this chapter, we survey, systematize, and evaluate different approaches to the statistical estimation of resource demands and propose a novel method relying on multiple statistical techniques for increased robustness and accuracy.

Our understanding of resource demands is based on the one used with stochastic performance models, such as QNs. A resource demand is the average time a unit of work (e.g., request or transaction) spends obtaining service from a resource (e.g., CPU or hard disk) in a system over all visits excluding any waiting times (Lazowska et al., 1984; Menascé et al., 2004). The resource demand for processing a request is influenced by different factors, for example, the application logic specifies the sequence of instructions to process a request, the hardware platform determines how fast individual instructions are executed, and platform layers (e.g., hypervisor, operating system, or middleware systems) may introduce additional processing overhead.

While the direct measurement of resource demands is feasible in some systems, it requires an extensive instrumentation of the application and typically introduces significant overheads that may distort measurements. For instance, standard profiling tools for performance debugging (Graham et al., 1982; Hall, 1992) can be used to obtain execution times of individual application functions when processing an individual request. However, the resulting execution times

are not broken down to the processing times at individual resources and profiling tools typically introduce high overheads significantly influencing the performance of a system. Furthermore, advanced instrumentation techniques have been proposed in the literature to measure resource demands on the operating system layer (Barham et al., 2004), or the application layer (Brunnert et al., 2013; Kuperberg et al., 2008a, 2009). These techniques build upon specific capabilities of the underlying platform and are not generally applicable.

Challenges In this chapter, we strive for a generic method to determine resource demands at system run-time without relying on a dedicated instrumentation of the application. Therefore, the goal is to estimate the resource demands based on indirect measurements of commonly available metrics (e.g., end-to-end response time, and resource utilization). However, such a generic approach needs to address the following challenges:

- The value of a resource demand is platform-specific (i.e., only valid for a specific combination of application, operating system, hardware platform, etc.). The hardware platform determines how fast a piece of code executes in general. Furthermore, each platform layer on top (e.g., hypervisor, operating system, and middleware systems) may add additional overheads influencing the resource demands of an application.

- Applications often serve a mix of different types of requests (e.g., read or write transactions), which also differ in their resource demands. For resource management purposes it is beneficial to be able to distinguish between different types of requests. Therefore, we want to quantify the resource demands separately for each type of request (also called workload classes).

- Modern operating systems can only provide aggregate resource usage statistics on a per-process level. Many applications, especially the ones running in data centers, serve different requests with one or more operating system processes (e.g., HTTP web servers). The operating system is unaware of the requests served by an application and therefore cannot attribute the resource usage to individual requests.

- Many applications only allow the collection of time-aggregated request statistics (e.g., throughput or response time), while they are serving production workloads. A tracing of individual requests is often considered too expensive for a production system, as it may influence the application performance negatively.

- Resource demands may change over time due to platform reconfigurations (e.g., operating system updates) or dynamic changes in the application state (e.g., increasing database size). Therefore, resource demands need to be continuously updated at system run-time based on current measurement data.

Research Questions In the last decades, many different approaches to resource demand estimation have been proposed in the literature. These approaches differ in their modeling assumptions and the underlying statistical techniques. In this chapter, we address the following research questions:

- Which approaches to resource demand estimation exist in the literature? What are their assumptions with regards to the system structure and the measurement data? We provide the first systematization and survey of the state-of-the-art on resource demand estimation.
- Which approach provides the highest estimation accuracy? Which factors influence the estimation accuracy of the approaches? We evaluate seven different approaches to resource demand estimation and compare their accuracy in a series of experiments with varying settings. Existing evaluations of estimation approaches are limited to a single or a small subset of approaches.
- How to automatically decide which set of estimation approaches to apply in a given scenario? How to derive the structural information required for resource demand estimation from DML models? We describe a method to resource demand estimation that automatically selects an estimation approach based on a cross-validation of the results.

Chapter Outline We first survey existing approaches to resource demand estimation and provide a systematization of them in Section 5.1. Section 5.2 presents the results of comparing different estimation approaches in a series of experiments and discusses the results. We describe our method to resource demand estimation employing multiple estimation techniques in Section 5.3. Section 5.4 concludes this chapter.

5.1 Systematization of Approaches

In this section, we survey the state-of-the-art in resource demand estimation and and provide a systematization of existing estimation approaches. The goal

of the systemization is to help performance engineers to select an estimation approach that best fits their specific requirements. We first survey existing estimation approaches and describe their modeling assumptions and their underlying statistical techniques. Then, we introduce three dimensions for systemization: input parameters, output metrics and robustness to anomalies in the input data. For each dimension, we first describe its features and then categorize the estimation approaches accordingly. This section is based on our article in Spinner et al. (2015a).

Methodology In order to obtain the estimation approaches listed in Table 5.2, we started the literature search by reading the titles and abstract of articles in the proceedings of 12 established conferences and workshops in the performance engineering community in the last 10 years. Relevant articles were analyzed further regarding references to other articles on resource demand estimation. Based on the found articles found we compiled a list of keywords to use for a broader search in common scientific search engines (scholar.google.com, portal.acm.org and citeseerx.ist.psu.edu). The keywords used for search were *resource demand (estimation)*, including synonyms *service demand*, *service time*, *service requirement*. Furthermore, we also considered the more general terms *workload characterization*, *parameter estimation* and *model calibration*. The list of articles resulting from this search was then filtered based on the titles and abstracts. After filtering, we got the list of 37 papers on resource demand estimation shown in Table 5.2.

Notation and Assumptions In the following, we use a consistent notation for the description of the different approaches to resource demand estimation. We denote resources with the index $i = 1 \dots I$ and workload classes with the index $c = 1 \dots C$. The variables used in the description are listed in Table 5.1. We assume the *Flow Equilibrium Assumption* (Menascé and Gomaa, 2000) to hold, i.e., that over a sufficiently long period of time the number of completions is approximately equal to the number of arrivals. As a result, the arrival rate λ_c is assumed to be equal to the throughput X_c. Furthermore, we use the term resource demand as a synonym for service demand and for simplicity of exposition we assume $V_{i,c} = 1$, i.e., no distinction is made between service demand and service time.

5.1.1 Estimation Approaches

In this section, we describe the different approaches to resource demand estimation that exist in the literature. Table 5.2 gives an overview of all approaches.

$D_{i,c}$	average resource demand of requests of workload class c at resource i
$U_{i,c}$	average utilization of resource i due to requests of workload class c
U_i	average total utilization of resource i
$\lambda_{i,c}$	average arrival rate of workload class c at resource i
$X_{i,c}$	average throughput of workload class c at resource i
$R_{i,c}$	average residence time of workload class c at resource i
R_c	average end-to-end response time of workload class c
$A_{i,c}$	average queue length of requests of workload class c seen on arrival at resource i, excluding the arriving job
$V_{i,c}$	average number of visits of a request of workload class c at resource i
I	total number of resources
C	total number of workload classes

Table 5.1: Explanation of variables.

5.1.1.1 Approximation with Response Times

Assuming a single queue and insignificantly small queueing delays compared to the resource demands, we can approximate the resource demand with the observed response times. However, this trivial approximation only works with systems under light load where a single resource dominates the observed response time. This approximation is used in (Brosig et al., 2009; Nou et al., 2009; Urgaonkar et al., 2007).

5.1.1.2 Service Demand Law

The Service Demand Law (see Equation 2.19 on page 35) is an operational law that can be used to directly calculate the the demand $D_{i,c}$, given the utilization $U_{i,c}$ and the throughput $X_{i,c}$ (Menascé et al., 2004). However, modern operating systems can only report the utilization on a per-process level. Therefore, we usually cannot observe the per-class utilization $U_{i,c}$ directly given that single processes may serve requests of different workload classes. Given a system serving requests of multiple workload classes, Lazowska et al. (1984) and Menascé et al. (2004) recommend to use additional per-class metrics if available (e.g., in the operating system) to apportion the aggregate utilization U_i of a resource between workload classes. Brosig et al. (2009) use an approximate apportioning scheme based on the assumption that the observed response times are proportional to the resource demands.

Technique	Variant	References
Approximation with response times		Urgaonkar et al., 2007 Nou et al., 2009 Brosig et al., 2009
Service Demand Law		Lazowska et al., 1984 Brosig et al., 2009
Linear regression	Least squares	Bard and Shatzoff, 1978 Rolia and Vetland, 1995, 1998 Pacifici et al., 2008 Kraft et al., 2009; Pérez et al., 2013
	Least absolute differences	Kelly and Zhang, 2006; Stewart et al., 2007; Zhang et al., 2007
	Least trimmed squares	Casale et al., 2008; Casale et al., 2007
Kalman filter		Zheng et al., 2008, 2005 Kumar et al., 2009a Wang et al., 2012, 2011
Optimization	Non-linear constrained optimization	Zhang et al., 2002 Menascé, 2008
	Quadratic programming	Liu et al., 2006, 2003; Wynter et al., 2004 Kumar et al., 2009b
Machine learning	Clusterwise linear regression	Cremonesi et al., 2010
	Independent component analysis	Sharma et al., 2008
	Support vector machine	Kalbasi et al., 2011
	Pattern matching	Cremonesi and Sansottera, 2012, 2014
Maximum likelihood estimation		Kraft et al., 2009 Pérez et al., 2013
Gibbs sampling		Sutton and Jordan, 2011 Wang and Casale, 2013
Demand Estimation with Confidence (DEC)		Kalbasi et al., 2012; Rolia et al., 2010

Table 5.2: Overview of estimation approaches categorized according to statistical techniques.

5.1.1.3 Linear Regression

Given a linear model $\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$ where $\boldsymbol{\beta}$ (see also Section 2.3.1) is a vector of resource demands $D_{i,r}$ and $\mathbf{Y}$, $\mathbf{X}$ contain observations of performance metrics of a system, we can use linear regression techniques to estimate the resource demands. Two alternative formulations of a such a linear model for resource demand estimation have been proposed in the literature:

- The Utilization Law (see Equation 2.18 on page 35) requires observations of the aggregate utilization U_i and the throughputs $\lambda_{i,c}$. This is a classical model used by different authors (Bard and Shatzoff, 1978; Casale et al., 2007; Kelly and Zhang, 2006; Kraft et al., 2009; Pacifici et al., 2008; Rolia and Vetland, 1995; Stewart et al., 2007; Zhang et al., 2007). Some of the authors include a constant term $U_{i,0}$ in the model to estimate the utilization caused by background work that cannot be directly attributed to the requests.

- Kraft et al. (2009) and Pérez et al. (2013) propose a linear model based on a multi-class version of the response time equation $R_i = D_i(1 + A_i)$ requiring observations of the queue length A_i seen by a newly arriving job, and its response time R_i. In their initial work (Kraft et al., 2009), they assume a FCFS scheduling strategy and Pérez et al. (2013) generalizes the model to PS queueing stations.

Bard and Shatzoff (1978), Kraft et al. (2009), Pacifici et al. (2008), and Rolia and Vetland (1995, 1998) use non-negative least squares regression for solving the linear model. Other regression techniques, such as least absolute differences regression (Kelly and Zhang, 2006; Stewart et al., 2007; Zhang et al., 2007) or least trimmed squares (Casale et al., 2008; Casale et al., 2007) were proposed to increase the robustness of regression-based estimation techniques to multi-collinearities, outliers, or abrupt changes in the demand values.

5.1.1.4 Kalman Filter

Resource demands of a system may vary over time, e.g., due to changing system states, or changing user behavior. These variations may be abrupt or continuous. In order to track time-varying resource demands, Kumar et al. (2009a), Wang et al. (2012), and Zheng et al. (2008) use a Kalman filter (see Section 2.3.2). The authors assume a dynamic system where the state vector $\mathbf{x}$ consists of the hidden resource demands $D_{i,r}$ that need to be estimated. Given no prior knowledge about the dynamic behavior of the system state exists,

they assume a constant state model, i.e., Equation 2.5 on page 27 is reduced to $\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{w}_k$.

The observation model $\mathbf{z} = h(\mathbf{x})$ requires a functional description of the relationship between the observations $\mathbf{z}$ and the system state $\mathbf{x}$. Wang et al. (2012) use the observed utilization U_i as vector $\mathbf{z}$ and $h(\mathbf{x})$ is based on the Utilization Law (see Equation 2.18 on page 35). Given the linear model, a conventional Kalman filter is sufficient. Kumar et al. (2009a) and Zheng et al. (2008) use an observation vector consisting of the observed response time $R_{i,c}$ of each workload class and the utilization U_i of each resource. The function $h(\mathbf{x})$ is based on the solution of a M/M/1 queue (see Equation 2.23 on page 36), as well as the Utilization Law. Due to the non-linear nature, it requires an extended Kalman filter design (see Equation 2.7 on page 27).

5.1.1.5 Optimization

Given a general QN, we can formulate an optimization problem to search for values of the resource demands so that the differences between performance metrics observed at a real system and the ones calculated using the QN are minimized. The main challenge is the solution of the QN. Depending on the structure of the QN, its solution may be computationally expensive and the optimization algorithm may need to evaluate the QN with many different resource demand values to find an optimal solution. Existing approaches (Kumar et al., 2009b; Liu et al., 2006, 2003; Menascé, 2008; Wynter et al., 2004) assume a product-form QN with an open workload. Then, the equations in Section 2.4.1.4 can be used to calculate the end-to-end response times.

Suppose N observations of the end-to-end response time $\tilde{R}_c$ and the utilization $\tilde{U}_i$, Liu et al. (2006) propose the following objective function:

$$\min_{\mathbf{D}} \sum_{n=1}^{N} \left(\sum_{c=1}^{C} p_c (R_c(\mathbf{D}) - \tilde{R}_c^{(n)})^2 + \sum_{i=1}^{I} (U_i(\mathbf{D}) - \tilde{U}_i^{(n)})^2 \right). \tag{5.1}$$

The function $R_c(\mathbf{D})$ is based on the solution of a M/M/1 queue (see Equation 2.23 on page 36) and $U_i(\mathbf{D})$ on the Utilization Law. The factor p_c introduces a weighting according to the arrival rates of workload classes: $p_c = \frac{\lambda_c}{\sum_{d=1}^{C} \lambda_d}$. The resulting optimization problem can be solved using quadratic programming techniques. Kumar et al. (2009b) extend this optimization approach to estimate load-dependent resource demands. Their approach requires prior knowledge of the type of function, e.g., polynomial, exponential or logarithmic, that best describes the relation between arriving workloads and resource demands.

Menascé (2008) formulates an alternative optimization problem that depends only on response time and arrival rate measurements:

$$\min_{\mathbf{D}} \sum_{c=1}^{C} (R_c(\mathbf{D}) - \tilde{R}_c)^2 \text{ with } R_c(\mathbf{D}) = \sum_{i=1}^{I} \frac{D_{i,c}}{1 - \sum_{d=1}^{C} \lambda_{i,d} D_{i,d}} \tag{5.2}$$

$$\text{subject to } D_{i,c} \geq 0 \quad \forall i, c \text{ and } \sum_{c=1}^{C} \lambda_{i,c} D_{i,c} < 1 \quad \forall i.$$

In contrast to Liu et al. (2006), this formulation is based on a single sample of the observed response times. Menascé (2008) proposes to repeat the optimization for each new sample using the previous resource demand estimate as the initial point. To solve this optimization problem we depend on a non-linear constrained optimization algorithm.

5.1.1.6 Machine Learning

Cremonesi et al. (2010) use cluster-wise regression techniques to improve the robustness to discontinuities in the resource demands due to system configuration changes. The observations are clustered into groups where the resource demands can be assumed constant, and the demands are then estimated for each cluster separately. Cremonesi and Sansottera (2012, 2014) propose a novel algorithm based on a combination of change-point regression methods and pattern matching to address the same challenge.

Independent Component Analysis (ICA) is a method to solve the *blind source separation* problem, i.e., to estimate the individual signals from a number of aggregate measurements. Sharma et al. (2008) describes a way to use ICA for resource demand estimation, using a linear model based on the Utilization Law. ICA can provide estimates solely based on utilization measurements, when the following constraints hold (Sharma et al., 2008): i) the number of workload classes is limited by the number of observed resources; ii) the arrival rate measurements are statistically independent; iii) the inter-arrival times have a non-Gaussian distribution while the measurement noise is assumed zero-mean Gaussian. ICA not only provides estimates of resource demands, but also automatically categorizes requests into workload classes.

Kalbasi et al. (2011) consider the use of Support Vector Machines (SVM) (Smola and Schölkopf, 2004) for estimating resource demands. They compare it with results from LSQ and LAD regression and show that it can provide better resource demand estimates depending on the characteristics of the workload.

5.1.1.7 Maximum Likelihood Estimation (MLE)

Kraft et al. (2009) and Pérez et al. (2013) use Maximum Likelihood Estimation (MLE) (see Section 2.3.4) to estimate resource demands based on observed response times and queue lengths seen upon arrival of requests. Suppose N response time measurements $R_i^1, \ldots, R_i^N$ of *individual* requests, the estimated resource demands $D_{i,1}, \ldots, D_{i,C}$ are the values that maximize the likelihood function $\mathbb{L}(D_{i,1}, \ldots, D_{i,C})$. They use the following likelihood function:

$$\max \mathbb{L}(D_{i,1}, \ldots, D_{i,C}) = \sum_{k=1}^{N} \log f(R_i^k \mid D_{i,1}, \ldots, D_{i,C}). \tag{5.3}$$

The density function f is obtained by constructing a phase-type distribution. The phase-type distribution describes the time to absorption in a Markov chain representing the current state of the system. Observations of the queues length are necessary to be able to construct the corresponding phase-type distribution. Kraft et al. (2009) describe the likelihood function for queueing stations with FCFS scheduling. Pérez et al. (2013) generalize this approach to PS scheduling.

5.1.1.8 Gibbs sampling

Sutton and Jordan (2011) and Wang and Casale (2013) both propose approaches to resource demand estimation based on Bayesian inference techniques (see Section 2.3.5). Sutton and Jordan (2011) assume an open, single-class QN. They develop a deterministic mathematical model that allows the calculation of service times and waiting times of individual requests given the arrival times, departure times and the path of queues of all requests in a QN. They assume that this information can only be observed for a subset of requests. Therefore, they propose a Gibbs sampler to sample the missing departure times of requests that were not observed. Given the posterior distribution of the departures times of all requests, they then derive the expected resource demands at the individual queues.

Wang and Casale (2013) assume a multi-class, closed QN that fulfills the BCMP theorem. Under this assumptions, the probability distribution of the queue lengths for given resource demands is well-known (see Equation 2.16 on page 34). They assume the availability of queue-length samples from a real system and construct a Gibbs sampler for the posterior distribution $f(\mathbf{D}|\mathbf{A})$, where $\mathbf{D}$ is a vector of resource demands $D_{i,c}$ and $\mathbf{A}$ is a vector of observed queue lengths $A_{i,c}$. They propose an approximation for the conditionals of the posterior distribution as required by the Gibbs sampling algorithm. A main

challenge is the calculation of the normalization constant G for the steady-state probabilities (see Equation 2.16 on page 34), which is non-trivial for closed QNs. Wang and Casale (2013) propose a Taylor expansion of G and apply a MVA algorithm to determine its value.

5.1.1.9 Other Approaches

Rolia et al. (2010) and Kalbasi et al. (2012) propose a technique for estimating the aggregate resource demand of a given workload mix, called Demand Estimation with Confidence (DEC). This technique assumes that a set of benchmarks is available for a system under study. Each benchmark utilizes a subset of the different functions of an application. DEC expects the measured demands of the individual benchmarks as input and then derives the aggregate resource demand of a given workload mix as a linear combination of the demands of the individual benchmarks. DEC is able to provide confidence intervals of the aggregate resource demand (Kalbasi et al., 2012; Rolia et al., 2010).

5.1.2 Input Parameters

Approaches to resource demand estimation often differ in terms of the set of input data they require. We do no consider parameters of the underlying statistical techniques (e.g., parameters controlling the optimization algorithm) because these are specific to the concrete implementation of an estimation approach.

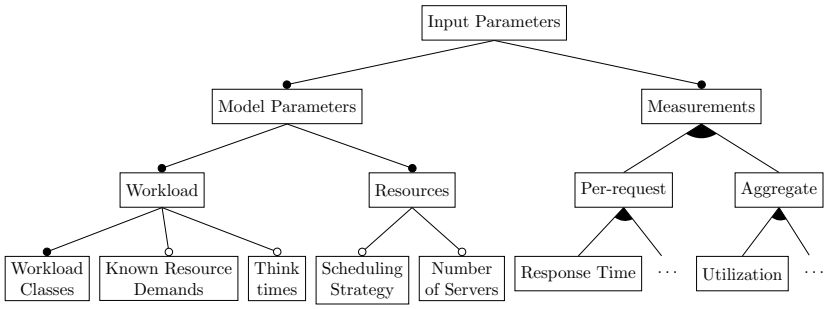


Figure 5.1: Types of input parameters.

Figure 5.1 depicts the main types of input parameters for demand estimation algorithms. The parameters are categorized into *model parameters* and *measurements*. In general, parameters of both types are required. Model parameters capture information about the performance model for which we estimate resource demands. Measurements consist of samples of relevant performance

metrics obtained from a running system, either a live production system or a test system.

Before estimating resource demands, it is necessary to decide on certain modeling assumptions. As a first step, resources and workload classes need to be identified. This is typically done as part of the workload characterization activity when modeling a system. It is important to note, that the observability of performance metrics may influence the selection of resources and workload classes for a system under study. In order to be able to distinguish between individual resources or workload classes, observations of certain per-resource or per-class performance metrics are necessary. At a minimum, information about the number of workload classes and the resources for which the demands should be determined is required as input to the estimation. Depending on the estimation approach, more detailed information on resources and workload classes may be expected as an input (e.g., *scheduling strategies*, *number of servers*, or *think times*).

Measurements can be further grouped into *per-request* or *aggregate*. Common per-request measurements used in the literature include response times, arrival rates, visit counts, and queue length seen upon arrival. Aggregate measurements can be further distinguished in *class-aggregate* and *time-aggregate* measurements. Class-aggregate measurements are collected as totals over all workload classes processed at a resource. For instance, utilization is usually reported as an aggregate value because the operating system is agnostic of the application internal logic and is not aware of different request types in the application. Time-aggregate measurements, e.g., average response times or average throughput, are aggregated over a sampling period. The sampling period can be evenly or unevenly spaced.

Categorization of Existing Approaches

We considered the approaches to resource demand estimation listed in Table 5.2 and examined their input parameters. Table 5.3 contains an overview of the input parameters of each estimation approach. Parameters common to all estimation approaches, such as the number of workload classes and the number of resources, are not included in this table. The required input parameters vary widely between different estimation approaches. Depending on the system under study and the available performance metrics, one can choose a suitable estimation approach from Table 5.3. Furthermore, approaches based on optimization can be adapted by incorporating additional constraints into the mathematical model capturing the knowledge about the system under study. For example, the optimization approach by Menascé (2008) allows one to

Table 5.3: Input parameters of estimation approaches (utilization U_i, response time R_c, throughput X_c, arrival rate λ_c, queue length $A_{i,c}$, visit counts $V_{i,c}$, demands $D_{i,c}$, think time Z, scheduling policy P).

Estimation approach	Measurements					Parameters		
	U_i	R_c	X_c/λ_c	$A_{i,c}$	$V_{i,c}$	$D_{i,c}$	Z	P
Approximation with response times								
Urgaonkar et al., 2007		✗[1]			✗			
Nou et al., 2009	✗	✗						
Brosig et al., 2009		✗						
Service Demand Law								
Lazowska et al., 1984			✗			✗[2]		
Brosig et al., 2009	✗	✗			✗			
Linear regression								
Bard and Shatzoff, 1978, Rolia and Vetland, 1995, 1998, Pacifici et al., 2008	✗		✗					
Kelly and Zhang, 2006; Zhang et al., 2007, Stewart et al., 2007	✗		✗					
Kraft et al., 2009; Pérez et al., 2015, 2013		✗		✗				✗
Casale et al., 2008; Casale et al., 2007	✗		✗					
Kalman filter								
Zheng et al., 2008, 2005	✗	✗	✗					
Kumar et al., 2009a	✗	✗	✗					
Wang et al., 2012, 2011	✗		✗					
Optimization								
Zhang et al., 2002	✗	✗	✗			(✗)[5]		✗
Liu et al., 2006, 2003; Wynter et al., 2004	✗	✗	✗		✗			✗
Menascé, 2008		✗	✗			✗[3]		
Kumar et al., 2009b	✗	✗	✗					✗
Machine learning								
Cremonesi et al., 2010	✗		✗					
Sharma et al., 2008	✗							
Kalbasi et al., 2011	✗		✗					
Cremonesi and Sansottera, 2012, 2014	✗		✗					
Maximum likelihood estimation								
Kraft et al., 2009		✗[4]		✗[4]			✗	✗
Pérez et al., 2015, 2013		✗[4]		✗[4]			✗	✗
Gibbs sampling								
Sutton and Jordan, 2011		✗[4]	✗[4]					✗
Wang and Casale, 2013				✗[4]			✗	
Kalbasi et al., 2012; Rolia et al., 2010			✗		✗			

[1] Response time per resource.
[2] Measured with accounting monitor. System overhead is not included.
[3] A selected set of resource demands is known a priori.
[4] Non-aggregated measurements of individual requests.
[5] Requires coefficient of variation of resource demands in case of FCFS scheduling.

specify additional known resource demand values as input parameters. These a-priori resource demands may be obtained from the results of other estimation approaches or from direct measurements.

Another approach that requires resource demand data is described by Lazowska et al. (1984, Chapter 12). Lazowska assumes that the resource demands are approximated based on measurements provided by an accounting monitor. Such an accounting monitor, however, does not include the system overhead caused by each workload class. The system overhead is defined as the work done by the operating system for processing a request. Lazowska et al. (1984, Chapter 12) describes a way to distribute unattributed computing time among the different workload classes providing more realistic estimates of the actual resource demands.

Approaches based on response time measurements, such as those proposed by Zhang et al. (2002), Liu et al. (2006, 2003), Wynter et al. (2004) and Kumar et al. (2009b), require information about the scheduling strategies of the involved resources abstracted as queueing stations. This information is used to construct the correct problem definition for the optimization technique. The estimation approaches proposed by Kraft et al. (2009), Pérez et al. (2013), and Wang and Casale (2013) assume a closed queueing network. Therefore, they also require the average think time and the number of users as input.

In addition to requiring a set of specific input parameters, some approaches also provide a rule of thumb regarding the number of required measurement samples. Approaches based on linear regression (Kraft et al., 2009; Pacifici et al., 2008; Rolia and Vetland, 1995) need at least $K + 1$ linear independent equations to estimate K resource demands. When using robust regression methods, significantly more measurements might be necessary (Casale et al., 2007). Kumar et al. (2009b) provide a formula to calculate the number of measurements required by their optimization-based approach. The formula only provides a minimum bound on the number of measurements and more measurements are normally required to obtain good estimates (Stewart et al., 2007).

5.1.3 Output Metrics

Approaches to resource demand estimation are typically used to determine the mean resource demand of requests of a given workload class at a given resource. However, in many situations the estimated mean value may not be sufficient. Often, more information about the confidence of estimates and the distribution of the resource demands is required. The set of output metrics an

estimation approach provides can influence the decision to adopt a specific method.

Generally, resource demands cannot be assumed to be deterministic (Rolia et al., 2010); for example, they might depend on the data processed by an application or on the current state of the system (Rolia and Vetland, 1995). Therefore, resource demands are described as random variables. Estimates of the mean resource demand should be provided by every estimation approach. If the distribution of the resource demands is not known beforehand, estimates of higher moments of the resource demands may be useful to determine the shape of their distribution.

We distinguish between point and interval estimators of the real resource demands. Confidence intervals would be generally preferable, however, it is often a challenge to ensure that the statistical assumptions underlying a confidence interval calculation hold for a system under study (e.g., distribution of the regression errors).

In certain scenarios, e.g., if Dynamic Voltage and Frequency Scaling (DVFS) or hyperthreading techniques are used (Kumar et al., 2009b), the resource demands are load-dependent. In such cases, the resource demands are not constant, but a function that may depend, e.g., on the arrival rates of the workload classes (Kumar et al., 2009b).

Categorization of Existing Approaches

Table 5.4 provides an overview of the output metrics of the considered estimation approaches. Point estimates of the mean resource demand are provided by all approaches. Confidence intervals can be determined for linear regression using standard statistical techniques, as mentioned by Kraft et al. (2009) and Rolia and Vetland (1995). These techniques are based on the central limit theorem assuming an error term with a normal distribution. Resource demands are typically not deterministic violating the assumptions underlying linear regression. The influence of the distribution of the resource demands on the accuracy of the confidence intervals is not evaluated for any of the approaches based based on linear regression. DEC (Kalbasi et al., 2012; Rolia et al., 2010) is the only approach for which the confidence intervals have been evaluated in the literature (Kalbasi et al., 2012; Rolia et al., 2010). The MLE approach (Kraft et al., 2009) and the optimization approach described by Zhang et al. (2002) are capable of providing estimates of higher moments. This additional information comes at the cost of a higher amount of required measurements.

All of the estimation approaches in Table 5.2 can estimate load-independent mean resource demands. Additionally, the Enhanced Inferencing approach (Ku-

Table 5.4: Output metrics of estimation approaches.

Estimation approach	Resource demands			
	Point estimates	Confidence interval	Higher moments	Load-dependent
Response time approximation				
Urgaonkar et al., 2007	✗			
Nou et al., 2009	✗			
Brosig et al., 2009	✗			
Service Demand Law				
Lazowska et al., 1984	✗			
Brosig et al., 2009	✗			
Linear regression				
Bard and Shatzoff, 1978 Rolia and Vetland, 1995, 1998, Pacifici et al., 2008	✗	✗[2]		
Zhang et al., 2007	✗	✗[2]		
Kraft et al., 2009; Pérez et al., 2015, 2013	✗	✗[2]		
Casale et al., 2008; Casale et al., 2007	✗	✗[2]		
Kalman filter				
Zheng et al., 2008, 2005	✗			
Kumar et al., 2009a	✗			
Wang et al., 2012, 2011	✗			
Optimization				
Zhang et al., 2002	✗		✗[1]	
Liu et al., 2006, 2003; Wynter et al., 2004	✗			
Menascé, 2008	✗			
Kumar et al., 2009b	✗			✗
Machine learning				
Cremonesi et al., 2010	✗			
Sharma et al., 2008	✗			
Kalbasi et al., 2011	✗			
Cremonesi and Sansottera, 2012, 2014	✗			
Maximum likelihood estimation				
Kraft et al., 2009	✗		✗	
Pérez et al., 2015, 2013	✗		✗	
Gibbs sampling				
Sutton and Jordan, 2011	✗			
Wang and Casale, 2013	✗			
Kalbasi et al., 2012; Rolia et al., 2010 (DEC)	✗	✗		

[1] Only feasible if a-priori knowledge of the resource demand variance is available.
[2] The accuracy of the confidence intervals is not evaluated.

mar et al., 2009b) also supports the estimation of load-dependent resource demands, assuming a given type of function.

5.1.4 Robustness

It is usually not possible to control every aspect of a system while collecting measurements. This can lead to anomalous behavior in the measurements. Casale et al. (2008), Casale et al. (2007), and Pacifici et al. (2008) identified the following issues with real measurement data:

- presence of outliers,
- background noise,
- non-stationary resource demands,
- collinear workload,
- and insignificant flows.

Background activities can have two effects on measurements: the presence of outliers and background noise (Casale et al., 2007). Background noise is created by secondary activities that utilize a resource only lightly over a long period of time. Outliers result from secondary activities that stress a resource at high utilization levels for a short period of time. Outliers can have a significant impact on the parameter estimation resulting in biased estimates (Casale et al., 2007). Different strategies are possible to cope with outliers. It is possible to use special filtering techniques in an upstream processing step or to use parameter estimation techniques that are inherently robust to outliers. However, tails in measurement data from real systems might belong to bursts, e.g., resulting from rare, but computationally complex requests. The trade-off decision as to when an observation is to be considered as an outlier has to made on a case-by-case basis taking into account the characteristics of the specific scenario and application.

The resource demands of a system may be non-stationary over time (i.e., not only the arrival process changes over time, but also the resource demands, which for example can be described by a $M_t/M_t/1$ queue). Different types of changes are observed in production systems. Discontinuous changes in the resource demands can be caused by software and hardware reconfigurations, e.g., the installation of an operating system update (Casale et al., 2007). Continuous changes in the resource demands may happen over different time scales. Short-term variations can often be observed in cloud computing environments

where different workloads experience mutual influences due to the underlying shared infrastructure. Changes in the application state (e.g., database size) or the user behavior (e.g., increased number of items in a shopping cart in an online shop during Christmas season) may result in long-term (over days, weeks, and months) trends and seasonal patterns. When using the estimated resource demands to forecast the required resources of an application over a longer time period, these non-stationary effects need to be considered in order to obtain accurate predictions. In order to detect such trends and seasonal patterns, it is possible to apply forecasting techniques on a time series resulting from the repeated execution of one the considered estimation approach over a certain time period. An overview of such forecasting approaches based on time series analysis can be found in Box et al. (2008a).

Another challenge for estimation approaches is the existence of collinearities in the arrival rates of different workload classes. There are two possible reasons for collinearities in the workload: low variation in the throughput of a workload class or dependencies between workload classes (Pacifici et al., 2008). For example, if we model *login* and *logout* requests each with a separate workload class, the resulting classes would normally be correlated (Pacifici et al., 2008). The number of logins usually approximately matches the number of logouts (Pacifici et al., 2008). Collinearities in the workload may have negative effects on resource demand estimates. A way to avoid these problems is to detect and combine workload classes that are correlated (Pacifici et al., 2008).

Insignificant flows are caused by workload classes with very small arrival rates in relation to the arrival rates of the other classes. Pacifici et al. (2008) experience numerical stability problems with their linear regression approach when insignificant flows exist. However, it is noteworthy, that there might be a dependency between insignificant flows and the length of the sampling time intervals. If the sampling time interval is too short, the variance in arrival rates might be high.

Categorization of Existing Approaches

Ordinary least-squares regression is often sensitive to outliers. Stewart et al. (2007) come to the conclusion that least-absolute-differences regression is more robust to outliers. Robust regression techniques as described by Casale et al. (2008) and Casale et al. (2007) try to detect outliers and ignore measurement samples that cannot be explained by the regression model. Liu et al. (2006) also include an outlier detection mechanism in their estimation approach based on optimization.

In general, sliding window or data aging techniques can be applied to the input data to improve the robustness to non-stationary resource demands (Pacifici et al., 2008). In order to detect software and hardware configuration discontinuities, robust and cluster-wise regression approaches are proposed in Casale et al. (2008), Casale et al. (2007), and Cremonesi et al. (2010). If such discontinuities are detected, the resource demands are estimated separately before and after the configuration change. Approaches based on Kalman filters (Kumar et al., 2009a; Zheng et al., 2008, 2005) are designed to estimate time-varying parameters. Therefore, they automatically adapt to changes in the resource demands after a software or hardware discontinuity. None of the considered estimation approaches are able to learn long-term trends or seasonal patterns (over days, weeks, or months).

Collinearities are one of the major issues when using linear regression (Chatterjee and Price, 1995). A common method to cope with this issue is to check the workload classes for collinear dependencies before applying linear regression. If collinearities are detected, the involved workload classes are merged into one class. This is proposed in Casale et al. (2007) and Pacifici et al. (2008). The DEC approach (Rolia et al., 2010) mitigates collinear dependencies, since it only estimates the resource demands for mixes of workload classes.

Pacifici et al. (2008) also consider insignificant flows. They call a workload class insignificant if the ratio between the throughput of the workload class and the throughput of all workload classes is below a given threshold. They completely exclude insignificant workload classes from the regression in order to avoid numerical instabilities (Pacifici et al., 2008).

5.2 Experimental Comparison

The goal of the experiments presented in this section is to compare the accuracy of different estimation approaches. A set of experiments was conducted to evaluate the impact of the following factors on the estimation accuracy of the considered estimation approaches: *(RQ1)* length of sampling interval, *(RQ2)* number of samples, *(RQ3)* number of workload classes, *(RQ4)* load level, *(RQ5)* collinear workload classes, *(RQ6)* missing jobs in workload model, and *(RQ7)* delays during processing. *(RQ8)* analyses the execution time of the considered estimation approaches. We describe the conducted experiments in detail and discuss the results. Section 5.2.1 describes the experiment setup used to obtain the measurement traces. The results of our experimental comparison have been published in our article in Spinner et al. (2015a).

5.2.1 Experiment Setup

In the experimental evaluation, we used two different sources to obtain the measurement traces for the comparison: a queueing simulator and a set of micro-benchmarks executed on a real system. The simulator and the micro-benchmarks each produce traces of observations of the performance metrics required for resource demand estimation. These traces are provided as input to the estimation approaches and the resulting resource demands are used to evaluate the estimation accuracy.

5.2.1.1 Dataset D1: Queueing Simulator

Dataset D1 consists of traces of arrival times and response times of individual requests from experiments with different number of workload classes $C = \{1, 2, 5\}$ and different load levels $U = \{10\%, 50\%, 90\%\}$. Each experiment was repeated 100 times resulting in a total of 900 different traces. We used a queueing simulator based on a M/M/1 queue with FCFS scheduling and an open workload that logs detailed statistics of each simulated request. Each experiment run simulated 3600 requests with exponential inter-arrival times. This corresponds to one hour of simulated time. Inter-arrival times and resource demands are both generated from exponential distributions. For each experiment run, the mean resource demand of each workload class is randomly drawn from a uniform distribution between 0 and 1 seconds, and scaled to yield the expected load level.

5.2.1.2 Dataset D2: Micro-Benchmarks

In order to obtain dataset D2, we performed a series of experiments running micro-benchmarks with a known CPU resource demand on a real system. The micro-benchmarks generate a closed workload with exponentially distributed think times and resource demands. As mean values for the resource demands, we selected 14 different subsets of the base set $[0.02s, 0.25s, 0.5s, 0.125s, 0.13s]$ with number of workload classes $C = \{1, 2, 3\}$. The subsets were arbitrarily chosen from the base set so that the resource demands are not linearly growing across workload classes. The subsets intentionally also contained cases where two or three workload classes had the same mean value as resource demand. The mean think times were determined according to the desired load level of an experiment. We again varied the number of workload classes $C = \{1, 2, 3\}$ the load level $U = \{20\%, 50\%, 80\%\}$ between experiments.

Each experiment run has a length of approximately one hour. Dataset D2 contains measurement traces from a total of 210 experiment runs. The mean

think time was calculated according to the required load level. We also used the micro-benchmarks to generate specialized traces for the scenarios evaluating a high number of workload classes (up to 20 classes) in Section 5.2.2.3, collinear workload classes in Section 5.2.2.5, background jobs in Section 5.2.2.6, and delayed processing in Section 5.2.2.7.

The micro-benchmarks were implemented with the Ginpex experiment framework (Hauck et al., 2014). The CPU load of the micro-benchmarks consists of the calculation of Fibonacci numbers, the number of iterations is calibrated by Ginpex before an experiment run to match the desired resource demand. We used a pool of machines with similar hardware configurations for the experiments. Each machine had an Intel Core 2 Quad Q6600 4 x 2.4 GHz CPU, 8 GB RAM, and 2 x 500 GB SATA2 disks, running a Ubuntu 10.04 64-bit operating system. We deactivated CPU cores in the operating system to prevent the parallel execution of the resource demands and to simulate a single-core machine.

During each experiment run we collected observations of the arrival times and execution times of individual requests, and the average CPU utilization. The execution times were measured by Ginpex (using the `System.nanoTime()` method provided by Java). The utilization was measured with the `sar` tool from the sysstat package (Godard, 2014), which is part of most Linux distributions. Average statistics for the throughput and response times were derived from the measurements afterwards.

5.2.2 Comparison of Estimation Approaches

Table 5.5 lists the approaches considered in the experimental evaluation. For reasons of conciseness, we use the abbreviations listed in the table to refer to estimation approaches in the following description. All estimation approaches were considered in the experimental evaluation with exception of response time approximation, Independent Component Analysis (ICA) (Sharma et al., 2008) and Maximum Likelihood Estimation (MLE) (Kraft et al., 2009; Wang and Casale, 2013). Response time approximation is a rather trivial approach where the assumptions are well-known, i.e., the observed response time must be close to the considered resource demand. In most practical scenarios this assumption does not hold, resulting in high estimation errors. ICA automatically groups the requests into workload classes besides estimating resource demands. However, the interpretation of the resulting classes is difficult and the resulting resource demands cannot be directly compared to other approaches. MLE has high computational requirements (both with respect to CPU and memory) and can take a long time to provide estimates compared to the other approaches (factor

10 to 100). The computational overhead made an application of MLE to our extensive datasets infeasible.

We used the following configuration for the experimental comparison: SDL uses the average utilization and throughput of the complete experiment length as input and apportions the aggregate utilization between workload classes using the observed average response time as described in Brosig et al. (2009). UR uses a standard non-negative least-squares regression algorithm (see `lsqnoneg`). The parameterization of KF follows the guidelines suggested by Zheng et al. (2008) (D1: state covariance Q=0.0025, observation covariance R=0.1; D2: Q=0.0001 R=0.0001). We also applied a moving average filter to the resulting demands with a window size of 10 minutes. MO uses the recursive optimization algorithm proposed by Menascé (2008). In contrast, LO executes the optimization algorithm once with the complete observation traces as input. RR comes in two different versions: one for FCFS (Kraft et al., 2009) and one for PS scheduling (Pérez et al., 2013). We used the FCFS variant for dataset D1 and the PS variant for dataset D2.

Abbreviation	Estimation Approach
SDL	Service Demand Law (Brosig et al. (Brosig et al., 2009))
UR	Utilization regression (Rolia et al. (Rolia and Vetland, 1995))
KF	Kalman filter (Kumar et al. (Kumar et al., 2009a))
MO	Menascé optimization (Menascé, 2008)
LO	Liu optimization (Liu et al. (Liu et al., 2006))
RR	Response time regression (Kraft et al. (Kraft et al., 2009))
GS	Gibbs Sampling (Wang et al. (Wang et al., 2012))

Table 5.5: Estimation approaches considered in the experimental evaluation.

To assess the accuracy of the estimation approaches, we rely on the mean relative demand error E_d as error metric. Equation 5.4 shows the definition of E_d. C is the number workload classes, D_c^{est} the estimated resource demand of class c and D_c^{act} the actual resource demand of class c.

$$E_d = \frac{1}{C} \sum_{c=1}^{C} \left| \frac{D_c^{est} - D_c^{act}}{D_c^{act}} \right| \tag{5.4}$$

In some of the experiments we also use the relative utilization error E_u and relative response time error E_r to show the effect of incorrect demand

estimates on the predicted utilization and response time. Equation 5.5 shows the definition of the utilization error

$$E_u = \left| \frac{\sum_{c=1}^{C} \lambda_c * D_c^{est} - U}{U} \right|. \tag{5.5}$$

C is the number workload classes, λ_c the observed throughput of class c, D_c^{est} the estimated resource demand of class c and U is the observed utilization. Equation 5.6 shows the definition of the response time error

$$E_r = \frac{1}{C} \sum_{c=1}^{C} \left| \frac{R_c^{cal} - R_c^{act}}{R_c^{act}} \right|. \tag{5.6}$$

C is the number of workload classes, R_c^{act} is the average observed response time of class c, and R_c^{cal} is the predicted average response time of class c obtained with Mean Value Analysis (MVA) (Bolch et al., 2006).

5.2.2.1 RQ1: Length of Sampling Interval

The sampling interval defines the time period for which average statistics, e.g., of utilization or response times, are calculated. The total experiment length is divided into fixed-length sampling intervals. In this experiment, we used observation traces from datasets D1 and D2 with medium load (U = 50%) and one workload class. The average statistics are calculated for different sampling intervals, varying between one second and and two minutes. A sampling interval of one second is usually the lowest resolution for operating system monitoring tools (e.g., the `sar` utility for obtaining resource usage statistics). The maximum sampling interval of two minutes is chosen so that there are at least 30 samples per experiment run.

From the considered estimation approaches, only UR, KF, MO, and LO rely on average statistics. To be concise, we leave out the results for RR and GS, which are based non-aggregated measurements of individual requests, and SDL, which always takes the average over the complete observation period. As expected, the latter estimation approaches are not influenced by the length of the sampling interval.

Figure 5.2 shows the relative demand errors E_d for dataset D1 under medium load (U = 50%) and one workload class. All four estimation approaches are negatively influenced by small sampling intervals. Under small sampling intervals with one second, estimation accuracy of LO suffers the most and the error decreases only slowly with longer sampling intervals. However, the

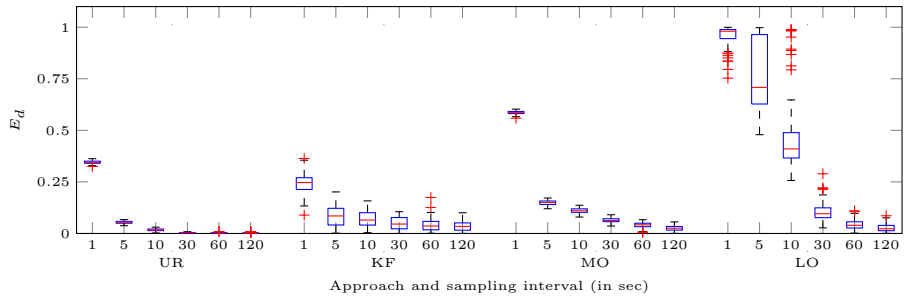


Figure 5.2: Boxplot of demand estimation error E_d for different sampling intervals (dataset D1, load level $U = 50\%$, number of workload classes $C = 1$).

relative error is comparable to the other approaches in case of 60 and 120 seconds sampling intervals (below 5%).

In addition to dataset D1, Table 5.6 shows the results for dataset D2. This table includes an additional column containing the mean number of requests N observed during each sampling interval. The average resource demands in dataset D2 were by a magnitude smaller than in dataset D1. Therefore, more requests are observed during each sampling interval and the peaks at the one second sampling interval are smaller in D2. However, we can again observe that LO shows the highest relative error for the one second sampling interval.

The influence of the length of the sampling interval can be explained by *end-effects* due to requests which are fully attributed to one sampling period, although they start and end in different intervals. For linear regression this has been identified before by Rolia and Lin (1994) and Rolia and Vetland (1995) as one source of inaccuracy. Zhang et al. (2007) come to the conclusion that longer sampling intervals improve the accuracy of regression-based approaches. However, in practice, the maximum length of the sampling interval is usually limited because it increases the required experiment length and may hinder the ability of the estimator to adapt to changes in the resource demands. Given that a good choice for the sampling interval always depends on the length of the resource demands, one should ensure that sufficient requests are observed in each sampling interval. The results in Table 5.6 suggest that a sampling interval length where on average $N > 60$ requests are observed yields acceptable estimates for all approaches.

			$mean[E_d]$ ($std[E_d]$)			
		N	UR	KF	MO	LO
D1	1s	1.00	34.54 (0.74)	24.35 (4.89)	58.67 (0.79)	95.82 (4.80)
	5s	4.99	5.43 (0.66)	8.44 (5.03)	14.91 (1.16)	77.20 (17.35)
	10s	9.99	1.74 (0.55)	7.00 (4.01)	11.03 (1.18)	46.55 (17.11)
	30s	29.97	0.31 (0.20)	4.80 (3.17)	6.37 (1.06)	10.42 (4.38)
	60s	59.95	0.23 (0.17)	4.20 (2.91)	4.04 (1.26)	4.31 (2.31)
	120s	119.90	0.19 (0.17)	3.61 (2.38)	2.57 (1.23)	2.68 (1.82)
D2	1s	11.58	8.60 (6.97)	13.41 (15.89)	15.04 (3.36)	27.31 (26.32)
	5s	57.89	0.59 (0.32)	5.66 (4.05)	9.79 (1.19)	3.42 (3.31)
	10s	115.79	0.60 (0.59)	3.51 (2.10)	8.78 (0.82)	2.01 (1.72)
	30s	347.36	0.77 (0.66)	1.41 (0.74)	8.03 (0.79)	1.41 (1.13)
	60s	694.40	0.80 (0.56)	1.73 (1.24)	7.82 (0.83)	1.38 (1.09)
	120s	1387.79	0.91 (0.81)	1.38 (1.50)	7.87 (0.79)	1.30 (1.04)

Table 5.6: Mean and standard deviation of demand estimation error E_d for different sampling intervals (dataset D1 and D2, load level $U = 50\%$, number of workload classes $C = 1$). N denotes the average number of requests observed in one sampling interval.

5.2.2.2 RQ2: Number of Samples

In this experiment, we employed dataset D1 and reduced the number of samples used for resource demand estimation from 3600 to 600. This corresponds to an experiment length of ten minutes. Dynamic, self-adaptive systems require an estimator to keep up with frequent changes. Therefore, we argue that an estimator should also be able to converge to a stable value in shorter time frames.

	N	SDL	UR	KF	MO	LO	RR	GS
$mean[E_d]$	600	0.13	0.79	7.3	4.1	6.6	2.5	4.9
	3600	0.023	0.23	4.2	4	4.3	1.4	4.8
stat. sig. (95%)		✓	✓	✓		✓	✓	
p-value		1.2e-24	4.2e-15	4.8e-129	0.81	1.3e-05	5.3e-05	0.75

Table 5.7: Mean demand estimation error E_d for different number of samples N (dataset D1, load level $U = 50\%$, number of workload classes $C = 1$).

Table 5.7 shows the results for dataset D1. Differences in the mean relative resource demand errors from the experiment runs are tested for statistical significance using a non-paired T-test with a 95% confidence level. The estimation approaches SDL, UR, KF and LO exhibit a significant dependency

on the number of available samples. With $N = 600$ they show a decreased accuracy compared to $N = 3600$. However, all approaches still yield results with acceptable accuracy (below 10%).

5.2.2.3 RQ3: Number of Workload Classes

A higher number of workload classes makes the estimation problem more complex since more variables need to be estimated. In RQ3, we analyze the sensitivity of the considered estimation approaches to the number of workload classes. The analysis is structured into three subquestions: RQ3.1 compares the relative demand errors of experiments with different number of workload classes, RQ3.2 explores properties of the dataset that influence the estimation accuracy in case of several classes, and RQ3.3 tests the behavior of the estimation approaches if the number of classes is scaled out.

RQ3.1: Comparison of relative demand errors We now compare the relative demand errors from runs with three different number of workload classes. We used a subset of dataset D1 containing samples for 1, 2 and 5 classes at a load level of 50% (in total 300 repetitions) and D2 for 1, 2, and 3 also at 50% (in total 70 repetitions).

	C	SDL	UR	KF	MO	LO	RR	GS
	1	0.023	0.231	4.2	4.04	4.31	1.44	4.76
$mean[E_d]$	2	127	27.7	88.3	83.4	98.8	8.56	93.4
	5	153	59.8	110	97.2	120	18.2	111
stat. sig. (95%)		✓	✓	✓	✓	✓	✓	✓
p-value		2.52e-04	6.53e-17	7.59e-04	1.12e-03	8.62e-04	1.34e-04	1.61e-03

(a) Dataset D1

	C	SDL	UR	KF	MO	LO	RR	GS
	1	0.833	0.8	1.73	7.82	1.38	1.11	2.87
$mean[E_d]$	2	1.02	12.8	3.84	5.23	4.33	1.85	3.56
	3	2.07	24.1	4.01	5.56	4.94	3.44	4.9
stat. sig. (95%)		✓	✓	✓	✓	✓	✓	✓
p-value		5.96e-06	1.02e-05	0.0368	5.05e-05	0.033	5.56e-05	0.0081

(b) Dataset D2

Table 5.8: Mean relative demand error E_d for number of workload classes $C = \{1, 2, 5\}$ (load level $U = 50\%$).

Table 5.8 shows the results for datasets D1 and D2. We used a single factor Analysis of Variance (ANOVA) with a confidence level of 95% to test for significant differences in E_d with different number of workload classes. The hypothesis that E_d is influenced by the number of workload classes cannot be rejected for any of the considered approaches. However, there are clear differences in the quantitative effect on E_d between both datasets. While most estimation approaches yield relatively accurate results for dataset D2 (E_d mostly below 10% except for UR), we consider the results for dataset D1 insufficient for most use cases. With 2 or 5 workload classes, the estimated resource demand largely deviates from the actual one by more than 50% in most cases on dataset D1. We analyze the reasons for these high deviations in RQ3.2.

UR shows a degraded accuracy for multiple workload classes on both datasets. A deeper analysis of the resulting estimates show that the estimates converge very slowly compared to the other estimation approaches. The linear regression is done based on measurements from approximately 60 measurement intervals, which is assumed to be sufficient for the considered number of workload classes. However, the performance of UR heavily depends on the workload (Stewart et al., 2007). We explain the poor accuracy of UR in our experiments with too few variations in the workload. Given that the utilization is kept at a fixed level during the experiments, UR can only explore a limited region of the complete space.

RQ3.2: Correlation Analysis The comparison in RQ3.1 shows a largely degraded accuracy of most estimation approaches on D1 with multiple workload classes. Given that high variances in E_d were observed between experiment runs, we performed a correlation analysis testing the influence of different properties of a sample set on E_d.

The property $mean[Q]$ stands for the mean queue length Q observed during an experiment run. $min[X * D]$ takes the minimum of the mean throughput X and the average resource demand D over all workload classes. A low value of $min[X * D]$ is an indicator that the workload includes classes with a small contribution in relation to the other classes. These are also called *insignificant flows*. $std[D]$ is the standard variance of the service demands. If this value is high, the mean service demands of workload classes are very diverse.

Table 5.9 shows the results of the correlation analysis. We used the Spearman's rank correlation coefficient (denoted with ρ) in order to be able to identify non-linear correlations. Table 5.9 summarizes the correlations of three properties of the sample set which were identified to influence E_d.

	C		SDL	UR	KF	MO	LO	RR	GS
$mean[Q]$	1	ρ	-0.042	-0.14	0.065	-0.42	-0.5	0.27	0.65
		p-value	0.68	0.16	0.52	1.3e-05	2.1e-07	0.0062	0
	2	ρ	**0.71**	0.27	0.67	0.68	**0.73**	0.46	**0.75**
		p-value	0	0.0073	0	0	0	1.8e-06	0
	5	ρ	0.52	0.25	0.65	0.63	0.66	0.39	0.63
		p-value	5.6e-08	0.013	0	0	0	5.8e-05	0
$min[X*D]$	1	ρ	-	-	-	-	-	-	-
		p-value	-	-	-	-	-	-	-
	2	ρ	-0.46	-0.54	-0.55	-0.56	-0.44	-0.52	-0.45
		p-value	2.2e-06	1.2e-08	5.2e-09	2.6e-09	7.7e-06	6.9e-08	4.5e-06
	5	ρ	-0.45	-0.44	-0.48	-0.47	-0.49	-0.61	-0.5
		p-value	4e-06	6.9e-06	5.7e-07	1.4e-06	3.6e-07	0	1.9e-07
$std[D]$	1	ρ	-	-	-	-	-	-	-
		p-value	-	-	-	-	-	-	-
	2	ρ	**0.91**	0.35	**0.88**	**0.89**	**0.88**	0.52	**0.9**
		p-value	0	0.0004	0	0	0	5.1e-08	0
	5	ρ	**0.72**	0.37	**0.78**	**0.8**	**0.8**	0.44	**0.79**
		p-value	0	0.0002	0	0	0	4.8e-06	0

Table 5.9: Correlation analysis results (dataset D1, load level $U = 50\%$). Entries with $\rho > 0.7$ are in bold letters.

We identified the highest correlations ($\rho > 0.7$) for SDL, KF, MO, LO, RR, GS with $std[D]$, i.e., if the differences between the resource demand of workload classes is higher, the relative demand error E_d is also higher. In these cases, the underlying model is based on the response time equation $R = D/(1-U)$. Assuming an open workload, this equation is only applicable to multi-class queues with FCFS scheduling if the service time of each workload class is equal (Harchol-Balter, 2013). This requirement does not hold for dataset D1. The higher the variation of the resource demands between workload classes is the more it lessens the estimation accuracy of the estimation approaches. The impact of this violated assumption increases if the mean queue length $mean[Q]$ in an experiment run is higher. The high correlations show that when using response times for resource demand estimation, it is important to ensure that the estimator is based on the correct scheduling strategy assumptions.

Furthermore, we could observe moderate negative correlations for $min[X*D]$ for all estimation approaches. That mean if in an experiment run, there exists a workload class with a low the total resource demand $X * D$ compared to the other workload classes, the relative demand error increases. We conclude that all considered estimation approaches are sensitive to workload classes with a low total resource demand (sometimes also called insignificant flows (Pacifici et al., 2008).

RQ3.3: High number of workload classes In Section 5.2.2.3, the results indicate an influence of the number of workload classes on the accuracy of certain estimation approaches. In the following experiment we consider scenarios with a higher number of workload classes than before. We employed the micro-benchmarks used to obtain dataset D2 and varied the number of workload classes between 5 and 20. In total, we performed 40 experiment runs.

	C	SDL	UR	KF	MO	LO	RR	GS
$mean[E_d]$	5	1.24	20.5	2.89	3.51	2.44	1.78	5.17
	10	2.53	36.2	3.99	3.36	2.39	3	8.55
	15	2.86	56.9	4.32	3.44	3.11	3.52	12.5
	20	2.99	57.8	5.33	4.04	3.28	3.58	13.6
stat. sig. (95%)		✓	✓	✓		✓	✓	✓
p-value		6.59e-09	3.58e-09	5.02e-05	0.303	0.00151	6.61e-06	9.76e-10

Table 5.10: Mean relative demand error E_d for high numbers of workload classes $C = \{5, 10, 15, 20\}$ (dataset D2, load level $U = 50\%$).

Table 5.10 shows the results from this experiment. We used a single factor ANOVA with a confidence level of 95% to test for significant differences in E_d with a different number of workload classes. Several estimation approaches (SDL, KF, MO, LO, RRPS) do not show a clear dependence on the number of workload classes in the considered range. In these cases, we could not observe a statistically significant difference in the estimation errors regarding the utilization and response times. The results for UR support the findings with multiple workload classes in Section 5.2.2.3.

5.2.2.4 RQ4: Load Level

We now explore the sensitivity of the estimation approaches under different system load levels using measurement traces with low, middle and high load. Dataset D1 contains data of runs with an average utilization of 10%, 50% and 90% (in total 300 repetitions), dataset D2 has runs with an average utilization of 20%, 50% and 80% (in total 30 repetitions). Only the workload intensity changed between the experiments runs, other factors were kept fixed.

Table 5.11 shows the mean relative demand error E_d for dataset D1 and D2 with sample sets from low, middle and high load. We used a single factor Analysis of Variance (ANOVA) with a confidence level of 95% to test for statistically significant differences in E_d between the load levels.

The results for dataset D1 suggest an influence of the load level on all estimation approaches except SDL. Apart from SDL, all approaches have a higher

	U	SDL	UR	KF	MO	LO	RR	GS
	10%	0.0232	0.219	2.36	0.81	0.427	0.434	3.39
$mean[E_d]$	50%	0.023	0.231	4.2	4.04	4.31	1.44	4.76
	90%	0.0279	0.843	33.1	5.33	90.5	1.75	24.2
stat. sig. (95%)			✓	✓	✓	✓	✓	✓
p-value		0.167	6.86e-67	1.41e-22	3.53e-95	5e-273	3.13e-16	1.93e-13

(a) Dataset D1

	U	SDL	UR	KF	MO	LO	RR	GS
	20%	2.85	2.71	2.37	2.98	1.8	1.05	3.17
$mean[E_d]$	50%	0.833	0.8	1.73	7.82	1.38	1.11	2.87
	80%	0.461	0.515	4.55	12.4	5.39	0.825	7.12
stat. sig. (95%)		✓	✓		✓	✓		✓
p-value		9.38e-08	2.55e-07	0.0606	5.49e-19	0.0146	0.505	0.000554

(b) Dataset D2

Table 5.11: Mean demand estimation error E_d for different load levels U and number of workload classes $C = 1$).

mean E_d at 90% utilization compared to 50% and 10%. Most conspicuous are the high relative errors (above 20%) for KF, LO and GS at high load. We explain these inaccuracies with underlying model assumptions of these estimation approaches, which are violated at high load levels. GS is based on a closed queueing model while the queueing simulator used to obtain dataset D1 executed an open workload. KF and LO use the response time equation $R = D/(1 - U)$ which is highly non-linear above 90% CPU utilization. We explain the observed inaccuracies of KF and LO with deficiencies of the underlying estimation algorithms which results in a reduced estimation accuracy in highly non-linear regions. While MO is similar to LO regarding the underlying model, MO uses an iterative optimization algorithm which seems to be more stable in high load scenarios.

On dataset D2 the differences between the estimation approaches at high loads are smaller in comparison to D1. KF, MO, LO and GS are again negatively influenced by the high utilization. However, with 80% the utilization is further away from the critical region close to 100% utilization. In summary, we conclude that it may be beneficial to avoid high-load situations (above 80%) during resource demand estimation, or best use one of the SDL, UR or RR approaches.

5.2.2.5 RQ5: Collinear Workload Classes

In the following experiments, the influence of collinear workload classes is evaluated. For determining the level of collinearity, we use the Variance Inflation Factor (VIF) which is defined as $VIF_i = \frac{1}{1-R_i^2}$. R_i^2 is the coefficient of determination if we calculate the regression of $X_i = \sum_{j=1}^{j \leq N, j \neq i} \beta X_j$. Based on the rule of thumb proposed by Kutner et al. (2003), we assume a strong collinearity between workload classes if $VIF_i > 10$ for the observed throughput.

The traces in datasets D1 and D2 both do not contain clearly collinear workload classes. The maximum VIF observed are 1.1772 and 3.1602. Therefore, we adapted the workload used for generating D2 so that one job of one workload class is followed by a job from another workload class with a certain probability p_c (including a certain think time between the two workload classes). The experiment is executed with $p_c = 0.33$) and $p_c = 1.0$. The observed VIF is on average 1.1624 and 26.2972, respectively. So for the case of $p_c = 1.0$ we can safely assume a strong collinearity between workload classes.

Collinearity		SDL	UR	KF	MO	LO	RR	GS
$mean[E_d]$	Low	2.68	39.7	3.86	5.63	5.39	3.26	4.81
	High	2.75	111	3.64	6.83	5.21	3.43	5.47
stat. sig. (95%)			✓		✓			
p-value		0.854	0.00234	0.675	0.00447	0.787	0.627	0.54
$mean[E_u]$	Low	0.0045	0.0457	1.94	4.72	0.735	0.704	2.12
	High	0.00123	0.0534	1.76	5.15	0.792	1.02	2.3
$mean[E_{rt}]$	Low	4.63	43.3	7.42	4	8.95	2.7	4.62
	High	5.47	120	7.33	3.99	9.9	3.22	4.66

Table 5.12: Sensitivity to collinearity in throughput observations.

Table 5.12 shows the results from experiments with low and high levels of multicollinearity. We used a non-paired T-test with a confidence level of 95% to test for statistically significant differences in E_d. The only estimation approach that is clearly influenced by high levels of multicollinearity in the workload is the UR approach. This issue has also been discussed in Pacifici et al. (2008) proposing different approaches to improve the robustness of UR in case of collinear workload classes.

5.2.2.6 RQ6: Missing Jobs in Workload Model

On real systems, it can be difficult to capture all tasks executed by the application, middleware system, or operating system in a workload model. Per-

formance engineers are often not aware of background processes that cannot be directly attributed to the processing of user requests and that may happen at points in time difficult to foresee. In order to evaluate the sensitivity of estimation approaches to missing workload classes, we adapted the micro-benchmarks used to obtain dataset D2. We implemented a workload consisting of 3 workload classes representing the user requests and one class representing the background process. The user requests incurred an average CPU utilization of $U = 50\%$. The intensity of the background job was varied between $U = 5\%, 10\%, 20\%, 30\%$. We executed a total of 40 experiment runs. The estimation approaches were only provided observations from the three workload classes processing user requests as input.

	U_b	SDL	UR	KF	MO	LO	RR	GS
$mean[E_d]$	5%	9.32	34.5	9.33	2.28	16.7	4.61	4.82
	10%	18.2	40	15.8	3.03	29.3	6.34	6.57
	20%	34.4	64.5	27.6	9.15	49.4	13.5	12
	30%	49.6	88.3	35.9	15.3	61.3	20.1	17.7
stat. sig. (95%)		✓	✓	✓	✓	✓	✓	✓
p-value		8.16e-50	6.17e-08	2.23e-21	7.6e-32	1.57e-48	6.15e-33	1.1e-15
$mean[E_u]$	5%	0.00104	0.0517	5.25	9.23	1.83	5.28	8.11
	10%	0.00369	0.0685	7.75	13.1	3.24	9.1	10.2
	20%	0.00404	0.0898	12.9	18.8	7.12	14.7	16.7
	30%	0.00413	0.123	17.1	22.9	13.5	18.8	21.1
$mean[E_{rt}]$	5%	15.1	38	11	4.14	21.6	2.77	5.46
	10%	26.6	50.1	15.4	4.08	35.3	3.34	3.9
	20%	48.7	87	21.7	4.82	54	3.79	4.13
	30%	72.5	124	22.8	5.3	56.3	4.33	4.64

Table 5.13: Demand error E_d, utilization error E_u and response time error E_r when system executes background job with intensity U_b.

Table 5.13 contains the results for this experiment. We used a single factor ANOVA with a confidence level of 95% to test for significant differences in E_d when the intensity of the background job is varied. All estimation approaches are significantly influenced by the hidden workload class. However, the influence seems to be stronger on approaches based on the CPU utilization (SDL, UR, KF, LO) compared to the other methods using response times. The direct influence on the utilization measurements seem to have a stronger influence on the estimation accuracy than the indirect effects of the background job on the observed response times. Table 5.13 also contains the relative errors E_u and E_{rt} to show the influence on predictions when using the estimated resource

demands.

5.2.2.7 RQ7: Delays during Processing

Experiment RQ7 simulates the situation when the processing of one request may consist of several visits to the CPU resource with a certain delay between the visits. The delay may be caused, e.g., by waiting for software resources (e.g., thread or connection pool), or for data from other hardware resources (e.g., hard disk or network). We adapted the micro-benchmarks used to generate dataset D2, by splitting up the Fibonacci calculation into two parts with equal length and inserting a delay period. We varied the delay period between 25ms, 75ms, and 125ms. In total we have 30 experiment runs.

	Delay	SDL	UR	KF	MO	LO	RR	GS
	25ms	5.82	19.5	5.27	6.19	3.56	6.52	6.28
$mean[E_d]$	75ms	14.8	19.8	18.2	14.2	21.2	14.8	15.4
	125ms	22.3	12	29.6	21.3	38.9	22.4	21.9
stat. sig. (95%)		✓		✓	✓	✓	✓	✓
p-value		8.7e-30	0.0771	1.31e-23	9.35e-29	1.06e-31	2.95e-27	5.19e-13
	25ms	0.00374	0.0283	1.35	1.55	0.252	2.24	1.44
$mean[E_u]$	75ms	0.00156	0.0227	2.07	4.12	0.669	8.14	6.12
	125ms	0.00214	0.0254	4.88	9.17	1.59	13.7	12
	25ms	1.33	21.7	3.93	3.9	3.65	2.07	4.04
$mean[E_{rt}]$	75ms	11.1	26.1	10.6	4.3	15.1	1.81	6.02
	125ms	19.1	25.5	18.2	4.66	30.3	1.99	4.28

Table 5.14: Demand error E_d, utilization error E_u, and response time error E_r when the jobs are interrupted by wait periods.

Table 5.14 shows the result for the experiment. We used a single factor ANOVA with a confidence level of 95% to test for significant differences in E_d when varying the length of the delays. The relative error E_d of all estimation approaches except UR is negatively influenced by the additional delay. UR is the only considered approach that is not relying on response time measurements. While in the case of SDL, KF, and LO E_{rt} are mainly impacted, it is E_u for MO, RR, and GS.

5.2.2.8 RQ8: Execution Time

We measured the execution times of the estimation approaches on dataset D1 to compare the computational effort associated with each approach. Dataset

D1 consists of 900 measurement traces each containing observations of 3600 individual requests observed over a simulation time of one hour.

	C	U	SDL	UR	KF	MO	LO	RR	GS
		10%	1.1	1.0	0.3	671.6	20.9	77.1	19413.7
	1	50%	0.5	0.4	0.2	873.1	22.9	75.9	19619.7
		90%	0.5	0.4	0.2	2288.0	21.5	78.8	20266.9
		10%	0.6	0.6	0.4	1028.8	23.1	80.0	42910.0
$mean[T]$	2	50%	0.6	0.5	0.2	1221.5	30.0	80.5	42685.1
		90%	0.6	0.5	0.2	3418.2	38.8	83.7	45921.4
		10%	0.8	0.7	0.6	2073.5	41.9	89.4	251675.4
	5	50%	0.8	0.7	0.5	2213.8	42.3	92.5	138163.4
		90%	0.8	0.7	0.5	6389.0	88.0	96.9	138735.7

Table 5.15: Mean execution time T (in milliseconds) partitioned by number of workload classes C and load level U.

Table 5.15 contains the average execution times T for each estimation approach. SDL, UR, and KF have a low computational effort, the execution times for a single measurement trace is on average below 1 millisecond. LO and RR have a moderate computational effort, on average between 20 and 100 milliseconds. The higher effort of RR compared to UR can be explained with the lack of measurement traces for the queue length seen on arrival in dataset D1. RR first needs to calculate this metric based on response times and arrival times. MO and GS show a significantly higher computational effort, on average between 0.5 seconds and 4 minutes. Although based on the same optimization algorithm, MO is slower compared to LO because it executes the optimization recursively for each new sample, while LO runs the optimization once for the complete measurement trace. GS has a high execution time compared to the other approaches because it needs to approximate the normalizing constant of state probabilities, which is very costly operation (Wang and Casale, 2013).

5.2.3 Results Summary

In this section, we summarize the results of our experiments. We identified the following sensitivities:

RQ1 When using estimation approaches based on time-aggregated observations (e.g., UR, KF, MO, LO), the length of the sampling interval is an important parameter that needs to be adjusted to the system under study. A good sampling interval length depends on the response times of requests and the number of requests observed in one interval. The sampling

interval should be significantly larger than the response times of requests to avoid end-effects and it should be long enough to be able to calculate the aggregate value based on the observations of a significant number of requests (more than 60 requests per sampling interval provided good results in our experiments).

RQ2 Most estimation approaches (except MO and LO) were negatively influenced when reducing the experiment length to 10 minutes (i.e., 10 samples). However, they still yielded results with acceptable accuracy (relative demand error below 8%).

RQ3 All estimation approaches are sensitive to the number of workload classes. The linear regression method UR that only uses utilization and throughput observations generally yielded a degraded accuracy in our experiments with several workload classes. Observations of the response times of requests can help to improve the estimation accuracy significantly (RQ3.1) even in situations with a very high number of workload classes (RQ3.3). However, it is crucial to ensure that the modeling assumptions of the estimation approaches using response times are fulfilled as they are highly sensitive to violated assumptions, e.g. wrong scheduling strategies (RQ3.2). Furthermore, insignificant flows can impair resource demand estimation (RQ3.2). Workload classes with a small contribution to the total resource demand of a system should therefore be excluded from resource demand estimation.

RQ4 When a system operates at a high utilization level (80% or higher), the estimation approaches KF, MO, LO and GS may yield inaccurate results.

RQ5 Collinearities in throughput observations of different workload classes impairs the estimation accuracy of UR. While it correctly estimates the total resource demand, the apportioning between workload classes is wrong. The other evaluated estimation approaches did not show a sensitivity to collinearities in throughput observations.

RQ6 Estimation approaches relying on response time observations (e.g., MO, RR and GS) are more robust to missing workload classes than approaches using utilization observations.

RQ7 Delays due to non-captured software or hardware resources has a strong influence on the estimation accuracy of estimation approaches based on observed response times. While some estimation approaches – e.g., Liu et al. (2006), Menascé (2008), and Zhang et al. (2002) – consider the

scenarios where multiple resources contribute to the observed end-to-end response time, only Pérez et al. (2013) have considered contention due to software resources in their estimation approach.

RQ8 There are significant differences in the computational complexity of the different estimation approaches. On our datasets, the estimation took between under 1 millisecond and up to 20 seconds depending on the estimation approach. When using resource demand estimation techniques on a production system (e.g., for online performance and resource management), the computational effort needs to be taken into account (especially in data centers with a large number of systems).

In summary, our evaluation shows that using response times can improve the accuracy of the estimated resource demands significantly compared to the traditional approach based on the Utilization Law using linear regression, especially in cases with multiple workload classes (see Section 5.2.2.3). However, approaches employing response time measurements are very sensitive if assumptions of the underlying mathematical model are violated (e.g., wrong scheduling strategy in Section 5.2.2.3, or delayed processing in Section 5.2.2.7).

5.3 Library for Resource Demand Estimation (LibReDE)

Section 5.1 provides an overview of the approaches to resource demand estimation that have been proposed in the last decades. While these approaches offer a great variety of different options for performance engineers, their practical use in industry is still limited, especially in autonomic systems. We see two major factors hindering the adoption of such estimation approaches:

- While the estimation approaches are documented in the literature, to the best of our knowledge, ready-to-use implementations are not generally available. Potential users need to develop their own implementations of each estimation approach.

- Choosing the right approach for a given system under study, requires a deep expertise of its underlying statistical techniques and its assumptions. While the systematization in Section 5.1 and the experimental comparison in Section 5.2 helps with this decision by documenting the different assumptions and factors impacting the accuracy of estimation approaches, it is not always possible to determine an approach in advance. For instance in a self-aware system, the actual performance model is learned at run-time. In consequence, we need to delay the decision on

which approach to use to the system run-time. Furthermore, the system may change over time requiring a dynamic switch-over to a different estimation approach.

In this section, we present a library for resource demand estimation we have developed called LibReDE. It provides ready-to-use implementations of eight common estimation approaches (see Section 5.3.3) and is available as open-source software under the Eclipse Public License (EPL). Furthermore, LibReDE realizes a complete estimation method automating the decision which estimation approach to use. The description in this section focuses on this estimation method.

The main idea of LibReDE is to base the resource demand estimation on *multiple statistical techniques* combined with a *feedback loop* to improve the accuracy of the estimation by iteratively: i) adapting the estimation problem, ii) selecting suitable statistical methods to be applied, and iii) optimizing the configuration parameters of each method. We use *cross-validation* techniques with an error metric based on the deviation between the observed response times and utilization, on the one hand, and the respective predicted metrics using the resource demand estimates, on the other hand. To the best of our knowledge, our proposed approach is the first to apply multiple statistical techniques at run-time automatically combining, weighting and iteratively refining their results in a feedback loop, to produce as accurate estimates as possible.

LibReDE can be used as a model variable characterization agent in our reference architecture for online model learning (see Section 4.2.5). For this purpose, it needs to be able to derive all information required for resource demand estimation solely from the models contained in the performance model repository and empirical monitoring data. In Section 5.3.2 we describe a model-to-model transformation for DML in order to automatically derive the input information required by LibReDE.

The LibReDE tool has been first published in Spinner et al. (2014a). In this section, we extend this description with a complete estimation method supporting the automatic selection of estimation approaches.

5.3.1 Method Overview

Figure 5.3 provides an overview of our end-to-end estimation method. The estimation method consists of the following steps:

1. Given a performance model for which resource demands need to be quantified, we first derive a workload description which is independent of the actual performance modeling formalism. Model transformation

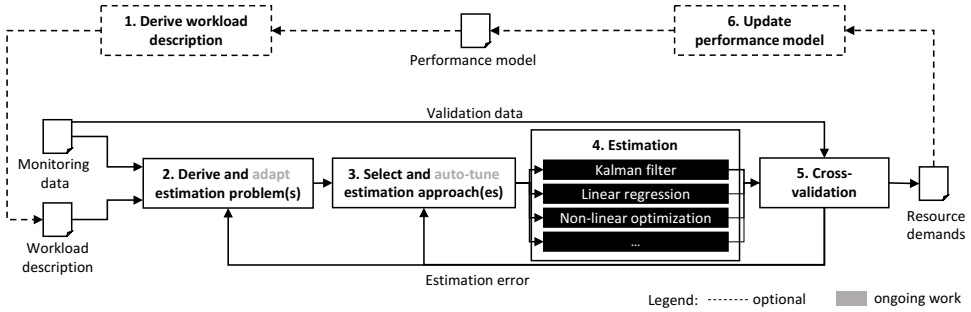


Figure 5.3: Overview of estimation method.

techniques from MDD (see Section 2.2.1) can be used to automate this step. LibReDE comes with a model-to-model transformation for the DML meta-model (as described in Section 5.3.2). This step is optional. A user may also provide the workload description directly with the meta-model described in Section 5.3.2.

2. In the next step, LibReDE derives possible mathematical formulations of the estimation problem based on the workload description and the input monitoring data. Different strategies to derive estimation problems are described in Section 5.3.3. This step also provides potential for further adaptations to the structure of the estimation problem. For instance, services with an insignificant overall resource demand may be merged with other services in order to improve the statistical stability of the estimation.

3. LibReDE selects suitable statistical estimation techniques to be applied based on checking the pre-conditions and feedback on the accuracy from previous iterations. This step may select multiple methods. In addition, auto-tuning strategies may be employed to improve certain configuration parameters of estimation methods.

4. The estimation techniques selected in the previous step are all called with the current monitoring data. The techniques should support iterative execution, so that they keep their estimation state between invocations. LibReDE calls each estimation technique k-times on different subsets of the monitoring data to allow for k-fold cross-validation.

5. LibReDE evaluates the accuracy of the estimated resource demand using a k-fold cross-validation scheme. The estimates of the technique with the lowest estimation error are selected.

6. The resource demands in the performance model are updated with the latest estimates.

The method is designed to be applied iteratively to support online estimation scenarios. Step 1 needs to be repeated only if the structure of the performance model changes. In this case, the statistical estimation techniques need to start from scratch as well. Step 2, 3 and 5 should be repeated in regular intervals to adapt the estimation problems, update the selected approaches, or auto-tune configuration parameters. Step 4 is repeated in short intervals as soon as new monitoring data gets available.

Workload Description All approaches to resource demand estimation require a description of the resources and services (also called workload classes) of a system under study. Furthermore, estimation approaches whose formulations are based on QNs with multiple service stations additionally require information on the flow of requests in a system (i.e., the visit counts of requests at individual resources). LibReDE expects an EMF-based model of the workload description as input. A user may manually specify this model, or it may be derived automatically from an existing performance model. Our meta-model is independent of a concrete performance modeling formalisms and model-to-model transformations from other metamodels may be supported in the future as well – e.g., QNs or QPNs.

Figure 5.4 shows the classes of our workload description meta-model. The abstract base class `ModelEntity` defines an attribute `name` that may contain a user-defined string for easier readability of the workload description and the result output of LibReDE. The root element `WorkloadDescription` contains a list of services and resources. A `Resource` may represent any hardware or software resource in a system. The number of servers and a scheduling strategy may be specified for a resource. A `Service` (or workload class) represents types of requests with similar resource demanding behavior. A service may be marked as a background service subsuming all resource demanding activities on a system not directly related to the processing of requests. Only one service accessing a resource may be marked as a background service. LibReDE currently assumes a Poison arrival process for services. This limitation may be relaxed as part of future work.

A service may contain multiple tasks. A `Task` can be either a `ResourceDemand` or an `ExternalCall`. A resource demand references the resource from which it

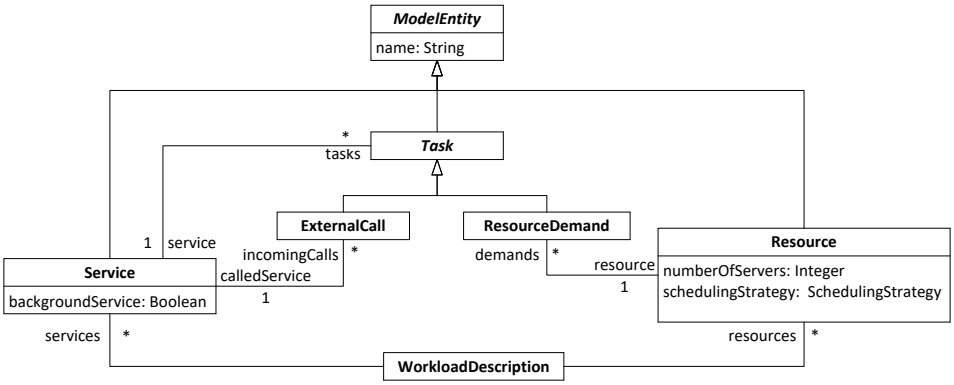


Figure 5.4: Workload description meta-model.

requires service. An external call references another service in the workload description that is called one or several times for each incoming request. The actual number of calls needs to be provided as part of the observation data. The ordering of tasks in a service is irrelevant for the resource demand estimation.

Observation Data The observation data needs to be provided as time series data. LibReDE expects the following additional meta-data for each time series:

- The metric of the observation (e.g., response time or utilization).
- The measurement unit of the observations.
- A reference to a model entity in the workload description.
- The type of aggregation if the observation data is sampled over time (e.g., summation or average).
- The sampling interval length (optional).

The monitoring data must be available as time series data with associated timestamps for each sample. The library can work on time series with individual events (e.g., arrival times and response times of individual requests) or on equidistant sampled time-aggregated data (e.g., average response times or average throughput).

5.3.2 Derivation of Workload Description

In this section, we describe our model-to-model transformation from a DML instance to a workload description as expected by LibReDE. The input DML contains no concrete parameter values for the resource demand distributions. In the following, we assume that the type of distribution is determined in a separate step and the goal is to estimate the average resource demand based on the observation data.

Prerequisites We first define a number of sets and functions helping us to describe the transformation rules. Given the set of component types C defined in a DML model, the function $instances : C \rightarrow SET(A)$ returns the set of all component instances A of a component $c \in C$. A DML model describes a containment tree of nested component instances (i.e., assembly contexts). For instance, a system contains one or several subsystems; each subsystem contains one or several composite components, and so on. A component instance is uniquely identified in a system by a sequence $A = (a_1, ..., a_n)$ of assembly contexts representing a path in the containment tree, where a_1 is a top-level assembly context in the system (Brosig, 2014, p. 48). The function $services : C \rightarrow SET(S)$ returns a set of provided component services S of component C. A component service in DML, is a single signature of an interface of a component's role (see Figure 2.5 in Section 2.2.2). We use the set $I_s = \{x \mid \exists c \in C_{basic} : x \in instances(c) \times services(c)\}$ to refer to all provided component services of all instances of basic components (the set C_{basic} contains all basic components in a DML model).

Given a graph of $G = (V, E)$ where V is the set of all containers in the resource landscape of a DML model, and $E = \{(v, v') \in V \times V \mid contains(v, v')\}$. The function $contains : V \times V \rightarrow boolean$ returns true if the first container directly contains the second container. Furthermore, we define a helper function $resource : V \times R \rightarrow P$, where R is the set of processing resource types defined in a DML model (e.g., CPU or HDD) and P is the set of all processing resource specifications (see Figure 2.4 in Section 2.2.2). This functions searches the tree G in upwards direction starting from the specified container $v \in V$ for a processing resource specification $p \in P$ that has the specified type $r \in R$. It returns the first match. The function $deployment : A \rightarrow V$ returns the deployment of a component instance $a \in A$ on a container $v \in V$.

Transformation Rules Our transformation takes a DML model as input. The output is a workload description conforming to the meta-model depicted in Figure 5.4. Furthermore, the user may specify a configuration parameter that

determines whether the fined-grained or coarse-grained service behaviors in the DML model are used. The transformation rules are:

- *T1:* Each service instance $i \in I_s$ is mapped to a service in the workload description. To be unique, the name of this service is a concatenation of the names of all assembly contexts on the path of the corresponding component instance, the name of the interface providing role and the name of the signature itself. We represent instances of the same component by different services, since components in different assembly contexts may exhibit a different resource demanding behavior: The resource demands of a component may also depend on the workload, besides the component implementation and the system configuration. For instance, the component services may be called with different parameter values influencing the required amount of resources for servicing a request. For each service instance i, transformation rules *T3* and *T4* are called.

- *T2:* Given the set $A_{basic} = \{a \in A \mid \exists c \in C_{basic} : a \in instances(c)\}$ containing all instances of basic components, we define the set of of containers $V_{basic} = \{v \in V \mid \exists a \in A_{basic} : v = deployment(a)\}$ which are deployment targets of basic components. Then each processing resource in $\{p \in P \mid \exists v \in V_{basic} \exists r \in R : p = resource(v, r)\}$ maps to a resource in the workload description. The scheduling strategy and number of servers parameters are copied directly from the processing resource specification.

- *T3:* For a given service instance i, we determine all resource demands in the service behavior defined in the corresponding component type. For each resource demand, we determine the type $r \in R$ of the resource accessed and look up the processing resource specification $p = resource(deployment(a), r)$. We then determine the mapping for p in transformation rule *T2*, where $a \in A$ is the component instance of service instance i.

- *T4:* For a given service instance i, we determine all external calls in the service behavior defined in the corresponding component type. For each external call, we determine the target service instance $i' \in I_s$ following the assembly connections defined in the DML model. We then determine the mapping for i' in transformation rule *T1*.

Limitations In the following we describe limitations that apply to fine-grained service behaviors regarding the use of internal actions and fork actions. Internal

actions in a fine-grained service behavior represent any type of work that requires a certain time at a resource. DML allows a very fine-grained specification of these internal actions (e.g., a sequence of several internal actions accessing the same resources). In order to be able to distinguish between multiple internal actions accessing the same resources, a very fine-grained instrumentation of the application would be required providing visit counts for the individual resources. However, in practice such a fine-grained instrumentation is usually prohibitive and if possible, the direct measurement of resource demands would be feasible as well – e.g., using the techniques by Barham et al. (2004) and Brunnert et al. (2013). Therefore, we assume that all accesses to the same resource are combined in a single internal action at the top level of a fine-grained service behavior.

Fork actions in DML are used to model parallel processing (with or without synchronization). Service behavior below a fork *without* synchronization are modeled as an independent request as the behavior below the fork does not influence the response time behavior of the current request directly. As a result, we model this part as an additional workload class whose arrival rate depends on the arrival rate of the component service. Forks with synchronization and acquire/release actions (used to model software synchronization) are left for future work as they cannot be modeled with standard QNs. This could be accomplished using Layered Queueing Networks (Neilson et al., 1995) or Queueing Petri-nets (Kounev et al., 2012a).

Architecture-level performance models, such as DML and PCM, allow the modeling of dependencies on input parameters of the system. We assume that these parameter dependencies have been resolved before resource demand estimation.

5.3.3 Derivation of Estimation Problems

In the context of LibReDE, an estimation approach is defined as the combination of a strategy to derive estimation problems from a workload description and a technique (e.g., least-squares regression, or Kalman filter) to solve the estimation problems. In this section, we describe strategies to derive estimation problems. Each estimation problem is defined as a triple $E = (D, f, h)$:

- The set D defines the estimation state, i.e., the possible range of demand values. It may be any subset of $\mathbb{R}^n$. The number n is less or equal to the number of resource demands defined in the input workload description. Different partitioning schemes are possible to split a workload description into several estimation problems.

- The function $f : D \rightarrow D$ calculates the next demand vector d_{k+1} based on the current vector d_k. In most cases, this is the identity function, i.e., we assume the resource demand does not change between iterations. If knowledge about the temporal changes of resource demands is available, this can be encoded into the function f. Currently, only Kalman filter techniques can exploit this information.
- The function $h : D \rightarrow \mathbb{R}^m$ calculates the output vector z_k based on the current demand vector d_k. The derivation strategy determines which observations are included and how they are calculated.

Partitioning Schemes In order to reduce the estimation complexity, it may be helpful to partition the set of all resource demands in a workload description into disjoint subsets. Each subset then forms its own estimation problem that can be solved independently. In the following, we describe three different schemes to automatically derive estimation problems from a workload description.

- *Per-system:* The complete system is covered by a single estimation problem. Underlying statistical techniques may benefit as it allows to minimize the overall estimation error. However, the complexity of solving the estimation problem may quickly increase with increasing system size.
- *Per-tier:* In distributed systems we can often partition the resources into system tiers. We can then derive separate estimation problems for each tier. However, a partitioning into tiers may no always be possible. A workload description containing a set of resources R and a set of services S must fulfill the following condition: It exists a partition T of the set of resources R, so that if $A, B \in T$ and $A \neq B$, then $accessedBy(A) \cap accessedBy(B) = \emptyset$. The function $accessedBy : R \rightarrow SET(S)$ returns for a given resource $r \in R$ the subset of services S that have a resource demand at resource r. We typically strive for a partition T where the number of resources in each part is minimized. Thus, it is possible to reduce the complexity of solving an estimation problem. However, in case of estimation approaches based on end-to-end response times, we are now relying on the availability of residence time measurements for the individual services in the workload description in order to be able to attribute the delays to individual tiers.
- *Per-resource:* We derive separate estimation problems for each individual resource in a system. This strategy is typically used by estimation ap-

proaches that are based on the Utilization Law or the Service Demand Law. However, we usually cannot exploit the additional information provided by response time or residence time observations, as this information is not available on the level of individual resources in most practical systems.

Estimation Approaches Table 5.16 shows the estimation approaches currently covered by LibReDE. These estimation approaches are based on the existing ones described in Section 5.1.1. The table also shows the underlying statistical technique of each approach and the partitioning scheme employed. The output functions we are using are compliant to those described in the literature. The optimization approach in Liu et al. (2006) is included in two variants differing the partitioning scheme. "Direct" as statistical technique means that the resource demands can be directly calculated.

Estimation Approach	Statistical Technique	Partitioning
Approximation with Response Times (Brosig et al., 2009)	Direct	T
Service Demand Law (Brosig et al., 2009)	Direct	R
Rolia and Vetland (1995)	Non-negative least squares	R
Kraft et al. (2009)	Non-negative least squares	S
Wang et al. (2012)	Kalman filter	S
Zheng et al. (2008)	Kalman filter	S
Menascé (2008)	Non-linear, constrained optimization	T
Liu et al. (2006)	Non-linear, constrained optimization	S/T

Table 5.16: List of estimation approaches including the underlying statistical techniques and the partitioning scheme (R: per-resource, T: per-tier, S: system).

Adaptations In this step, the estimation problems may also be adapted. Possible adaptations that may be employed in this step are, for instance, the preprocessing steps described by Pacifici et al. (2008) in order to merge services with high collinearities, are combine services with low contribution to the total resource demand.

5.3.4 Cross-Validation and Approach Selection

Approach Selection As a first step, LibReDE checks all estimation problems that resulted from the derivation described in Section 5.3.3 and filters the ones for which required observation data is missing. The check comprises the different metrics of observation data required for calculating the output function (e.g.,

response time or utilization), as well as, any requirements on the amount of observation data (e.g., the linear model of least-squares regression requires a certain number of observation samples to be identifiable).

LibReDE comes with a set of rules to automatically derive time series of additional metrics from the input observations. For instance, if time series of arrivals and departures of individual requests are provided, it will automatically derive time series of the response times, the throughput and the arrival rate. Furthermore, it provides automatic conversions of the sampling interval length if the estimation requires longer sampling intervals than the input data. These derivations and conversions of the input observation data avoid additional pre-processing of the observation data before invoking LibReDE.

The results from the cross-validation is the main factor used to rank the estimation approaches according to their cross-validation result. We currently choose the approach with the highest ranking and return its estimates as result of the estimation method. Future work may extend this selection and implement ensemble estimation through dynamic weighting of the results of the different estimation approaches.

Auto-tuning The estimation approaches exhibit different parameters that control their behavior and that can have influence on the accuracy of the results. These parameters currently need to be configured manually by a user when calling LibReDE. However, it can be challenging to determine good values for these parameters. While some guidelines for parameterization are available in the literature, e.g., for Kalman filters in Zheng et al. (2008), they typically also depend on the characteristics of the system under study and the observations. Using our cross-validation scheme, an auto-tuning mechanism can be integrated to automatically search for parameter settings that provide a good accuracy for the resulting estimates. For instance, the search algorithm proposed by Noorshams et al. (2013) can be integrated here. The integration of auto-tuning mechanisms is part of ongoing work. Preliminary results are presented in Grohmann (2016a) and Grohmann et al. (2017).

Cross-Validation For a given system under test and given observation data, more than one estimation approach may be applicable after checking the preconditions. While we identified a number of factors influencing the estimation accuracy in Section 5.2, it is still unclear whether the set of influencing factors is complete. Furthermore, the identification of static decision rules on the input observation data including the quantification of threshold values would require extensive experiments on a large set of different systems. Given the

lack of sufficient experiment data, we decided to use a cross-validation scheme to automatically evaluate the accuracy of the resource demands resulting from an estimation approach.

The input observation data is split randomly into N disjoint, balanced subsets. The estimation is repeated N times, each time using a different subset for validation and the remaining subsets for training. We use the following fitness function $F(D)$ to determine the quality of the estimated resource demands:

$$F(D) = \frac{1}{K} \sum_{n=1}^{K} (\alpha \cdot \Theta(D,n) + (1-\alpha) \cdot \Phi(D,n)) \tag{5.7}$$

D is a vector containing the resource demand estimated using the training set. K is the number of samples in the validation set. $\Theta(D,n)$ is the average response time error for sample n and the resource demands D. $\Phi(D,n)$ denotes the average utilization error respectively. α (default is $\alpha = 0.5$) is a weighting factor to control the weight of response time errors with regards to utilization errors.

$$\Theta(D,n) = \sum_{c \in C^{sys}} w_c \cdot \left| \frac{R_c(D) - \tilde{R}_c^{(n)}}{\tilde{R}_c^{(n)}} \right| \quad \text{with} \quad w_c = \frac{X_c}{\sum_{l \in C^{sys}} X_l} \tag{5.8}$$

C^{sys} is the set of system-entry services. $\tilde{T}_c^{(n)}$ is the observed response time of service c and $R_c(D)$ is the calculated one using the estimated resource demands. Furthermore, we introduce a weighting factor w_c for the response time error so that errors for services that a higher throughput X_c are weighted stronger.

$$\Phi(D,n) = \frac{1}{I} \sum_{i \in I} \left| \frac{U_i(D) - \tilde{U}_i^{(n)}}{\tilde{U}_i^{(n)}} \right| \tag{5.9}$$

I is the set of all resources in the system. $\tilde{U}_i^{(n)}$ is the observed utilization of resource i and $U_i(D)$ is the calculated one using the estimated resource demands.

The workload description is required to be a product-form queueing network where the global delays can be broken down to local delays. This is the case for all BCMP networks (Baskett et al., 1975).

5.4 Concluding Remarks

In this chapter, we provided a systematization and experimental comparison of existing approaches to resource demand estimation and proposed a new

estimation method based on multiple statistical techniques. The systematization helps performance engineers to choose an estimation approach for a given system under study. It categorizes existing approaches according to their required input parameters, their provided output metrics, and their measures to improve their robustness to anomalies in the measurement data.

We evaluated the influence of different factors (sampling interval, number of samples, number of workload classes, load level, collinear workload classes, background jobs, and delayed processing) on the estimation accuracy of different estimation approaches. The results show, that response times measurements can improve the accuracy of the estimated resource demands significantly compared to a linear regression based on the Utilization Law, especially in cases with multiple workload classes. However, approaches employing response time measurements are very sensitive if assumptions of the underlying mathematical model are violated (e.g., wrong scheduling strategy, or delays due to other resources). In order to fully leverage the benefits of using response time measurements, it is therefore necessary to include knowledge on the system architecture into the resource demand estimation. This allows to derive the correct mathematical model for resource demand estimation.

We proposed a novel method to resource demand estimation that automatically derives the required information from an architecture-level model. It relies on multiple statistical techniques for improved robustness and uses a cross-validation scheme to dynamically select an estimation approach relieving a user from this decision. This simplifies the usage of resource demand estimation techniques for performance engineers. Furthermore, it is a crucial building block for our self-aware approach to resource management improving the model learning capabilities by allowing to delay the decision which estimation approach to use until system run-time.

Chapter 6

Model-based Vertical Scaling of Virtualized Applications

In this chapter, we describe two model-based controllers for *vertical scaling* of virtualized applications where resources are dynamically added to or removed from individual VMs at runtime. These controllers reason on the knowledge captured in performance models in order to improve on resource allocation decisions with regards to efficiency and SLA fulfillment compared to rule-based approaches. The performance models can be automatically created and updated using the techniques described in the previous two chapters.

Although in virtualized data centers, horizontal scaling would also be feasible by cloning and starting additional VM instances for an application, it typically takes at least minutes for the new instances to be ready. Moreover, it requires the application's capability to detect and use new instances automatically, which adds additional complexity to the application architecture (e.g., load balancers and session replication mechanisms) and may not be supported by all applications (e.g., database servers).

In most recent years, hypervisors added the capability to add (or remove) virtual resources, such as virtual CPUs, memory, or I/O devices to running VMs. This is referred to as hot-add (or hot-remove) of resources. This way, for example, one can reconfigure a VM from a 2-CPU, 8 GB memory configuration to an 8-CPU, 32 GB memory configuration without restarting it. This is referred to as vertical scaling of a VM. As a result, the vertical scaling of individual VMs is a viable alternative for virtualized applications as the configured CPU and memory capacity of a VM can be changed *quickly* and *frequently* in a hypervisor via its hot-add/-remove capability.

Challenges Many business-critical applications are subject to SLOs defined on application performance metrics (e.g., response time or throughput). To determine thresholds so that the end-to-end application SLO is fulfilled poses a major challenge due to the non-trivial relationship between the resource

allocation and the application performance. An application administrator has to take into account the following factors influencing the application performance:

- *Complex application architectures:* An application may comprise several tiers, each deployed in one or more VMs. The application latency depends on the processing in each tier and the flow of requests between tiers. Furthermore, asynchronous communication and limited software resources (e.g., thread pools or connection pools) also influence the achievable application performance.

- *Heterogeneous resource access:* The processing of application requests requires access to different types of resources (e.g., CPU, memory, or IO). The extent to which each resource contributes to the end-to-end latency may vary between different application tiers.

- *Resource contention:* Due to the shared nature of a virtualized infrastructure, the achievable performance of one application can be severely impacted by possible resource contention or interference from the co-hosted applications, a problem referred to as *noisy neighbors.*

Furthermore, too many or badly timed reconfigurations can cause degradations in the application performance. Rule-based often suffer from the following short-comings leading to significant overheads:

- *Oscillations:* Rule-based approaches are inherently prone to an oscillating behavior resulting in unnecessary reconfigurations. A system administrator needs to manually find optimal values for various parameters (e.g., frequency of checks, quiet times after reconfigurations) to reduce oscillations. However, the optimal values for these parameters are application-specific and no general guidelines can be determined.

- *Limited application elasticity:* Many applications (especially legacy ones) are unable to immediately start consuming the newly available resources (e.g., memory); in some cases, the guest OS or the application itself needs to be restarted. On the other hand, we cannot just allocate the maximum amount of resources to a VM that it may require at peak times, as this would create huge overheads in terms of scheduling overheads and larger page tables or it may lead to serious resource over-provisioning. As a result, it is important for the resource allocation system to determine the required resources of a VMs in advance, and to adapt the allocations prior to the incoming demand peaks.

In order to overcome the deficiencies of threshold-based approaches and to enable a fully automated approach to dynamically control the resource allocation of virtualized applications, performance models capturing the relationship between the resource allocation to an application and the application performance as well as demand forecasting techniques are required. The former enables predictive analyses to determine the impact of reconfigurations on the application performance, while the latter enables proactive planning of reconfigurations.

Research Questions This chapter addresses the following research questions:

- How to use performance models for short-term reasoning when adding or removing virtual CPUs to applications? We describe a layered queueing model that enables the prediction of the impact of changing the number of virtual CPUs on the application performance. We design a model-adaptive, feed-forward controller based on this model that adjusts the number of CPUs according to application-level SLOs.
- Which types of contention need to be considered when estimating resource demands in a virtualized environment? We distinguish between three types of resource demands in our layered queueing network explicitly taking contention at different layers into account. We present an approach to resource demand estimation exploiting observed scheduling statistics of the hypervisor to estimate the hypervisor contention.
- How to schedule mid- and long-term reconfigurations proactively in an autonomic manner? We present a proactive, feed-forward controller to automatically schedule expensive reconfigurations (e.g., changing the memory size of a VM) during phases of low load.
- Are time-series forecasting techniques appropriate to predict the application demand on longer time frames (e.g., complete days)? We develop a demand forecasting method that relies on multiple time series analysis techniques and incorporates additional meta-knowledge (e.g., calendar information) in the models to better capture the seasonality patterns in the workloads. We evaluate its accuracy on three real-world workload traces and compare it to other state-of-the-art time-series forecasting techniques.

Chapter Outline The remainder of the chapter is organized as follows. Section 6.1 describes the short-term controller for adapting virtual CPUs of a virtualized application in response to changes in the application workload.

Section 6.2 presents the proactive controller for mid- to long-term changes to resource allocations of applications. Section 6.3 concludes this chapter.

6.1 Short-term Vertical CPU Scaling

In this section, we propose a model-adaptive, feed-forward controller for vertical CPU scaling. The goal of the controller is to improve SLA compliance while ensuring resource efficiency. It is model-adaptive as it continuously estimates the resource demand parameters of a performance model during run-time. In Section 6.1.1, we introduce our end-to-end control loop. Section 6.1.2 contains a description of the underlying modeling approach used for reasoning on the application performance. In Section 6.1.3, we discuss how the resource demands are estimated also capturing contention effects at the physical hardware layer. Finally, Section 6.1.4 presents the resource control algorithm for vertical CPU scaling. This section if based on our paper in Spinner et al. (2014b).

6.1.1 Model-Adaptive Control Loop

A virtualized application may comprise multiple VMs running on one or more physical hosts. Each VM may host different parts of an application (e.g., application server or database). We assume that the application owner is able to provide a tuple $\langle metric, target \rangle$ for each application specifying the SLO, where $metric$ (denoted as p) defines the application performance metric to be managed (e.g., end-to-end response time, or throughput), and $target$ (denoted as p_{ref}) contains the desired value of the corresponding metric. We assume that the user-specified performance $metric$ can be monitored at runtime without significant overheads.

Our approach is based on a prediction model $p = f(\lambda, \mathbf{a})$ describing the relationship between the application performance p and the current workload λ and resource allocation vector $\mathbf{a}$. The structure and parameterization of the performance model depends on the application architecture. We rely on the reference architecture for model learning described in Chapter 4 to provide us with the DML instance reflecting the current system architecture. Our prediction model $p = f(\lambda, \mathbf{a})$ is automatically derived from the DML instance. A coarse-grained description of the application behavior is sufficient as we only require the explicit representation of quickly changing factors (e.g., arrival rates, or scheduling delays at the hypervisor) in our model. Other factors are assumed to only change slowly around the current operating point over time (over hours and days) and are implicitly integrated in the model parameters.

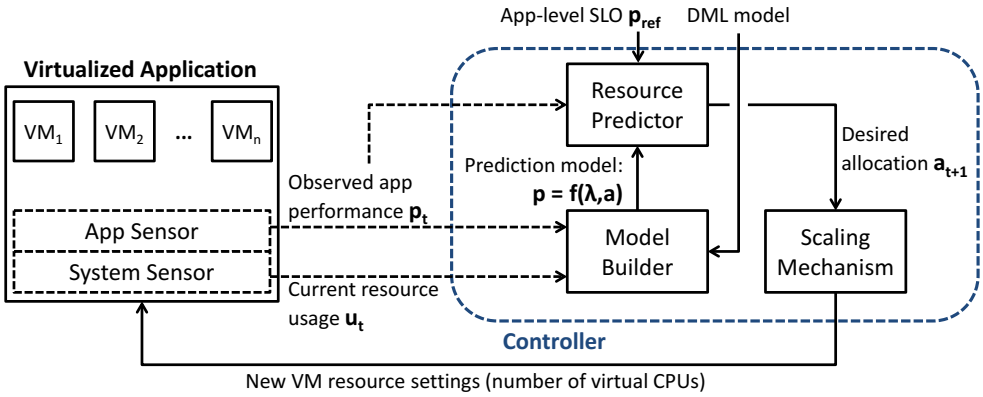


Figure 6.1: Overview of the model-adaptive control loop

In order to capture changes in these factors, we frequently update the model parameters based on real-time monitoring data.

Figure 6.1 gives an overview of the model-adaptive control loop. Our control loop consists of two major steps – the *model builder* and the *resource predictor*. The model builder derives a layered prediction model from the input DML model and determines its current resource demands using LibReDE (see Section 5.3). It receives the current application performance statistics (average response time, throughput, and if available the queue length of application requests) from the *application sensors* and resource usage statistics of all VMs of a virtualized application from the *system sensor*. We expect the system sensors to also provide detailed scheduling statistics for individual VMs. Most modern hypervisors keep track of scheduling statistics, such as wait times due to contention. For instance, the VMware ESX hypervisor collects fine-grained scheduling statistics in the *vCenter PerformanceManager* component (VMware, Inc., 2013).

The resource predictor uses the model to predict the VM-level resource allocation vector $\mathbf{a}_{t+1}$ that will be required in the next control interval to fulfill the user-specified performance target p_{ref} at the application level. The *scaling mechanism* translates the desired allocation to hypervisor resource settings and uses hot-add of virtual CPUs to dynamically applied the new settings to the appropriate VMs. The model-adaptive control loop is executed at regular time intervals – the *control interval* – which typically lies in the range of seconds to a few minutes, so that the controller can react quickly to system or workload changes.

Our approach considers both, the scale up of VMs in response to an increasing workload intensity as well as the scale down during phases of low usage.

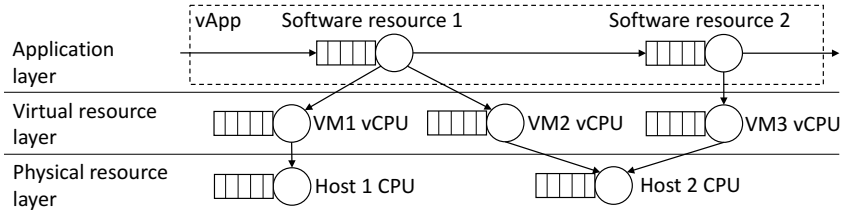


Figure 6.2: Overview of layered queueing model.

While the hypervisor is able to reschedule resources not used by one VM to another, a large VM (i.e., with a lot of configured resources) still causes additional overheads resulting in inefficient usage of physical resources. For instance, the VMware vSphere hypervisor implements a co-scheduling (a.k.a. gang-scheduling) policy for all CPUs of one VM. This can result in additional scheduling delays if a VM is assigned several CPUs. The resource predictor also determines when the number of virtual CPUs can be reduced for a VM.

In the following, we assume that the total available physical resources of a host are sufficient to fulfill the SLOs of all hosted applications. We consider the problem of performance isolation between applications co-hosted on an over-committed host (i.e., more virtual resources are configured for VMs than there are physical ones available) orthogonal to our problem. Approaches to the global scheduling or resources in a virtualized data center (see Chapter 3.1.3) may be combined with our approach in future work in order to balance the load on a global level in a data center.

6.1.2 Modeling Approach

Given a DML model that describes the system architecture, we automatically derive a layered prediction model $p = f(\lambda, \mathbf{a})$. The prediction model is used to determine *a*) whether the performance target can be fulfilled under the current workload λ and the current resource allocation $\mathbf{a}$, and *b*) which resource is currently the performance bottleneck. In order to answer these questions we represent the system with a queueing model. Due to the complexity of virtualized environments, we adopt a layered modeling approach for describing the application performance, where a virtualized system consists of a layered architecture, with each layer contributing to the externally visible application performance. We distinguish between the following three layers, as shown in Figure 6.2:

- The *physical resource layer* consists of the hardware resources (CPUs, main memory, etc.) of one or more physical hosts.

- The *virtual resource layer* consists of the virtual resources (number of CPUs, size of main memory) which are configured for each VM. The hypervisor dynamically schedules the virtual resources on the physical ones allowing for sharing of physical resources between VMs.

- The *application layer* captures the performance behavior of the application, including software resources (e.g., caches or thread pools) and the control flow between different VMs.

Each layer introduces additional sources of contention, which may slow down the processing of application requests. In today's hypervisors, the physical resources are not dedicated, i.e. the hypervisor dynamically schedules resources to a VM depending on its current demand. In order to increase consolidation ratios and improve resource efficiency, it is possible to *over-commit* the physical resources of a host. That means, the sum of the configured virtual resources for all the VMs can exceed the capacity of a physical resource of the host. In over-committed scenarios, the different VMs may contend for the same resources at the physical resource layer forcing the hypervisor to time-division scheduling. At the virtual resource layer, different processes may request processing time at resources resulting in delays due to guest operating system scheduling. At the application level, software resources (e.g., thread or connection pools) can lead to software contention limiting the possible application throughput. These different levels contribute to the complex relationship between resource allocation and application performance.

In order to address the complexity of the layered architecture of virtualized systems, we adopted a modeling approach based on LQNs (see Section 2.4.2) and the Method of Layers (MOL) (Menascé, 2002; Rolia and Sevcik, 1995). MOL is an extension to traditional queueing networks enabling hierarchical modeling. The service time of a queue at level l is equal to the response time of an underlying closed queueing network at level $l-1$, i.e., the service times at higher layers include delays due to contention in the lower layers. In the following, we describe the modeling of CPU resources in a virtualized environment.

Application Layer Each software resource limits the number of requests which can be processed concurrently in a subpart of an application. LQNs support multiple layers of software resources. For reasons of simplicity, we limit the following description to one layer of software resources, however, it can be

generalized to multiple layers. The end-to-end response time T_c of workload class c at the system is:

$$T_c = \sum_{l}^{L} R_{l,c} \tag{6.1}$$

The residence time $R_{l,c}$ of workload class c at software resource l consists of the waiting time $W_{l,c}$, if a request needs to wait for a free server and the service demand $D_{l,c}^{app}$ for processing the request. In the following, this service demand is called *application demand* in order to distinguish it from the service demands on the virtual and physical resource layers. The scheduling at software resource l depends on the application implementation. Typical scheduling strategies on the application level are IS (i.e., no contention due to software resources), and FCFS. Assuming a $M/M/m_l$ queue with m_l corresponding to the maximum concurrency level of software resource l, the residence time $R_{l,c}$ can be calculated analytically using the following equations:

$$R_{l,c} = \begin{cases} D_{l,c}^{app} & \text{if IS} \\ D_{l,c}^{app}(1 + \frac{Q_l}{m_l} PB_l) & \text{if FCFS (assuming } |C| = 1) \end{cases} \tag{6.2}$$

Q_l is the mean queue length seen on arrival of a new request at software resource l (excluding the requests currently being processed). PB_l is the probability that a newly arriving request will find all servers busy. PB_l can be computed using the Erlang C formula (see Equation 2.21 on page 36). It is important to note that Equation 6.2 is valid for FCFS only in cases with a single workload class. See Equation 2.24 on page 36 for an approximation for multiple classes. Additional scheduling strategies may be included here, for instance, as described by Rolia and Sevcik (1995).

Virtual Resource Layer The processing at this layer is described by a closed QN where each virtual resource (e.g., CPU or I/O resource) is represented by a queueing station. The utilization U_l of software resource l determines the think times in the closed QN. In general, the closed QN could describe the fine-grained application control flow including accesses to different virtual resources (CPU, hard disk, network). In order to avoid the intrinsic complexity of explicitly modeling all virtual resources (including the control flow between them), we choose a coarse-grained modeling approach, focusing on the CPU behavior. The performance influence of other resources is captured by a generic delay resource with IS scheduling. Then the application demand $D_{l,c}^{app}$ is:

$$D_{l,c}^{app} = \sum_{v}^{A_l} (W_{v,l,c}^{os} + D_{v,l,c}^{virt}) + D_{l,c}^{other}. \tag{6.3}$$

A_l is the set of virtual CPUs accessed by software resource l. $D^{virt}_{v,l,c}$ is the service demand of requests of workload class c arriving through software resource l at the virtual CPU v, ie., the CPU time at operating system-level required to process one request. In the following we call it the *virtual resource demand*. $W^{os}_{v,l,c}$ is the waiting time of a request due to OS scheduling when accessing a CPU. The OS scheduling of processes to CPUs can be typically described using a PS strategy, for which well-known analytical solutions for the waiting time $W^{os}_{v,l,c}$ exist. $D^{other}_{l,c}$ is the time a request spends at other resources in the system. As described in Section 6.1.3, we continuously update this value based on monitoring data to reflect changes in $D^{other}_{l,c}$.

Physical Resource Layer The hypervisor schedules virtual CPUs on physical ones introducing additional waiting times if no physical resources are available. Each physical CPU is represented by a closed QN consisting of a single queueing station. The think times depend on the utilizations U_v of each virtual CPU. Therefore, the virtual resource demand $D^{virt}_{v,l,c}$ is defined as:

$$D^{virt}_{v,l,c} = W^{hyp}_{v,l,c} + D^{phys}_{v,l,c} = C_v \cdot D^{phys}_{v,l,c}. \tag{6.4}$$

$D^{phys}_{v,l,c}$ is the service demand of requests of workload class c arriving through software resource l at the physical CPU excluding any contention effects due to hypervisor or operating system scheduling. We call this service demand the *physical resource demand*. $W^{hyp}_{v,l,c}$ is the waiting time of requests due to hypervisor scheduling.

Given that the hypervisor is not aware of different workload classes when scheduling virtual CPUs on physical ones, we argue that the processing of requests of a VM is equally slowed down across workload classes. Therefore, we introduce the contention factor C_v describing the relative slow-down of VM v due to hypervisor scheduling if several VMs are contending for the same CPU. The contention factor is defined as:

$$C_v = \frac{W^{hyp}_{v,l,c}}{D^{phys}_{v,l,c}} + 1 \quad \forall c \in C, l \in L. \tag{6.5}$$

However, the contention factor may differ between VMs running on the same host. For instance in the VMware ESX hypervisor, there are two main reasons for scheduling delays at the physical resource layer:

- *Over-commitment:* When several VMs require CPU resources at the same time in an over-committed scenario, the hypervisor may be forced to

put some of the VMs in a *ready* state where they wait until a physical CPU becomes available. Assuming that all VMs have exactly one virtual CPU and equal scheduling priorities, a PS scheduling strategy is able to describe the hypervisor scheduling. In cases where VMs have different priorities, support for a GPS scheduling strategy (Parekh and Gallager, 1994) is required. Current solvers for LQNs (Franks et al., 2009; Pérez and Casale, 2013a; Waizmann and Tribastone, 2016) lack support for this type of scheduling.

- *Co-scheduling:* Furthermore, if a VM has two or more virtual CPUs, the hypervisor will try to schedule them at roughly the same time in order to ensure that the CPU time of each virtual CPU of the same VM progresses simultaneously. If there are not sufficient free physical CPUs to schedule all virtual CPUs, the hypervisor may put the VM in a *costop* state until enough physical resources are available at the same time. This behavior is known as co-scheduling (Ousterhout, 1982).

While a VM is in the ready or costop state the application processing is delayed. The delays due to scheduling at the hypervisor layer result in a variable service rate of the virtual CPU that not only depends on the current application workload of this VM, but the workloads of all other VMs on the same host.

6.1.3 Model Estimation

The parameters of the layered performance model are estimated based on monitoring data provided by the application and the hypervisor. The following parameters are tracked continuously: the *application demand*, the *virtual resource demand*, and the *physical resource demand*. In the following, we describe our approach to estimate these parameters.

The estimation of the different demands is based on existing techniques for resource demand estimation (see Chaper 5.1 for an overview of such techniques). However, these techniques do not take the different layers of a virtualized system into account. For instance, the techniques described in (Kraft et al., 2009; Liu et al., 2006; Menascé, 2008; Zheng et al., 2005) are using end-to-end application response time observations. However, as the response time also includes delays due to contention effects at the hypervisor level, the dynamically changing workload at the hypervisor layer may result in time-varying resource demands.

The demand estimates are updated in a regular interval, called the *estimation interval*. At the end of each estimation interval, new readings from the application and system sensors are obtained, and the demand estimates are

updated accordingly. For demand estimation only the last M measurements are considered, so that the model estimator can adapt to changes in the system configuration and application behavior. The time period consisting of M measurements is called the *estimation window*. For instance, in our experiments in Section 7.5 we use an estimation interval of five minutes and an estimation window of one hour.

Application Demand The application demand is estimated based on observations of the system response time T_c and the average queue length Q_l seen by a request on arrival at software resource l. Q_l can either be observed directly if the application provides these statistics or derived using queueing theory (Menascé et al., 2004) (e.g., from the observed utilization of the software resource). In the latter case, it is important to ensure that the system is stable during one estimation interval, i.e., $\lambda \approx X$. This assumption holds for many interactive systems (e.g., Web servers), if the response times are small compared to the estimation interval and a large number of requests are processed in each estimation interval. If the system is not stable (e.g., in case of batch processing), we rely on direct observations of Q_l. Using Equations 6.1 and 6.2, we can estimate the application demand $D_{l,c}^{app}$ based on the queue-length and response time observations using non-negative least squares regression (as described in Section 5.1.1.3).

Virtual Resource Demand In order to estimate the virtual resource demand, we rely on scheduling statistics reported by the hypervisor. VMware ESX provides the c_v^{ready} and c_v^{costop} performance counters for each VM, reporting the total time VM v is in a wait state due to CPU contention from other VMs or due to co-scheduling. This allows us to estimate C_v, the slow down due to the VM being in a wait state:

$$C_v = \frac{c_v^{ready} + c_v^{costop}}{N}. \tag{6.6}$$

N is the length of the estimation interval. By estimating the contention factor C_v for each estimation interval and combining it with the physical resource demand $D_{v,l,c}^{phys}$, we obtain the virtual resource demand $D_{v,l,c}^{virt}$.

Physical Resource Demand Given the estimates of the application demand $D_{l,c}^{app}$ and the contention factor C_v, we can estimate the physical resource demand

$D_{v,l,c}^{phys}$ based on Equation 6.3 and – assuming PS scheduling – Equation 2.20:

$$D_{l,c}^{app} = \sum_{v}^{A_l} C_v \cdot D_{v,l,c}^{phys} \left(1 + \frac{1}{m_v} \cdot \frac{PB_v}{1 - U_v}\right) + D_{l,c}^{other} \tag{6.7}$$

In the above equation, m_v is the number of virtual CPUs and PB_v is the probability of a request having to wait at the virtual CPU v. PB_v can be calculated using the Erlang-C equation (see Equation 2.21 on page 36). The utilization U_v is determined using the scheduling statistics from the hypervisor as following: $U_v = \frac{1}{m_v} \cdot (C_v + \frac{c_v^{run}}{N})$ where c_v^{run} is the total CPU execution time c_v^{run} of a VM v as reported by the hypervisor, and N is the length of the estimation interval. Equation 6.7 can then be solved using a Kalman filter or a generic optimization algorithm to determine $D_{v,l,c}^{phys}$. In the special case of a single workload class, the physical resource demand $D_{v,l,c}^{phys}$ may be estimated directly based on observations of the average application throughput X_c and the total CPU execution time c_v^{run} of a VM v using the Service Demand Law.

Demand Variations Changing the number of virtual CPUs can have a profound impact on the application performance behavior. Therefore, we expect the application demand $D_{l,c}^{app}$ to change depending on the number of CPUs m_l. For instance, when adding CPUs to a VM, application thread pools might become a limiting factor due to the increased parallelism resulting in an additional slowdown of the application. Therefore, the application demand $D_{l,c}^{app}$ (and also $D_{l,c}^{other}$) is learned in relation to the number of CPUs m_l. After a reconfiguration, it is therefore important that the estimation of $D_{l,c}^{app}$ quickly adapts to the new values. In our prototype for the case study in Section 7.5, we discard all previous observations in case of a reconfiguration and start from scratch with the resource demand estimation. Future work may improve this behavior and investigate whether the usage of a Kalman filter may help to shorten the transition periods.

6.1.4 Resource Control

In this section, we show how the prediction model described in Section 6.1.2 is used to dynamically scale the number of CPUs of a virtualized application. We first describe the resource control algorithm used for reasoning. Then we discuss how the model can help to detect bottlenecks in different system layers. Finally, we show how the reconfiguration can be implemented in a hypervisor.

6.1.4.1 Reasoning Algorithm

At the beginning of each control interval, the resource controller analyses our layered performance model along with the measured application performance to determine whether VMs need to be scaled up or down, either to mitigate SLO violations or to improve resource efficiency. The current approach is focused on adding and removing virtual CPUs from individual VMs during system runtime. Changing the number of CPUs of a VM is a relatively cheap reconfiguration given that modern operating systems support hot-plugging of CPU resources without the need to reboot the guest operating system. Thus applications can directly benefit from the additional computing power.

Algorithm Overview Our resource control algorithm is a hill-climbing optimization algorithm executed for each virtualized application at the beginning of a control interval. Algorithm 6.1 shows the steps of the algorithm. The algorithm expects a fully-parameterized performance model of the application as input. Additionally, it requires the target response time T_c^{ref} for each workload class c as provided by the system administrator, the current arrival rate λ_c of requests of workload class c at the application level, the current queue length q_l at software resource l, and an allocation vector $\mathbf{a} = (a_1, ..., a_n)$ containing the current number of virtual CPUs for each VM of an application. It returns the desired CPU allocation vector $\mathbf{a}^{next}$ for the next control interval, which is then applied to the system.

The algorithm answers *(a)* whether a reconfiguration is needed to ensure the application performance target or improve the resource efficiency, and *(b)* which VM should be reconfigured. The first part of the algorithm evaluates if the application can still fulfill its performance targets if a CPU is removed from any of the member VMs and chooses the VM which has the least impact on the application performance (lines 3-11). The second part of the algorithm is executed if the application performance targets are or will soon be violated, and determines which VM is best scaled up to improve the application performance (lines 12-19).

Model Analysis The algorithm uses the `AnalyseModel` helper function to predict the expected application performance for a given number of CPUs assigned to a VM. Existing model solution techniques for LQNs may be used here, such as the Method of Layers (MOL) (Rolia and Sevcik, 1995) or the fluid limits solver LINE (Pérez and Casale, 2013b). In Müller et al. (2016), we evaluate different solution techniques for LQNs with regards to their accuracy and speed. Both LINE and MOL can provide sufficiently fast solutions within clearly under a

Algorithm 6.1: Resource control algorithm

Input: target latencies $(T_1^{ref}, ..., T_{|C|}^{ref})$, arrival rates $\boldsymbol{\lambda} = (\lambda_1, ...\lambda_{|C|})$, current queue lengths $\mathbf{q} = (q_1, ..., q_{|L|})$, current number of CPUs for each VM $\mathbf{a} = (a_1, ..., a_n)$

Output: desired number of CPUs $\mathbf{a}^{next} = (a_1^{next}, ..., a_n^{next})$

$\mathbf{a}^{next} = \mathbf{a}$

$T_c^{cur}, X_c^{cur} \leftarrow$ `AnalyseModel(`$\boldsymbol{\lambda}$, $\mathbf{q}$, $\mathbf{a}$`)` $\forall c \in C$

for $v = 1$ **to** n **do**

  $T_{c,v}^{down}, X_{c,v}^{down} \leftarrow$ `AnalyseModel(`$\boldsymbol{\lambda}$, $\mathbf{q}$, $\mathbf{a} - \mathbf{e}_v$`)` $\forall c \in C$

  $u_v \leftarrow 0$

  if $\forall c \in C : T_{c,v}^{down} < \delta \cdot T_c^{ref}$ **then**

    $u_v \leftarrow \sum_{c=1}^{C} (T_c^{ref} - T_{c,v}^{down})$

if $\exists v \in \mathbb{N} \mid 0 < v \leq n : u_v \neq 0$ **then**

  $d \leftarrow \underset{v}{\operatorname{argmax}}(u_v)$

  if $\forall c \in C : \lambda_c \leq X_{c,d}^{down}$ **then**

    $a_d^{next} \leftarrow a_d - 1$

else

  if $\exists c \in C : T_c^{cur} > T_c^{ref}$ **then**

    for $v = 1$ **to** n **do**

      $T_{c,v}^{up}, X_{c,v}^{up} \leftarrow$ `AnalyseModel(`$\boldsymbol{\lambda}$, $\mathbf{q}$, $\mathbf{a} + \mathbf{e}_v$`)` $\forall c \in C$

      $s_{c,v} \leftarrow \frac{X_{c,v}^{up}}{X_{c,v}^{cur}} \forall c \in C$

    $d \leftarrow \underset{v}{\operatorname{argmax}}(\sum_{c \in C} s_{c,v})$

    if $\exists c \in C : s_{c,d} > s_{min}$ **then**

      $a_d^{next} \leftarrow a_d + 1$

second for models of similar complexity. Simulation techniques are typically not appropriate given that the resource control algorithm is typically executed in relatively short intervals (e.g., every 20 seconds).

In the case study described in Section 7.5, we used an approximation instead of a full model solution. Given the FCFS scheduling at the application level, we can assume in the case study that the number of requests in service is equal to or below the number of CPUs and hence the waiting time $W^{os}_{v,l,c}$ in Equation 6.3 is always zero. Furthermore, we assume that the contention factor C_v will always be the same in the next control interval. Evidently, this is only an approximation as the reconfiguration of CPUs can have significant impacts on the co-stop time c_v^{costop}. With this approximation, we can calculate the end-to-end response time directly without performing a full MVA.

Besides the current arrival rate $\boldsymbol{\lambda}$ and the resource allocation $\mathbf{a}$, the function `AnalyseModel` also requires the current queue length in the system $\mathbf{q}$. With short control intervals in the range of seconds or minutes, subsequent control intervals may be correlated as queue lengths build up over several intervals. Therefore, the model analysis needs to include the current state of the system into the analysis of the model. However, we only consider the state of the queues in the application layer here assuming the processing of request in the virtual and physical resource layers finish within a single control interval.

Scale Down In order to determine possible candidate VMs for scale down, the algorithm calculates the expected end-to-end response time $T^{down}_{c,v}$ if one CPU is removed from VM v (line 4). $\mathbf{e}_v$ is the unit vector of dimension v. If the predicted $T^{down}_{c,v}$ is less than $\delta \cdot T^{ref}_c$ for all workload classes $c \in C$ (line 6), we calculate the distance between the predicted response time and its target (line 7). The result is stored in variable u_v. The factor δ is a configurable parameter that controls the aggressiveness with which the controller scales down VMs. In our experiments, we typically used a value of $\delta = 0.75\%$.

The VM for which removing one CPU maximizes the distance u_v is selected for scale down (lines 8 and 9). Before scale down, an additional check is performed to ensure that the system as a whole is still stable afterwards (lines 10 and 11). The stability check avoids unnecessary oscillations, because the queue size would increase again after scale down. This check is required for non-interactive, job-based systems, where it is often acceptable queue up work for a short time as long as the SLOs are not violated.

Scale Up If there is no potential for a scale-down identified, the controller checks whether a scale-up is required. It compares the expected end-to-end

response time T_c^{cur} with the target application latency T_c^{ref} (line 13). If any SLOs are violated for any of the workload classes, we search for the VM v that provides the maximum overall speedup $s_{c,v}$ when adding one CPU (lines 14 - 17). The speedup $s_{c,v}$ is the ratio between the predicted throughput $X_{c,v}^{up}$ with one additional CPU and the current throughput $X_{c,v}^{cur}$. If the expected speedup is above a minimum value s_{min}, a CPU is added. The value s_{min} is configurable and controls when to stop adding CPUs if the expected speedup is minimal.

In each control interval, the algorithm adds or removes only one CPU at a time. This helps the model estimator to learn the application demands D_k^{app} gradually by exploring the reconfiguration space. In our experiments, the control interval length was set to 20 seconds. With such a short control period, the controller can also react to fast workload changes adequately over several control periods.

6.1.4.2 Bottleneck Analysis

While the performance model and the resource control algorithm described previously are focused on capturing the impact of changing the number of virtual CPUs on the application performance, its layered structure can also be useful to detect non-CPU bottlenecks during runtime and trigger additional reconfigurations. This is especially important if the resource control reaches a point where adding virtual CPUs will not improve the application performance further. There are different reasons why an application may not benefit from additional CPUs. In the following, we will discuss possible situations and describe when additional reconfigurations may be necessary.

The VMware ESX hypervisor implements a co-scheduling scheme for virtual CPUs, i.e., all CPUs of the same VM are scheduled roughly at the same time. With increasing number of CPUs, the probability increases that a VM has to wait because there are less idle physical resources than the VM has CPUs. Given the current physical demand D^{phys} and the virtual demand D^{virt}, we have an estimate of the time a request is delayed due to hypervisor CPU scheduling. The proportion $\omega_{virt} = \frac{W^{hyp}}{D^{app}}$ can be used as an indicator for excessive CPU contention on a host. If this value reaches a certain threshold (e.g., 30% of the application processing time is due to scheduling delays at the hypervisor level), mitigation actions to reduce the contention on the host should be taken. See (Lu et al., 2014) on an orthogonal approach optimizing scheduler settings enabling SLO differentiation of virtualized applications. If the physical resources of a host are insufficient to serve the needs of all VMs, it is possible to relocate the VM to a less-utilized host using VM live-migration facilities of the hypervisor (see Gulati et al., 2012).

The proportion $\omega_{app} = \frac{D^{other}}{D^{app}}$ can be used as an indicator for bottlenecks in other hardware or software resources (e.g., main memory or I/O) not explicitly captured in the performance model. In order to pinpoint the bottleneck more precisely, more monitoring data about the current state of the application or hardware might be required. Such metrics may be available as the Zimbra case study in Section 7.5 shows. However, the capabilities to solve these bottlenecks during system runtime without service interruption may be limited.

6.1.4.3 Reconfiguration

We have implemented the resource control algorithm for the VMware ESX hypervisor. This hypervisor currently supports CPU hot-plug, i.e., adding more virtual CPUs to a VM without service interruptions. However, hot-remove of CPUs is currently not supported by most guest operating systems. It is necessary to reboot a VM to reduce the number of virtual CPUs. In order to avoid this limitation, we use CPU hot-plug mechanisms included in some guest operating systems (e.g., the sysfs kernel interface in Linux). We use these mechanisms to deactivate individual cores in the operating system to simulate the influence on the application performance. Given that the hypervisor is not aware of the deactivated cores, these cores may cause additional scheduling overheads in the hypervisor.

After a reconfiguration takes place, the model estimator is suspended for a short period of time, because some requests observed directly after the reconfiguration may be enqueued or currently in service during the scale-up/-down. The response times of these requests are only insufficiently represented in Equation 6.3 as those requests experience two different service rates. In order to prevent these observations from influencing the model estimator negatively, we skip all control intervals where the observed average reponse time T indicates that the requests arrived in the system before the reconfiguration at time t_{last} (i.e., $t_{cur} - T < t_{last}$).

6.2 Mid- and Long-term Vertical Scaling

The reconfiguration of resource allocations is not always instantaneous limiting the elasticity of applications. In such situations, reconfigurations need to be planned proactively on a mid- to long-term scale (i.e., hours or days) in order to be able to react in time before a workload peak. Otherwise, application SLOs regarding performance and availability may be violated. In this section, we show how time-series forecasting techniques can be integrated into an autonomic

controller to proactively plan the vertical scaling of virtualized applications. In the following, we use memory as an example for a resource where reconfigurations can be costly. However, the approach is independent of the type of resource. Section 6.2.1 discusses factors limiting the application elasticity using memory as an example. In Section 6.2.2, we describe our proactive control loop and propose an improved forecaster to address workloads with multiple overlapping seasonal patterns. In Section 6.2.3, we evaluate whether our forecaster is able to provide long-term forecasts with sufficient accuracy for our resource controller. This section is based on our paper in Spinner et al. (2015b).

6.2.1 Factors Limiting Application Elasticity

Modern hypervisors (e.g., VMware ESX) support two different approaches to dynamically adapt the actual amount of memory available to a virtualized application: memory hot-add (and hot-remove), or memory ballooning. Memory ballooning techniques (Waldspurger, 2002) allow to reclaim memory from the guest OS at run-time. This mechanism is used to reallocate the physical memory between VMs in over-commited scenarios. However, memory-ballooning only works within the bounds of the initially configured memory size. While it would be possible to set this size to the maximum physical memory size, it is not recommended as it usually results in memory overheads. Furthermore, it can also severely limit the possibilities to migrate such a VM between physical hosts in a heterogeneous data-center. In the following, we describe the challenges of memory scaling caused by deficiencies of many modern operating systems and applications.

Most current versions of the Linux and Windows OS can dynamically activate additional memory without a restart. However, memory reconfigurations can fail at the OS level. For instance, Linux requires a contiguous physical memory space for its internal memory management tables. If there is a high memory pressure by the application, we observed frequent failures when activating the additional memory as the OS is unable to find enough space to enlarge its memory management tables.

Many enterprise applications, including their underlying middleware and database systems, implement their custom memory management mechanisms. Examples include database systems, such as the MySQL server, which maintains a buffer cache to keep frequently used pages in memory, and process VMs, such as the Java VM, with their garbage collected heap space. These applications also need to be made aware of any additional memory added at runtime so that they can adapt their behavior accordingly. However, the parameters controlling the application's memory management mechanisms

often cannot be changed without restarting the application (e.g., the MySQL buffer pool size, or the maximum heap size of a Java application).

Salomie et al. (2013) propose an extension to MySQL and OpenJDK to integrate memory ballooning techniques in the application memory management. However, this approach works only within certain bounds configured initially and it requires profound changes in the OS and application source code (Salomie et al., 2013). As long as the application does not allow to dynamically change its memory management configuration, the only option is to restart the application although it is an expensive operation, both in terms of availability and performance. In particular, data-intensive applications need to reload their working set data back into memory after a restart. The Zimbra Collaboration Server[1], which is an example for such a data-intensive application, becomes unavailable for up to 10 minutes due to the restart and shows a severely degraded application performance (a slowdown factor of 10) for over half an hour while reloading its internal caches (see Section 7.5.5).

In summary, the major challenges when reconfiguring memory of virtualized applications are the following: a) the settings of the memory management mechanisms of applications must be adjusted to reflect the additional capacity, b) many practical applications need to be restarted to update the memory management configuration, and c) the reconfiguration is unreliable when memory demand is high and may cause additional overheads.

6.2.2 Proactive Control Loop

Our approach is based on a control loop which proactively adds or removes memory resources to VMs to match their future workload demand and to improve application availability and performance. We use a proactive approach, as it enables us to plan the reconfiguration in advance and schedule it to be executed during a phase of low application load (e.g., at night). This has the following benefits: a) reconfiguration failures at the OS level are avoided, and b) if an application restart is required, the impact on the performance and availability can be significantly reduced (see Section 7.5.5).

We assume that the application is subject to a dynamic, time-varying workload with different seasonal patterns, such as daily and weekly patterns, as well as different long-term trends (increasing or decreasing). If an application restart is required, the memory reconfiguration may take place during pre-defined maintenance windows when a short application outage is acceptable (e.g., between 3:00 and 3:15 AM). This results in significantly longer control periods

[1] `http://www.zimbra.com`

compared to many other runtime management systems. While the long control period limits the elasticity of the system, it helps to avoid counter-productive reconfigurations at peak workload.

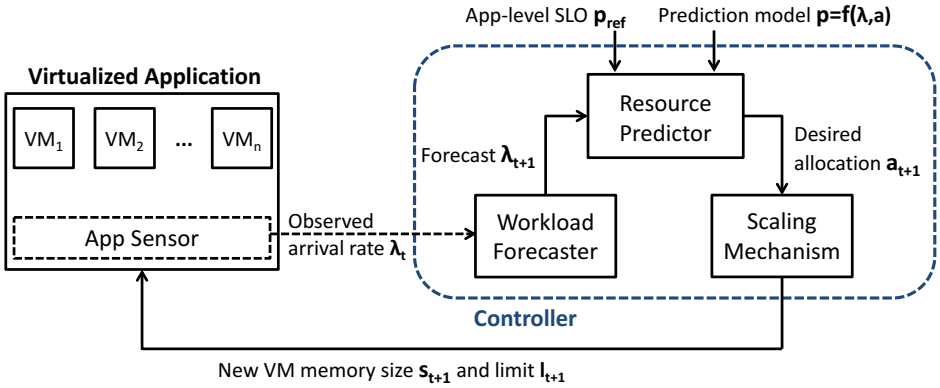


Figure 6.3: Overview of proactive control loop.

The control loop consists of the components shown in Figure 6.3. For each virtualized application, our approach periodically adjusts the memory allocations to the VMs using a closed-loop controller. As described in Section 6.2.1, a memory reconfiguration may also require changing the memory settings of the application, including a possible restart of the application. Our approach therefore needs to determine a memory allocation a_{t+1} sufficient to serve the peak workload during the next control interval $t+1$ (e.g., next 24 hours) and then it reconfigures the system in advance during a maintenance window. The *App Sensor* continuously monitors the arriving workload and stores the aggregate observations in a time series $\boldsymbol{\lambda}_t = \{\lambda_t^1 ... \lambda_t^m\}$. The parameter m controls the smoothing of the input data and also the computational overhead of the approach. Assuming a control interval of one day, we usually use $m = 48$ or $m = 24$ corresponding to a sampling interval of 30 to 60 minutes.

In order to predict the workload for the next interval, the time series $\boldsymbol{\lambda}_t$ is fed as input to the *Workload Forecaster* component. This component builds a statistical model based on the historical data $\boldsymbol{\lambda}_1 ... \boldsymbol{\lambda}_t$. Our workload forecaster relies on multiple time series analysis techniques and exploits additional calendar information to distinguish between different types of seasonal patterns (e.g., work days vs. non-working days). We refer to this approach as *splitTs* in the rest of this section. The result of the workload forecaster is the expected peak workload λ_{t+1} of the control interval $t+1$.

The *Resource Predictor* component determines the required memory alloca-

tion a_{t+1} to a VM to be sufficient for the peak workload λ_{t+1}. To this end, it requires a function $p = f(\lambda, a)$ that maps a given workload λ and a memory requirement a to the expected application performance p. In our case study in Section 7.5.5, we used a fixed step function that we determined in an offline setting using experiments with a discrete set of memory sizes (e.g., in 2 GB steps) by determining the corresponding sustainable application workload. However, more generic functions can be used depending on the application. The function may be either provided to the resource predictor as input knowledge or learned over time by analyzing historic application performance data.

Based on the required memory allocation a_{t+1}, the *Scaling Mechanism* component determines the new VM memory settings for the application. This includes the configured memory size s_{t+1} and the memory limit l_{t+1} of the VM. The limit setting is used to reduce the memory consumption of a VM using memory-ballooning techniques. This is because hot-removal of memory is currently not supported by guest operating systems without a restart of the VM. Therefore, the scaling mechanism uses the limit setting to scale down the memory of a VM if it does not require all configured memory. The freed memory can then be used by other co-located VMs enabling a higher consolidation ratio on the physical host.

6.2.2.1 Workload Forecaster

Based on the observed arrival rates $\boldsymbol{\lambda}_t$ the workload forecaster predicts the expected arriving workload during the next control interval. After observing several days (typically three complete days), it is possible to forecast the arriving workload for a complete day using time series analysis methods as described in (Hyndman and Khandakar, 2008). To avoid under-/over-provisioning of memory resources, it is crucial to predict the peak workload of the next day accurately.

A shortcoming of all forecasting methods based on time series analysis is their limited capability to identify and cope with multiple overlapping seasonal patterns at the same time. The effects of days, weeks and months are examples of such overlapping patterns as commonly found in real-world workload traces. Many workload traces from public webservers (e.g., in (Arlitt, 2000; Urdaneta et al., 2009)), or enterprise systems (e.g.,in (Herbst et al., 2014)) show regular differences in the workload intensities between different week days (e.g., working vs. non-working days).

In order to cope with these types of overlapping seasonal patterns, splitTs classifies the observed, historic data $\boldsymbol{\lambda}_1 ... \boldsymbol{\lambda}_t$ into subsets $S_d = \{\boldsymbol{\lambda_i} : f(\boldsymbol{\lambda_i}) = d, 1 \leq i \leq t\}$ with $d \in D$. The classificator f is based on calendar information

provided as static meta-knowledge. When predicting the arrival rate $\boldsymbol{\lambda}_{t+1}$, first $d = f(\boldsymbol{\lambda}_{t+1})$ is determined, and then only subset S_d is used for the forecast. In our evaluation, we use a fixed $D = \{workingday, nonworkingday\}$. More complex classifications are possible, however, it comes at the cost of an increased number of days required to learn the forecast model. As shown in Section 6.2.3, this classification significantly improves the forecasting accuracy on the considered real-world traces. Future work may extend splitTs to automatically cluster the different days based on historic data.

In order to obtain a forecast from a subset S_d, we use the WCF method described in (Herbst et al., 2014). WCF dynamically selects between different underlying statistical methods based on time series analysis depending on the forecasting objectives and incorporates direct feedback on the forecast accuracy.

6.2.2.2 Scaling Mechanism

The scaling mechanism expects the required memory allocation a_{t+1} for the next control interval as input. It first determines the new memory size (s_{t+1}) and memory limit (l_{t+1}) of the VM considering the current memory size and technical constraints from the hypervisor (e.g., VMware ESX expects the memory size to be a multiple of 128 MB). If a_{t+1} is larger than the VM's current memory size, the hot-add functionality of the hypervisor is used to add additional memory to the VM without restarting it. Furthermore, the memory limit is also set to the required memory size. If instead, a_{t+1} is less than its current memory size, only the memory limit is adjusted accordingly.

The individual steps to adjust the memory settings of a VM also include the necessary adjustments to the OS and application configurations. The following steps are executed in the given order:

1. The application is stopped in order to adjust static configuration settings (e.g., Java maximum heap space, or the database buffer pool size).

2. The new memory settings of the VM are fed to the hypervisor using the supported reconfiguration API.

3. Any additional memory is activated in the OS to become available to the memory scheduler. Depending on the OS, and its version this step is either triggered automatically by the OS or needs to be executed manually.

4. Any application-specific, memory-related configuration settings are updated to reflect the new memory size. The required automation scripts need to be provided by a system administrator and included in the VM.

5. The application is started with the new settings and it resumes serving user requests.

Two observations are worth noting. First, if memory usage is high, step 3) may fail if the OS cannot find enough free, contiguous physical memory to extend its memory management tables. This is why step 1) is executed before step 3) to reduce the memory usage in the VM and prevent reconfiguration failures. Second, steps 1) and 5) are not needed if the application is able to dynamically adapt its memory management to the new memory size of the VM.

6.2.3 Evaluation of Forecast Accuracy

In this section, we evaluate the splitTs forecaster with regards to the improvement in accuracy through exploiting calender meta-knowledge compared to state-of-the-art time-series forecasters.

6.2.3.1 Workload Traces

In order to evaluate the forecast accuracy, request arrival traces are needed that fulfill the following requirements: (a) they should contain request arrivals from a real-world application, (b) they should include daily patterns, as this is an assumption of our approach, (c) they should cover a period of several weeks (at least four weeks) in order to learn a forecast model that also captures weekly patterns. We found three request arrival traces fulfilling our requirements: *FIFA'98 World Cup*, *Wikipedia*, and *CICS transactions*. The FIFA'98 traces (Arlitt, 2000) were taken from the web servers of the official FIFA'98 world cup web site. We used the first five weeks of the traces for our evaluation. The wikipedia traces (Urdaneta et al., 2009) contain every 10th request to the official wikipedia site. For our evaluation, we used four weeks of trace data from the English language site (September 19th to October 21st, 2007). The CICS transaction time series reports the number of started transactions at a real-world deployment of an IBM z10 mainframe server. The trace data was taken from the case study described in (Herbst et al., 2014). We used a total of four weeks of this trace (January 31st to February 27th, 2011).

6.2.3.2 Error Metrics

We use the Mean Absolute Scaled Error (MASE) to evaluate the accuracy of the forecast values. Given a time series $Y_1 ... Y_n$ of actual observations from the

system and a time series of forecasts $F_1 ... F_n$, the forecast error e_t is generally defined as $e_t = F_t - Y_t$ for $t = 1...n$. Then the MASE is defined as:

$$MASE = mean\left(\frac{|e_t|}{\frac{1}{n-1}\sum_{i=2}^{n}|Y_i - Y_{i-1}|}\right) \tag{6.8}$$

MASE scales the error with the error from a one step naïve forecaster, which takes the last observation as the forecast. The MASE error is scale-independent and can be used for comparisons across multiple time series (Hyndman and Koehler, 2006). In contrast to the mean relative error, it is not influenced by skewed error distributions and thus ensures unbiased comparisons (Hyndman and Koehler, 2006). Assuming a forecast horizon of one interval, a MASE < 1 indicates that on average the considered forecast method yields smaller errors than the one step naïve approach. In our experiments, we forecast the workload for a complete day resulting in a forecast horizon of 24 or 48 depending on the sampling interval. Therefore, MASE is expected to be larger than one (Hyndman and Koehler, 2006).

6.2.3.3 Results

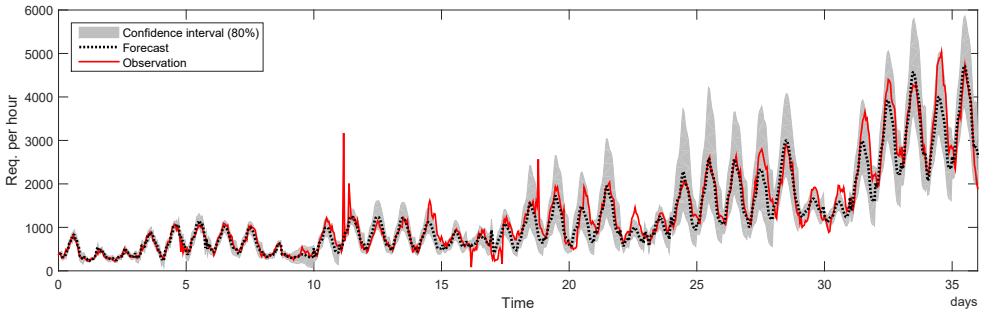


Figure 6.4: Forecasts using splitTs on the FIFA'98 traces (first day is Friday).

Figure 6.4 shows the resulting forecasts by the splitTs method on the FIFA'98 traces. The splitTs is able to capture the daily and weekly patterns (i.e., the differences between weekdays and weekends) in its model and to reflect them in the forecasts. The forecasting method is executed at 3AM each night when the workload is low. More information on the shape of the Wikipedia and CICS transaction traces can be found in (Urdaneta et al., 2009) and (Herbst et al., 2014).

In order to assess the improvement achieved in our splitTs approach, we compare its forecast accuracy to three state-of-the-art approaches, WCF (Herbst

et al., 2014), ARIMA (Box et al., 2008b) and tBATS (De Livera et al., 2011) (using their implementations from the R package `forecast`[2]). The first days required to learn the seasonal patterns are excluded from the comparison (in total 6 complete days as the splitTs approach requires three workdays and three non-workdays to learn the seasonal patterns).

		splitTs	WCF	ARIMA	tBATS
FIFA'98	MASE	1.42	2.18	2.65	2.33
	Infs	0	0	0	24
	C.I width	840	1075	852	954
Wikipedia	MASE	1.14	1.28	1.68	2.21
	Infs	0	0	0	0
	C.I width	39878	50820	58683	43323
CICS transactions	MASE	1.23	3.01	4.97	3.28
	Infs	0	0	0	0
	C.I width	9584	24774	15322	24536

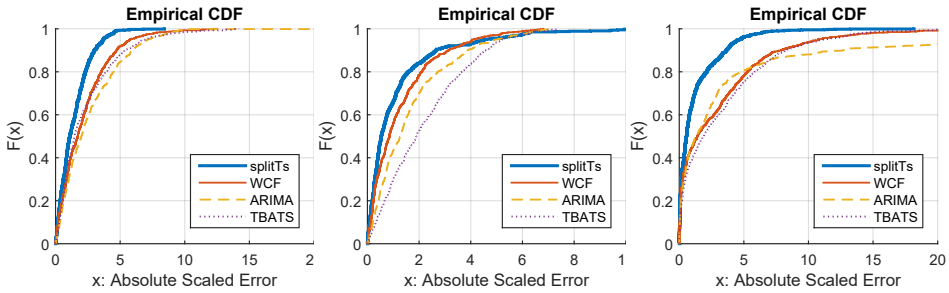
Table 6.1: Forecast accuracy.

Table 6.1 shows the summarized forecast errors of the splitTs compared with the WCF, ARIMA and tBATS methods. The splitTs approach can reduce the MASE errors by between 11% (Wikipedia traces) and 59% (CICS transaction traces). The differences can be explained by the different weekly patterns present in these two traces. The CICS transaction traces stem from an enterprise application and the differences between the weekend and weekday workloads are higher. In contrast, the Wikipedia traces have less distinctive weekend patterns and therefore, the improvements through the splitTs approach are smaller. Figure 6.5 shows the distribution of the absolute scaled errors.

In Table 6.1, we also included the number of infinite values forecast by the methods and the mean width of the 80% confidence interval. On the FIFA'98 traces, the tBATS approach fails to fit a forecast model in one interval resulting in 24 infinite values. The mean confidence interval length is significantly reduced by the splitTs on all three traces indicating a better fit of the forecast model to the observations.

In summary, the results show that by classifying different types of days and executing the time series analysis only on subsets with similar seasonal patterns, splitTs improves forecast accuracy significantly compared to existing state-of-the-art techniques based on time-series analysis. On the considered data sets, existing time-series analysis techniques were not able to forecast

[2] `http://cran.r-project.org/web/packages/forecast`



(a) FIFA'98 **(b)** Wikipedia **(c)** CICS transactions

Figure 6.5: Cumulative distribution function of absolute scaled errors.

the overlapping weekly seasonal patterns (workingdays vs. nonworkingdays) correctly.

6.3 Concluding Remarks

In this chapter, we proposed two autonomic controllers for vertical scaling of virtualized applications. Whereas the potential of horizontal scaling has been extensively evaluated in the literature before, vertical scaling based on hot-adding resources to a running VM is a relatively new way to dynamically adapt the resource allocations of virtualized applications. It can enable high elasticity through quick reconfiguration times if all platform layers support vertical scaling. This is typically the case for CPU resources, for which we propose a model-adaptive control loop with a short control interval. The short interval enables quick reactions even in case of unpredictable workload changes and the model allows us to predict the impact of a reconfiguration on the application performance in advance. Furthermore, we specifically designed our controller to be aware of physical resource contention effects in order to include this information during reasoning.

Our second autonomic controller leverages time-series forecasting techniques to proactively adapt resource allocations on a mid- to long-term scale. If applications are limited in their elasticity, reconfigurations need to be planned accordingly in advance to prepare the system for workload changes. In certain cases the reconfiguration may even cause service interruptions. Our controller is able to proactively schedule such reconfigurations (e.g., increasing the memory size) in a pre-defined maintenance window (e.g., during phases of low usage). We also propose a forecasting method incorporating calendar information and

evaluate it on three different request traces from real-world applications. The results show that incorporating calendar information in the forecast model significantly improves its accuracy compared to state-of-the-art statistical forecast methods. The MASE error metric is improved between 11% and 59% for all three traces.

Part III

Validation and Conclusions

Chapter 7

Validation

In this section, we provide an end-to-end validation of our approach to self-aware resource management in virtualized data centers. In the following, we first describe the goals of our evaluation, before we discuss our validation results.

7.1 Evaluation Goals

The main goal of this thesis is the development of a self-aware approach to resource management to improve the elasticity of virtualized applications. For our evaluation, we break this down into the following three evaluation goals:

- *Goal 1 (Applicability):* Our approach depends on a deep integration into state-of-the-art software systems in order to provide model learning capabilities transparently for a system administrator. Given that today a wide range of different applications are hosted in virtualized data centers, we focus on three example software systems that are often used in such a context. We consider the VMware vSphere virtualization platform, the Wildfly middleware platform and the Zimbra collaboration server. VMware vSphere is the market leader in the area of x86 server virtualization according to Gartner (Bittman et al., 2016). Wildfly is an open-source Java EE-compliant application server provided by Red Hat providing a middleware platform for custom Java enterprise applications. Zimbra is a collaboration server used by more than 500 service providers world-wide with over 400 million users according to the vendor's website[1]. We provide proof-of-concept implementations of VAs for these systems including integrated model learning capabilities for our reference architecture described in Chapter 4. In Section 7.2, we discuss these proof-of-concept implementations focusing on the interfaces these

[1] `www.zimbra.com`

systems provide to obtain the information required for model learning at system run-time.

- *Goal 2 (Automation):* Our approach aims at a high degree of automation for the resource management of applications in a virtualized data center. Given the complexity of modern systems, the manual creation and maintenance of detailed performance models of an application is infeasible. We evaluate the degree of automation achieved with our reference architecture for online model learning (see Chapter 4), as well as the accuracy of the resulting performance models. The latter is always a combined evaluation also considering the accuracy of the estimated resource demands using the method described in Chapter 5. In Section 7.4, we present a case study based on the SPECjEnterprise2010 full system benchmark using our Wildfly and VMware vSphere VAs developed as part of *Goal 1*. We determine the number of model elements that agents in the VAs can determine automatically and analyze the type of context information a user still needs to provide to obtain a complete model. In experiments with different workload intensities, we evaluate the prediction accuracy of the performance models resulting from online model learning by comparing its predictions to measurements from a real system. We perform additional case studies in Section 7.3 to validate our method for resource demand estimation comparing the estimated resource demands to directly measured ones and assessing the accuracy of resource demand estimation in multi-tenant environments.

- *Goal 3 (Elasticity):* Our approach should be able to adapt the resource allocations so that they match the demand of the application as closely as possible. If resources are under-provisioned, the application cannot fulfill its application SLOs. If they are over-provisioned, the resource efficiency is reduced. Furthermore, our approach should also avoid unnecessary reconfigurations which may incur additional overheads on a system. In a case study based on the Zimbra collaboration server with real-world workloads, we compare the two autonomic controllers proposed in Section 6 to a static allocation and to rule-based approaches under time-varying workloads. We focus on the virtual resources allocated to the individual VMs of a virtualized application. We assume that the virtualization platform is able to optimize the overall usage of physical resources in a data center if we can determine the required resource allocations to virtualized applications as closely as possible depending on their current workloads.

In the following, we describe the integration of our approach into existing software systems in Section 7.2. Furthermore, we present two use cases of our method to resource demand estimation as part of joint work with other researchers in Section 7.3. Then we show the results of our SPECjEnterprise2010 case study in Section 7.4 and of the Zimbra case study in Section 7.5.

7.2 Integration into Existing Software Systems

We describe our implementation of three VAs with integrated model learning capabilities. In Section 7.2.1, we analyze interfaces the VMware vSphere virtualization platform provides to obtain the current structure and configuration of the virtualized platform. Section 7.2.3 contains a description of the mechanisms we exploit in the Wildfly application server to discover the application architecture at system run-time and to automatically insert instrument points to collect empirical observation data. In Section 7.2.3, we describe an extension for the existing Zimbra VA to collect the monitoring data required for model learning.

7.2.1 VMware vSphere

In the following, we describe the prototype implementation of an model extraction agent for the data center scope and platform scope integrated with the VMware vSphere virtualization platform. We chose the VMware vSphere due to its wide-spread use in industry (Bittman et al., 2016). The agent itself runs in a system VM with access to the management network of VMware vSphere.

The VMware vSphere platform consists of the ESX hypervisor and the vCenter server for hypervisor management in a cluster of virtualized hosts. A vCenter server manages an inventory of the physical hardware, the hypervisor configuration and the VMs deployed in a cluster. System administrators can perform cluster-wide reconfigurations at a central place (e.g., deploy, migrate, or change resource allocations to VMs). Furthermore, the vCenter server collects and stores detailed monitoring statistics from all hypervisor instances. We access this information through a set of Simple Object Access Protocol (SOAP) web services provided by the vCenter server. The documentation of the web service interface is publicly available at the vendor's website (VMware, Inc., 2013).

The inventory is based on an object model representing all entities managed by a vCenter server. Figure 7.1 gives a simplified overview of the main classes in the object model. For easier representation, we excluded classes offered by

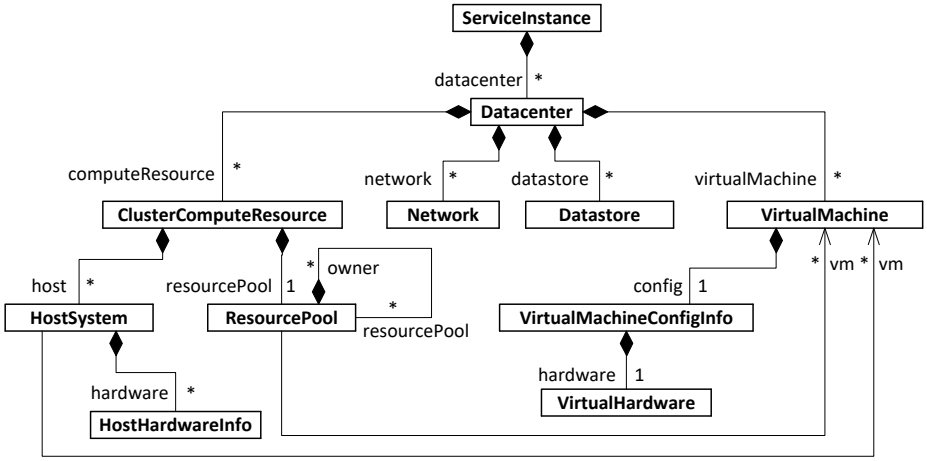


Figure 7.1: Excerpt of the vCenter object model (VMware, Inc., 2013).

vCenter to structure the inventory using folder hierarchies. When comparing the object model to the resource landscape model of DML (see Figure 2.4 on page 21), we can come up with a direct mapping between elements of the two models:

- Each `ServiceInstance` maps to a `DistributedDataCenter` and each child `Datacenter` object maps to the identical element in DML.
- A `ClusterComputeResource` is a logical grouping of host systems. Their hardware resources (e.g., CPUs and memory) are combined into one large resource pool from which they can be allocated to VM. Each cluster maps to a `CompositeHardwareInfrastructure` element in DML.
- A `HostSystem` maps to a `ComputeNode` in DML. The contained object `HostHardwareInfo` provides the information about hardware resources. They map to `ProcessingResourceSpecification` elements.
- Each `VirtualMachine` maps to a `RuntimeEnvironment` of type `OS_VM`. The virtual resources of a VM described in the `VirtualHardware` object are transformed to corresponding `ProcessingResourceSpecification` elements in DML. Furthermore, we check the current state of a VM in vCenter. If the VM is running, we determine its host system, and add the `RuntimeEnvironment` as a child to the corresponding `ComputeNode`. If not,

we treat it as a template and add the `RuntimeEnvironment` to the container repository model of DML.

- The `Network` and `DataStore` objects may be mapped to corresponding `NetworkInfrastructure` and `StorageNode` elements in DML. This is not implemented in the current prototype.

- A `ResourcePool` represents an aribtrary subset of all physical resources in a cluster. A resource pool limits the maximum amount of resources a group of VMs may consume in total across multiple host systems. Arbitrary hierarchies of resource pools are allowed for fine-grained control of resource consumption. Each `ResourcePool` object may be mapped to a specialized `ConfigurationSpecification` element in DML. However, current solvers for DML (Huber et al., 2017) do not support the concept of resource pools when predicting the performance of a system. This is a limitation of DML that needs to be addressed as part of future work.

This mapping shows that the extraction of a DML resource landscape model describing the data center and the virtualization platform is a straightforward mapping given the information in the vCenter inventory.

Clients can register at a vCenter server to be automatically notified of changes in the inventory – see the `PropertyCollector` managed object (VMware, Inc., 2013). This notification mechanism covers manual reconfigurations of a system administrator as well as any inventory changes from the system itself. Our agent registers for any changes that need to be reflected in the DML resource landscape model. Each notification contains a pointer to the changed objects, so that only the corresponding subset of the DML model needs to be updated.

7.2.2 Zimbra Collaboration Server

The Zimbra Collaboration Server (in the following we refer to it as Zimbra) is a groupware server based on common open-source components. It provides mail, calendar and address book functionality to users. According to vendor information on their website[2], Zimbra is used by over 500 service providers world-wide with more than 400 million users.

Application Architecture The architecture of Zimbra is divided into three main components: *mailbox server*, *Mail Transfer Agent (MTA)* and *LDAP server*. Each user of Zimbra has a mailbox on one mailbox server. The mailbox contains the

[2] http://www.zimbra.com

user's mails, calendars, address books, etc. The user can access the mailbox either using a Web interface provided by the mailbox server through HTTP(S) or using different desktop clients (SOAP, IMAP, or POP3). The mailbox server stores the mailbox data in different locations: a MySQL database contains all meta-data, whereas the content of mails, is stored directly in the file system. In addition, a Lucene search index is maintained to speed up full-text searches. When a user sends a mail, the mailbox server passes it to the MTA that delivers it to the recipient's server. The MTA may also receive mails from external internet servers. The MTA runs a number of checks on each mail it receives. Most noteworthy are the spam and virus checks. The LDAP server manages the central configuration for multiple Zimbra instances and handles the user authentication.

Virtual Appliance The developers of Zimbra offer a stock VA to quickly deploy a server instance in a virtualized data center. The VA conforms to the OVF standard and comes with an integrated installer program that is started on its first boot. The installer asks for a couple of configuration options from a deployer and initializes the Zimbra instance accordingly.

Zimbra supports different deployment alternatives of its components. When deploying a VA, one can choose between the following alternatives:

1. All servers in the same VM.
2. Mailbox and LDAP server in one VM and the MTA in a separate VM.
3. Mailbox, LDAP and MTA server each in a separate VM.

In the following explanations and case studies, we assume the second deployment option.

The stock VA does not come with integrated model learning capabilities. We have extended this VA with an additional operating system service that provides these capabilities. It runs in a separate process and accesses the Zimbra server through publicly documented interfaces as described in the following paragraphs. The deployer of our extended VA needs to provide the location and the user credentials of a model repository as parameters when booting it for the first time.

Static Analysis Our extended VA includes a pre-determined model skeleton that is loaded into the central model repository during start. The model skeleton abstracts the application at a high level. Components are the mailbox server and the MTA. Only the control flow between these two components is considered.

The reason for this coarse-grained modeling is the complexity of the underlying application and the lack of documentation of its internal structure and behavior. However, as we will see later in the case studies on vertical scaling this model is already sufficient to help with resource allocation decisions.

Before loading the model skeleton into the model repository, we determine the hostname of mailbox and MTA servers using the `zmprov` administration tool of Zimbra. The names of model elements are renamed accordingly to make them unique and discoverable.

Instrumentation We reuse the instrumentation already included in the stock VA to determine the throughput and response times of requests. On the mailbox server the access and response time statistics are logged to the file `/opt/zimbra/zmstat/soap.csv` with a resolution of one minute. These statistics are directly forwarded to the model repository. In the case of the MTA, the individual steps when processing mails are reported in detail to the log file `/var/log/maillog`.

Dynamic Analysis On the MTA, it is necessary to determine the control flow of mails and calculate the end-to-end response of mails. For this purpose, we continuously parse the log file `/var/log/maillog`. The log file contains entries for each individual mail after it passes a certain step in the processing pipeline (e.g., spam or virus check). The log entry contains the queueing and processing times of a mail in each step. In addition, it contains a unique message identifier that stays the same until a mail is completely processed in the MTA. We correlate the log entries using the message identifier in order to calculate the end-to-end response time.

7.2.3 Wildfly Application Server

Wildfly (formerly known as JBoss) is an open-source application server[3] fully compliant with the Java EE standard. It is written in Java and runs on any standard Java VM. We built a virtual applicance based on a CentOS 6 Linux operation system, an OpenJDK 7 Java VM and the Wildfy 8.2 application server. Several Wildfly instances can form a cluster to fulfill high-availability goals using replication and load-balancing techniques. Our virtual appliance is configured to support clustering and runs in a domain mode for easier cluster management. There are two types of Wildfly server instances:

- The *Wildfly domain controller* provides centralized configuration management for a cluster of Wildfly server instances. The domain controller acts

[3] http://wildfly.org/

as a master node in the cluster to which all slave nodes connect on start up to obtain their initial configuration. When the configuration on the domain controller changes at run-time, the slave nodes are notified and update their local configuration. The Wildfly domain controller typically does not serve any production workloads.

- A *Wildfly slave node* runs the actual applications. A cluster typically consists of several slave nodes, which are potentially replicated for better availability. Replicated instances always synchronize their session state between each other, so that one node can take over the processing of a failed node without service interruption.

Virtual Appliances We created separate VAs for these two types of servers. The domain controller VA needs to be started before any slave nodes. On instantiation of a slave VA, a user needs to provide values for the following parameters:

- A *node name* that uniquely identifies the slave node within a cluster.
- The *IP address* of the domain controller.
- A *secret* used to authenticate at the domain controller.
- A *server group identifier* that is used to lookup the actual configuration on the domain controller. The configuration of slave nodes may differ between server groups.

After the user has provided this information, the slave node can retrieve its configuration from the domain controller. The configuration also determines which applications will be deployed on a slave node.

On initialization of a slave VA, it is not yet clear which application components will be deployed on this instance. Furthermore, the deployment of components may change dynamically during system run-time. We need to determine at run-time which components are deployed on a server, as well as the control flow between these components. The latter requires the insertion of instrumentation points at providing and requiring interface roles to observe the inwards and outwards flow of requests. We developed a custom module for the Wildfly server that is loaded directly into the server process and that has full access to the current server state. The module automatically intercepts all deployments of components and inserts the required instrumentation points. In the following, we describe the static analysis steps that are performed by our module when new components are deployed, and then we give an overview of the employed

instrumentation techniques. Finally, we describe the dynamic analysis steps implemented in our module.

Static Analysis Applications need to comply to the Java EE 7 standard (Oracle America, 2013). The standard distinguishes between four different component types (Oracle America, 2013) :

- *Web Components* "typically execute in a web container and may respond to HTTP requests from web clients" (Oracle America, 2013, p. 8). Web components are either Servlets, Java Server Pages (JSP) pages, Java Server Faces (JSF) applications, filters, or web event listeners.

- *Enterprise Java Beans (EJBs)* are transactional components for executing the application business logic. Clients may invoke EJBs either using local calls if executed in the same container, using remote method invocations – e.g., Remote Method Invocation (RMI) or Internet Inter-ORB Protocol (IIOP) –, or using web service protocols – e.g., SOAP.

- *Application Client Components* are "Java [...] programs that are typically GUI programs that execute on a desktop computer" (Oracle America, 2013, p. 8).

- *Applets* are Java GUI components that can be embedded into HTML pages and are executed by an internet browser.

The model extraction focuses on components running inside the data center and does not include client components (i.e., application client components and applets).

When the component deployment on a Wildfly slave node changes, our model extraction logic needs to be informed of these changes. Wildfly offers an extension point to provide custom deployment unit processors that are invoked when application modules are added or removed. Custom deployment unit processors need to implement the `org.jboss.as.server.deployment.DeploymenUnitProcessor` interface. We provide an implementation that does the following steps for each application module:

- *Static analysis of component structure:* It determines the components contained in an application module as well as their type (i.e., web or EJB components). For each component, it identifies the interface providing ports.

- *Instrumentation setup:* It adds instrumentation points to observe incoming and outgoing invocations of a component. This information is required for the dynamic analysis.

Interface requiring roles are difficult to determine statically. Java EE provides two different ways to obtain references to required components: *dependency injection* and *Java Naming and Directory Interface (JNDI) lookup*. In the former case, the container already knows all required components at deploy time. However, in the latter case, an analysis of the complete bytecode of a component would be required to find all JNDI lookups of required components. To avoid a full bytecode analysis, we resort to dynamic analysis techniques to determine the interface requiring ports of components.

Instrumentation In order to observe the control flow of the application, we instrument its components at different locations. The focus lies on the inter-component control flow, i.e., we want to observe external calls of one component to another one. We abstract the intra-component control flow, i.e., we extract coarse-grained service behaviors in DML.

Table 7.1 lists the types of instrumentation points used by our module including the interception techniques we are using.

- *HTTP handlers* are classes that implement the interface `io.undertow.server.HttpHandler`. These classes can be integrated into the HTTP request pipeline of a Wildfly server and are called before any servlets, JSPs, JSF pages, or filters are executed. HTTP handlers have access to the HTTP request and the response data. We preferred the proprietary HTTP handlers to the standard Java EE filters, as they can be configured once for a complete server. Furthermore, they also have access to the arrival timestamp of each request enabling more accurate response time measurements.

- *View interceptors* need to implement the interface `org.jboss.invocation.Interceptor`. These interceptors are invoked on the server side before a request is passed to an EJB or web service. Our deployment unit processor automatically registers an interceptor at each interface providing port of a newly deployed component.

- *Client interceptors* are called on the client side for each outgoing invocation to a EJB component. They need to extend the interface `org.jboss.ejb.client.EJBClientInterceptor`. These interceptors need to be registered only once for each application module. Our deployment unit processor automatically registers an interceptor for each application module.

Web Components		
Servlets	HTTP	HTTP handler
	SOAP	View interceptor
JSP	HTTP	HTTP handler
JSF	HTTP	HTTP handler
Filter	HTTP	HTTP handler
Web event listener	Java	-
EJBs		
Session bean	Java	View interceptor
	RMI/IIOP	View interceptor
	SOAP	View interceptor
Message-driven bean	JMS	View interceptor
Entity bean	Java	-
Backend services		
EJBs	Java	Client interceptor
	RMI/IIOP	Client interceptor
Web services	SOAP	SOAP handler
Data source	JDBC	Statement interceptor
Messaging	JMS	JNDI delegator
Mail session	SMTP	-
JCA	Custom	-

Table 7.1: Instrumentation points.

- *SOAP handlers* are integrated into the request pipeline of outgoing web service invocations. They need to extend the `org.jboss.ws.api.handler.GenericSOAPHandler` interface and are configured server-wide.

- *Statement interceptor* are a mechanism provided by the Java Database Connectivity (JDBC) driver of MySQL to intercept all SQL statements sent to a database server. The `com.mysql.jdbc.StatementInterceptorV2` interface needs to be implemented by such interceptors.

- In the case of a *JNDI delegator*, we replace the actual resource (e.g., a JMS queue) in the JNDI context with a delegator intercepting all calls before forwarding them to the original resource. This is transparent to the application as long as the resource can be accessed through a standardized interface.

We do not observe calls to web event listeners, entity beans, mail sessions and Java EE Connector Architecture (JCA). Web event listeners are not directly related to individual requests and therefore not included in the extracted model. Entity beans are simple Java objects since Java EE version 5. We do not model them as explicit components in the model, the overhead induced by them is included in the enclosing component that uses an entity bean. The lack of support for mail sessions is currently a limitation of our model extraction. JCA does not support generic intercepting of requests. A solution can only be found for concrete connector implementations.

Dynamic Analysis The interceptors count each incoming or outgoing invocation and measure its execution time. We do not maintain statistics for each individual invocation in order to keep the volume of monitoring data low. Instead, we accumulate the values for each component service and each external call within a component. In regular intervals (e.g., every minute), we send the aggregated statistics to the performance model repository. If we detect a component service or an external call that has not been observed before, we trigger an update of the model skeleton. The update is performed asynchronously to avoid delaying the application processing.

Each new component service or external call is added to the model skeleton initially created when performing the static analysis of the component structure during deployment. In case of external calls, we automatically add assembly connectors between source and target component instances if required. We use technical identifiers (e.g., URLs) as used by the Wildfly server internally to identify components in a unique way. The assumption is that we can always determine the target component instance on the client-side. Updates to the model skeleton are sent in batches to the performance model repository in order to reduce communication overhead.

7.2.4 Discussion

The presented proof-of-concept implementations of VAs with integrated model-learning capabilities demonstrate the applicability of our approach in practical software systems. We considered the integration of model learning capabilities into a state-of-the-art virtualization platform (VMware vSphere), an open-source Java EE middleware platform (Wildfly) and a widely used standard enterprise application (Zimbra collaboration server). The experience with these VAs shows that static and dynamic analysis techniques can be integrated deeply into the corresponding software systems relying on structural and monitoring data routinely collected at system run-time. However, it also shows

that the design and implementation of such model learning capabilities can be a complex and time-consuming task requiring a profound understanding of the system architecture and of – potentially proprietary – interfaces to obtain its current state. Therefore, we expect that such VAs will typically be built by the vendor of a software system, or an independent system expert. A system administrator would only need to set up the platform components of our reference architecture (i.e., performance model repository, and message bus), and deploy the third-party VAs.

7.3 Use Cases of LibReDE

LibReDE is designed as a library for usage in different tools for performance model extraction and resource management. In the following, we describe collaborations with other researchers where we have used LibReDE in the context of their work.

7.3.1 Resource Usage Control in SAP HANA Cloud

Multi-tenant Software-as-a-Service (SaaS) cloud environments require a mechanism for performance isolation between different tenants. Multi-tenancy refers to an architecture where groups of users from different customers share the same instance of a software application. Performance isolation is a quality attribute of a multi-tenant system and describes the degree to which "for customers working within their quotas the performance is not affected when other customers exceed their quotas" (Krebs et al., 2014b).

Operating systems are not aware of different tenants in application layers and can only provide performance isolation at the level of system processes. As a result, the design of multi-tenant applications requires additional architectural concerns to ensure performance isolation. Admission control mechanisms are commonly used to throttle requests from customers exceeding their quotas. However, state-of-the-art admission controllers typically work with fixed limits on the number of requests and do not consider the actual resource usage of a tenant.

Krebs (2015) proposes different autonomic controllers in his PhD thesis to ensure performance isolation in multi-tenant Cloud environments. We developed one of these controller as part of joint work published in Krebs et al. (2014a). Our controller is utilizing resource demands of individual requests to control the resource usage of tenants. In the context of this thesis, we contributed estimation approaches based on LibReDE required to obtain the resource de-

mands to this work. In the following, we provide a short description of the controller for performance isolation including the integration of our estimation approaches. Finally, we give an overview of the results of our joint work.

The resource usage controller distinguishes between tenants $t \in T$ and request types $c \in C$. At system run-time, we observe the average throughput $X_{t,c}$ and the end-to-end response time $R_{t,c}$ for each tenant t and request type C. Furthermore, we observe the aggregate CPU utilization U as reported by the operation system. Given these observations, we estimate the resource demands $D_{t,c}$ for each tenant and request type. We distinguish between tenants as their resource demands may differ significantly (e.g., due to different database sizes). However, this can quickly result in a high number of different resource demands to be estimated – in total $|T| \times |C|$ values.

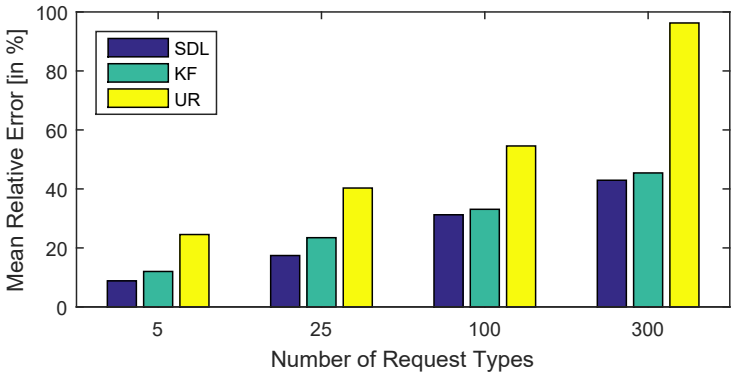


Figure 7.2: Accuracy of resource demands with different numbers of tenants and requests.

Figure 7.2 shows the total mean relative error of the estimated resource demands for different numbers of request types. For this experiment, we used a queueing simulator with a closed QN. Each tenant (in total 5 tenants) is represented by a separate queue with infinite server scheduling and an exponentially distributed think time with a mean value of 6 seconds. The application is modeled as a single queue with FCFS scheduling. The resource demands follow a Gaussian distribution with randomly chosen parameters. An analysis of the requirements for our resource usage controller yielded three candidate estimation approaches: Service Demand Law (*SDL*) (Brosig et al., 2011), Kalman Filter based on Utilization Law (*KF*) (Wang et al., 2012), and least-square regression based on Utilization Law (*UR*) (Rolia and Vetland, 1995). In all cases, *UR* shows only a slow convergence resulting in an error at the end

of the experiment of almost double the value of the other approaches. *KF* and *SDL* yield similar accuracy. However, *KF* required a longer convergence period, therefore, *SDL* was chosen as the most appropriate estimation approach.

To evaluate the effectiveness of our resource usage controller based on the *SDL* estimation approach, we performed a case study in the SAP HANA cloud environment. The multi-tenancy application benchmark MT-TPC-W (Krebs et al., 2013) serves as the application and an admission controller exploits the resource demands estimated for individual requests to determine the total resource usage of each tenant. If a tenant exceeds its resource quota, its requests are queued for a certain time in order to avoid negative effects on other tenants in the system. The results published in (Krebs et al., 2014a) show that our controller can effectively enforce performance isolation between tenants sharing the same application instance. It is able to quickly adapt to changes in the workload and adjust the priorities between tenants in the admission control accordingly.

7.3.2 Offline Generation of Palladio Component Models

In joint work with researchers from the Fortiss research institute, we integrated LibReDE into their Performance Management Works (PMW) tool (Brunnert et al., 2013). PMW supports the automatic generation of PCM instances based on traces of requests collected at a real system. PCM is a domain-specific modeling language for analyzing the quality properties of system architectures at design time. Resource demands are required parameters to enable performance predictions. So far, PMW depends on direct measurements of the resource demands. In Willnecker et al. (2015), we integrated LibReDE with PMW and compared the resulting resource demands with directly measured values.

Compuware Dynatrace is a leading provider of APM tools. Dynatrace supports the tracing of individual transactions in an application across multiple tiers. It uses high-resolution timers to measure the CPU time consumed by individual threads in an application. Through fine-grained instrumentation of the application, it can determine which requests a thread is currently processing and thus it can attribute the measured per-thread CPU time to individual requests. PMW uses the traces collected by Dynatrace to generate PCM instances capturing the observed control flows.

Our experiment environment consists of a JBoss 7.0 application server and a Derby database both deployed in the same VM. We use the SPECjEnterprise2010 benchmark in a non-distributed setup with disabled manufacturing and supplier domains. In an experiment with 600 concurrent users, we generated two different PCM instances: the first with the resource demands measured by Dy-

natrace, and the second with the ones estimated by LibReDE. We let LibReDE automatically decide which estimation approach to use as described in Section 5.3. We then compared the predictions of these models with measurements at the real system.

		Dynatrace	LibReDE
CPU utilization		1.53%	1.01%
Response time	Browse	8.86%	14.61%
	Manage	4.55%	2.74%
	Purchase	14.27%	5.19%

Table 7.2: Comparison of Dynatrace and LibReDE (the values are the mean relative errors between the predicted and measured values).

Table 7.2 shows the resulting mean relative errors between the predicted and the measured utilization and response times. LibReDE is able to provide resource demands of similar accuracy compared to Dynatrace. In contrast to Dynatrace, it however only uses measurements of the end-to-end response time at the system entry points. In contrast, Dynatrace requires a fine-grained instrumentation of the application which is limited to certain platforms and may introduce additional overheads. We performed additional experiments with varying workloads and configurations which sustain the hypothesis that LibReDE is able to provide similar accuracy compared to direct measurements. We refer the interested reader to Willnecker et al. (2015).

7.3.3 Discussion

In this section, we presented two collaborations with external researchers using LibReDE in combination with their approaches. LibReDE is a generic library that can also help to solve problems outside the scope of this thesis as demonstrated by these use cases. The experiences of these collaborations influenced the design of LibReDE as a library simplifying the integration into other tools. Furthermore, the results show that LibReDE can provide accurate resource demand estimates that can compete with state-of-the-art methods for their direct measurement.

7.4 Case Study: Distributed SPECjEnterprise2010

SPECjEnterprise2010 is an industry-standard full system benchmark for Java EE application servers[4]. The goal of the benchmark is to enable the comparison of different Java EE application servers with regards to their scalability and their efficiency under real-world applications. It covers the full system stack and uses an application workload representative of many real-world enterprise systems. The benchmark is designed to exploit a large set of different Java EE 5 technologies covering dynamic web pages (Servlets and Java Server Pages), web services, (distributed) transactional EJBs, asynchronous messaging (Java Messaging Service) and object persistence (Java Persistence API).

In this case study, we evaluate the degree of automation and the prediction accuracy of the models obtained using our reference architecture for online performance model extraction (see Chapter 4). We especially consider the estimation of resource demands validating our method described in Section 5.3. We chose the SPECjEnterprise2010 benchmark as it provides a complex workload representative of many real-world enterprise applications and exploits a broad set of technologies of the Java EE standard. These properties make the benchmark also an ideal candidate to evaluate the capabilities of approaches for performance model learning. As a result, it has become a de-facto benchmark in this research area (e.g., Brosig et al., 2011; Brunnert et al., 2013).

Workload The benchmark workload consists of Customer Relationship Management (CRM), manufacturing and supply-chain management applications. The business scenario of the benchmark is modeled after an automotive manufacturer with car dealerships, manufacturing sites and suppliers interacting with the system. Car dealerships use an interactive web applications to access the order domain where they can browse, purchase and sell cars. The manufacturing sites use remote EJB and web service calls to start and complete manufacturing processes in the manufacturing domain. Suppliers are triggered through a web service interface by the supplier domain if parts need to be purchased for manufacturing.

SPECjEnterprise2010 comes with two workload drivers: one for generating workloads from car dealerships (*DealerDriver*) and the other for the manufacturing sites (*MfgDriver*). The generated workloads are based on transactions;

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

each transaction sending a sequence of different requests to the system. Car dealerships interact with the order domain using either *Browse*, *Purchase* or *Manage* transactions. The *Browse* transaction is dominated by read requests, whereas the latter two transactions are a mixture of read and write requests. The manufacturing sites communicate either through a SOAP-based web service or through binary RMI-based protocols. We distinguish between *Mfg WS* and *Mfg EJB* transactions accordingly. The sequence of requests in a transaction is defined by a first-order Markov chain. We use the standard workloads as defined in the standard (Standard Performance Evaluation Corporation, 2010). The workload drivers can be configured with a transaction rate which determines the number of concurrent threads in the load driver sending requests to the system. The transaction rate scales the interarrival times of the different types of transactions accordingly.

The external suppliers are represented by one or multiple emulators, which wait for requests from the supplier domain. It simulates the processing of purchase orders for components required to manufacture a car in the manufacturing domain. After receiving a purchase order, it sleeps for a certain time defined by the lead time of the requested component. Then it signals the shipment of the component to the supplier domain.

Deployment The benchmark is originally designed for a three-tier deployment, consisting of a web, an application and a database server. In addition, the three domains may be deployed on separate servers. However, modern enterprise applications often follow a service-oriented paradigm implementing the functionality as multiple independent services that can be deployed separately. In order to better reflect the architecture of a service-oriented, distributed system, we adapted the SPECjEnterprise2010 benchmark, so that the EJBs in the business logic tier can be deployed individually as services. Figure 7.3 shows the resulting deployment. The benchmark is deployed on a cluster of Wildfly 8.2 application servers. The order domain is distributed over several fine-granular services each deployed in a separate virtual machine. The services of the manufacturing and supplier domains are all deployed in the same VM. The data tier is shared by all business services and hosted by a MySQL 5.6.25 relational database. The communication between business services is based on the RMI protocol as provided by the application server. The relational database is accessed using the standard JDBC drivers provided by MySQL.

The physical resource environment consists of 4 servers, each equipped with 1 Intel Xeon E3-1230 CPU with 4 cores, 16 GB main memory, 500 GB HDD and 1 Gbit network connection. VMware vSphere 5.5 is used as hypervisor. All VMs

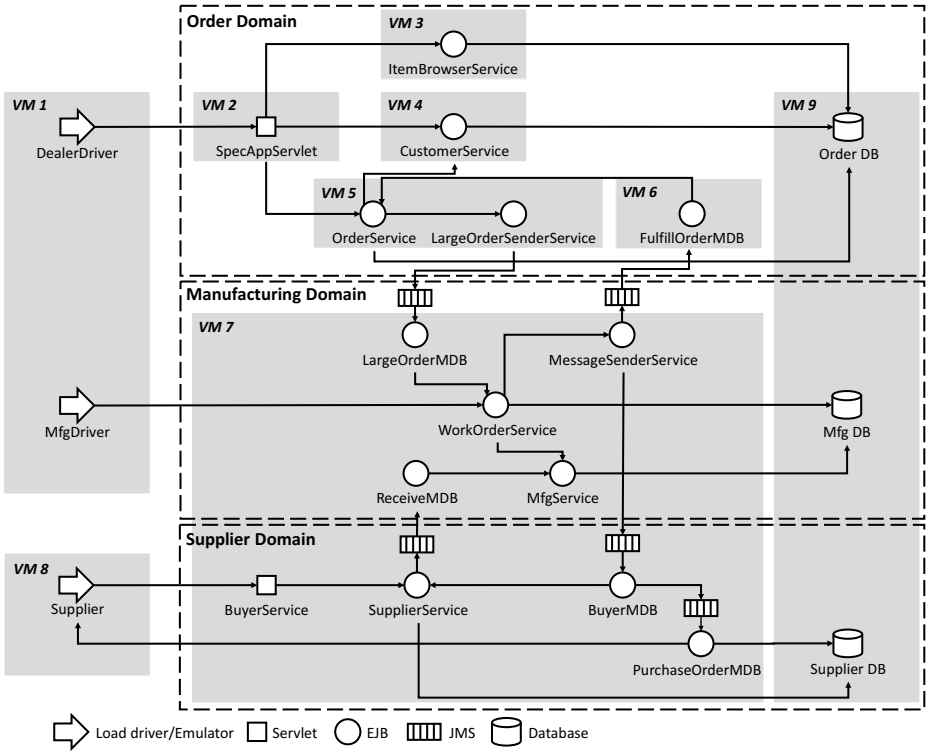


Figure 7.3: Distributed deployment of SPECjEnterprise2010.

are equipped 1 virtual CPU and 4 GB memory, except for VM 7 (2 vCPUs and 8 GB memory) and VM 9 (2 vCPUs and 4 GB memory) which have a higher resource requirement. The VMs are distributed evenly between hosts so that resources are not over-committed (Host 1: VM 1, VM 8; Host 2: VM 3, VM 5, VM 6; Host 3: VM 2, VM 7; Host 4: VM 9, VM 4). Each VM runs a CentOS 6.6 Linux 64-bit operating system.

Experiment Runs We collected the observations of throughput, response and residence times, as well as the CPU utilization of the different VMs used for model learning during an eight hour benchmark run at a transaction rate of 60. Our evaluation is based on the DML model extracted automatically by our Wildfly agents (see Section 7.2.3) as the basis for our evaluation. For validation purposes, we performed five more benchmark runs each with a duration of one hour varying the transaction rates between 20 (corresponds to a lightly utilized system) and 100 (which is close to the maximum sustainable load at VM 3).

7.4.1 Degree of Automation

We rely on the VAs for the Wildfly application server and the VSphere virtualization platform (see Section 7.2) to extract a DML model at run-time. In addition, we created a VA for the MySQL database server, which delivers a static, pre-determined model skeleton on start up. The following agent roles (see Section 4.2) are automated in this case study:

- VSphere platform: *Data Center Structure* (D1), *Compute Node Configuration*, *Platform Configuration* (P1), *Platform Monitoring* (P3).
- Wildfly VA: *Component Boundary* (A1), *Fine-Grained Behavior* (A4), *Component Assembly* (A2), *Application Monitoring* (A8), and *Component Deployment* (A3).
- MySQL VA: *Component Boundary* (A1) and *Component Deployment* (A3). The database server is represented by a single component. The agent only delivers the interface providing roles of this component. The services including their behavior description is created by the Wildfly agent when it detects JDBC calls to the database.

In order to obtain a complete DML model, we performed additional manual steps in this case study. The agent roles *System Interface Providing Roles* (D5), *Application Assembly* (D6), and all agent roles in the usage scope were not covered by automated agents.

Table 7.3 depicts the statistics for the degree of automation achieved for the DML model of the SPECjEnterprise2010 benchmark extracted during our experiments. The "Others" category subsumes model entities with no own identity. The resulting model consists of six submodels containing a total of 1739 model entities. Out of these model entities 253 were created manually, resulting in an overall degree of automation of 85.5%. In the following, we discuss how the current limitations can be relieved to reach the goal of full automation:

- The manual effort for creating the usage profile makes up for 11.9% of the model entities. In Section 4.2.2, we list several techniques that can be used here. For instance, Hoorn et al. (2014) propose an technique to automatically extract usage profiles that they have already successfully validated with the SPECjEnterprise2010 workload (Hoorn et al., 2015). We leave the integration of such techniques into our reference architecture as future work.

Model Entity	All	Manual
Interfaces	16	-
Operations	79	-
Basic Components	18	1
Actions	250	6
Resource Demands	80	-
Composite Components	9	-
Subsystems	2	2
Interface Roles	114	7
Assembly Contexts	33	3
Assembly Connectors	31	2
Delegation Connectors	44	5
Deployment Contexts	14	-
Compute Nodes	3	-
Runtinme Environments	17	-
Usage Scenarios	5	5
User Actions	57	57
Others	967	165
Total	1739	253

Table 7.3: List of extracted model entities.

- The emulator is not part of the extraction as it represents an external component, which may be hosted outside of the data center. For performance prediction purposes, we manually added a component representing the emulator in our model. 1.9% of the model entities account for the emulator. Future work should consider the automatic extraction of more appropriate models for external services.

- The overall application assembly, i.e., the applications in a data center and the communication paths between applications need to be defined manually. The monitoring of the control flow across different applications in a data center is an open challenge. Today's monitoring tools for applications are mostly focused on single applications. However, a system administrators typically needs to know the high-level control flow of an application (i.e., the externally provided and required services) anyways to configure the system correctly. Therefore, we think that it would be an acceptable manual effort. In our case study, the manual effort for the

application assembly only makes up 0.7% of all model entities.

Manual parts can be easily integrated into the end-to-end performance model by creating model skeletons by hand. As model skeletons are valid instances of DML, the existing Eclipse tooling for the graphical and textual editing of DML models (Brosig, 2014) can be used to create them. They can be uploaded to our model repository through the same interface used by the model extraction agents.

7.4.2 Accuracy of Estimated Resource Demands

LibReDE automatically selects an estimation approach using the cross-validation scheme described in Section 5.3.4. The approach selection relies on a sufficiently accurate and fast way to predict the utilization and response time based on the estimated resource demands to perform the cross-validation. First we compare the response times and utilization predicted by LibReDE to a state-of-the-art simulation solver of DML. Then we show the results of the cross-validation; finally we evaluate the overhead required by the different approaches to resource demand estimation.

We consider five estimation approaches here (see Chapter 5 for a description):

- *LO*: Recursive optimization using response times and utilization observations (Liu et al., 2006)
- *RR*: Least-squares regression using response time and queue lengths (Kraft et al., 2009)
- *UR*: Least-squares regression using utilization (Rolia and Vetland, 1995)
- *SDL*: Service Demand Law (Brosig et al., 2009)
- *KF*: Kalman filter using response time and utilization (Zheng et al., 2008)

We do not consider any estimation approaches based on observed residence times here, due to the reasons discussed later. The observations are averages over 15 minutes and we used a sliding window of 60 samples. The observation data is split into 5 equally sized sample sets for cross-validation.

Accuracy of Fitness Function The cross-validation of LibReDE depends on an accurate and fast calculation of response times and utilization in order to obtain reliable results for approach selection. Given that the cross-validation of the estimated resource demands should be performed continuously, simulation

techniques are typically too expensive. A DML model may contain fine-grained service behavior description, for which no general exact analytical solution exists. However, our model-to-model transformation (see Section 5.3.2) to the workload description which is input to LibReDE abstracts from certain details enabling the mapping to a product-form QN.

We compare response times and the utilization predicted by LibReDE for cross-validation purposes with the results of a full simulation of the resulting DML model. The DML model contained the same resource demand values. We use the QPN solver of DML (see Huber et al., 2017) which is based on the SimQPN simulator (Kounev and Buchmann, 2006).

	LO	RR	UR	SDL	KF
VM2	0.72%	0.86%	0.80%	0.79%	0.80%
VM3	0.91%	0.59%	0.84%	1.05%	1.19%
VM4	0.14%	0.05%	0.18%	0.29%	0.19%
VM5	0.07%	0.20%	0.21%	0.21%	0.19%
VM6	-0.00%	0.00%	0.05%	0.05%	0.05%
VM7	-0.19%	-0.15%	-0.30%	-0.23%	-0.11%
VM9	-0.03%	0.34%	0.13%	0.13%	0.21%

Table 7.4: Absolute errors between the calculated utilization and the simulation results.

		LO	RR	UR	SDL	KF
Purchase	Calc.	43.8	51.4	49.1	56.8	48.9
	Sim.	46.0 +/- 6.1	48.8 +/- 5.0	51.1 +/- 6.8	58.7 +/- 5.8	50.8 +/- 6.7
Manage	Calc.	32.8	48.3	24.2	32.1	36.4
	Sim.	33.8 +/- 3.8	42.9 +/- 5.7	25.0 +/- 2.5	33.4 +/- 4.4	38.5 +/- 5.1
Browse	Calc.	73.0	78.9	82.6	76.5	85.1
	Sim.	75.4 +/- 9.6	73.5 +/- 5.5	84.9 +/- 10.1	77.3 +/- 6.8	86.0 +/- 9.7
Mfg EJB	Calc.	19.1	**32.4**	**32.4**	13.8	**22.1**
	Sim.	19.0 +/- 1.3	27.0 +/- 2.5	31.0 +/- 1.2	13.6 +/- 0.5	21.2 +/- 0.8
Mfg WS	Calc.	19.3	**32.5**	6.0	13.8	**22.3**
	Sim.	19.3 +/- 1.3	27.1 +/- 2.4	6.2 +/- 0.2	13.7 +/- 0.5	21.3 +/- 0.7

Table 7.5: Calculated (Calc.) and simulated (Sim.) response times including 95% c.i. Calculated values outside of the c.i. are bold.

Table 7.4 and 7.5 compare the results calculated by LibReDE for cross-validation with the results obtained by the simulation. Table 7.4 shows the utilization of each VM in our application. Table 7.5 shows the response time of the different types of transactions. For the simulation results, we also list the 95% confidence

interval of the response time. The results show that both the utilization and the response times calculated by LibReDE are close to the ones obtained using simulation of the complete DML model. The mean response times from LibReDE are mostly within the confidence interval of the simulation results. Significant deviations are visible for the response times of the *Mfg EJB* and *Mfg WS* transactions. We explain these differences with the insufficient representation of the asynchronuous message-based communication between the manufacturing domain and the other domains. LibReDE does not capture this when solving the QN. It represents each asynchronous flow as an external entry point of requests and uses the observed throughput of requests as the arrival rate of these requests. Therefore, the causal relationship between the processing on the side of the sender of a message and its impact on the arrival process at the receiver side is lost.

	KF	RR	LO	UR	SDL
Cross-Validation Error [%]	7.3	31.1	1.7	29.9	17.2
Execution Time per Iteration [s]	0.010	0.094	9.644	0.001	0.002

Table 7.6: Cross-validation errors and execution time.

Fitting Accuracy Table 7.6 shows the results from the cross-validation. The error is calculated using Equation 5.7 on page 149. The error covers the complete experiment duration and is the average of all five folds of the cross-validation. Furthermore, we also measured the execution time of the estimation approach per iteration. We updated the resource demands estimates iteratively every 15 minutes.

The results show that *LO* yields the lowest overall cross-validation errors at the cost of significantly higher execution times. However, given an iteration length of 15 minutes, *LO* is still sufficiently fast to provide updates in time for online estimation. Furthermore, it may be sped up at the cost of accuracy by lowering the maximum number of iterations the underlying optimization algorithm searches for a solution. As an alternative, *KF* may provide estimates with reasonable cross-validation errors at a significantly lower computational cost.

Residence Time Measurements In distributed systems, the residence time is the time a request is processed at a node in the system, whereas the end-to-end response time describes the total processing time of this request in the system.

When estimating resource demands using response-time based techniques (e.g., Liu et al., 2006; Zheng et al., 2008), there are two alternatives: repeatedly apply the technique for each node separately using the observed residence times, or apply the technique on a system level using the observed end-to-end response times. Assuming that both metrics can be observed at a system, the advantage of a per-node approach is the reduced complexity of the estimation problem: the control flow between nodes does not need to be considered for resource demand estimation and the number of resource demands that need to be estimated for each individual node is lower than the total number for the complete system. However, a system-wide approach promises to be able to reduce estimation error across all nodes.

We also applied the *LO* approach using residence time measurements on a per-node level. We observed significantly higher cross-validation errors (53.5% compared to 7.3% with end-to-end response times). A further analysis using tracing tools to measure CPU times showed that a significant amount of CPU time is consumed by request processing before our instrumentation points for residence time measurement in the Wildfly application server are executed. Therefore, the measured residence times are not accurate enough for resource demand estimation.

7.4.3 Model Prediction Accuracy

In this section, we evaluate the prediction accuracy of the DML model extracted in our case study under different transaction rates. To evaluate the prediction accuracy for scaling decisions, we now use our extracted model and compare the predicted utilization and end-to-end response times at the transaction levels 20, 40, 60, 80, and 100 with measurements from corresponding benchmark runs at the real system. The transaction levels are chosen to cover a wide range of resource utilizations. A transaction level of 100 is close to the maximum load sustainable with the given configuration. Higher transaction rates require additional resources to ensure system stability.

In the previous section, we compared the fitting accuracy of different approaches to resource demand estimation for a transaction rate of 60. However, as a result of overfitting the ranking of estimation approaches may change with regards to their prediction accuracy for different transaction rates. Figure 7.4 shows the mean relative error of the predictions when using the estimated resource demands resulting from the considered estimation approaches. The mean relative error is calculated corresponding to the cross-validation error (using Equation 5.7 on page 149). However, we use the results obtained through a full simulation of the DML model as calculated values. We again used the

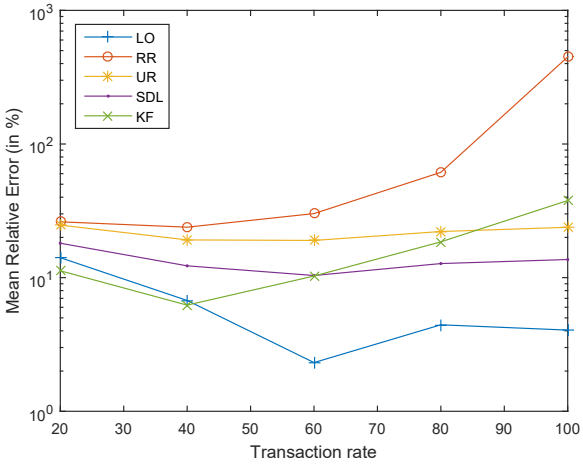


Figure 7.4: Mean relative error for different transaction rates.

model transformation to QPNs (Brosig, 2014) and the SimQPN even-discrete simulator (Kounev and Buchmann, 2006) to solve the DML model.

The results show that the ranking between estimation approaches with regards to their prediction accuracy changes with the transaction rate. At lower rates *KF* provides slightly better results than *LO*. However, the error of *KF* increases rapidly with a rate of 60 and above. At the highest rate, *KF* yields the second-worst results. The ranking of the other estimation approaches is consistent across the complete range. We conclude, that *LO* provides the best overall prediction accuracy over all transaction rates. This matches with the recommendation of the automatic approach selection of LibReDE.

The complete results are listed in Table 7.7 for the CPU utilization and in Table 7.8 for the end-to-end response time. The response times are shown for complete transactions consisting of multiple individual requests to the system. In summary, the extracted DML model yields a high prediction accuracy. When using *LO* approach to resource demand estimation as recommended by LibReDE, the absolute errors of the utilization are all below 4% and the relative errors of the end-to-end response times are less than 21%.

Impact of Interrupt Coalescing Modern operating systems and hypervisor implement different optimizations with the goal of reducing the number of hardware interrupts caused by network communication. In the most simple case, each incoming network packet triggers a hardware interrupt in the operating system or hypervisor in order to analyze the network packet and forward it to

Entity	Meas.	LO	RR	UR	SDL	KF
VM 2	20.2	17.3 (2.9 %)	18.0 (2.2 %)	17.6 (2.6 %)	17.7 (2.5 %)	17.7 (2.5 %)
VM 3	22.6	19.5 (3.1 %)	13.2 (9.4 %)	19.8 (2.8 %)	19.9 (2.7 %)	21.3 (1.3 %)
VM 4	8.4	7.2 (1.2 %)	2.4 (6.0 %)	7.2 (1.2 %)	7.2 (1.2 %)	7.2 (1.2 %)
VM 5	7.4	5.9 (1.5 %)	5.8 (1.6 %)	5.9 (1.5 %)	6.0 (1.4 %)	6.1 (1.3 %)
VM 6	1.2	0.7 (0.5 %)	0.3 (0.9 %)	0.6 (0.6 %)	0.6 (0.6 %)	0.7 (0.5 %)
VM 7	8.9	8.0 (0.9 %)	6.6 (2.3 %)	8.1 (0.8 %)	8.0 (0.9 %)	9.6 (0.7 %)
VM 9	11.9	11.1 (0.8 %)	21.1 (9.2 %)	11.2 (0.7 %)	11.1 (0.8 %)	12.6 (0.7 %)

(a) Transaction rate 20.

Entity	Meas.	LO	RR	UR	SDL	KF
VM 2	37.3	34.7 (2.6 %)	35.8 (1.5 %)	35.3 (2.0 %)	35.3 (2.0 %)	35.3 (2.0 %)
VM 3	42.9	39.2 (3.7 %)	26.2 (16.7 %)	39.7 (3.2 %)	39.7 (3.2 %)	42.2 (0.7 %)
VM 4	15.6	14.3 (1.3 %)	4.8 (10.8 %)	14.4 (1.2 %)	14.4 (1.2 %)	14.4 (1.2 %)
VM 5	12.2	11.9 (0.3 %)	11.5 (0.7 %)	12.0 (0.2 %)	11.9 (0.3 %)	12.2 (0.0 %)
VM 6	1.7	1.3 (0.4 %)	0.5 (1.2 %)	1.2 (0.5 %)	1.3 (0.4 %)	1.3 (0.4 %)
VM 7	16.6	16.0 (0.6 %)	13.2 (3.4 %)	16.1 (0.5 %)	16.3 (0.3 %)	18.9 (2.3 %)
VM 9	22.7	22.1 (0.6 %)	42.2 (19.5 %)	22.3 (0.4 %)	22.4 (0.3 %)	24.9 (2.2 %)

(b) Transaction rate 40.

Entity	Meas.	LO	RR	UR	SDL	KF
VM 2	53.4	52.0 (1.4 %)	53.6 (0.2 %)	52.9 (0.5 %)	52.9 (0.5 %)	52.9 (0.5 %)
VM 3	60.2	58.6 (1.6 %)	39.3 (20.9 %)	59.5 (0.7 %)	59.3 (0.9 %)	63.1 (2.9 %)
VM 4	21.8	21.5 (0.3 %)	7.2 (14.6 %)	21.6 (0.2 %)	21.5 (0.3 %)	21.6 (0.2 %)
VM 5	18.0	18.0 (0.0 %)	17.2 (0.8 %)	17.9 (0.1 %)	17.9 (0.1 %)	18.3 (0.3 %)
VM 6	2.2	2.0 (0.2 %)	0.8 (1.4 %)	1.9 (0.3 %)	1.9 (0.3 %)	1.9 (0.3 %)
VM 7	24.4	24.1 (0.3 %)	19.6 (4.8 %)	24.4 (0.0 %)	24.4 (0.0 %)	28.3 (3.9 %)
VM 9	33.4	33.2 (0.2 %)	63.1 (29.7 %)	33.5 (0.1 %)	33.5 (0.1 %)	37.4 (4.0 %)

(c) Transaction rate 60.

Entity	Meas.	LO	RR	UR	SDL	KF
VM 2	69.2	69.1 (0.1 %)	71.4 (2.2 %)	70.2 (1.0 %)	70.4 (1.2 %)	70.2 (1.0 %)
VM 3	76.1	78.0 (1.9 %)	52.2 (23.9 %)	78.9 (2.8 %)	79.0 (2.9 %)	83.8 (7.7 %)
VM 4	28.9	28.5 (0.4 %)	9.5 (19.4 %)	28.7 (0.2 %)	28.7 (0.2 %)	28.8 (0.1 %)
VM 5	24.4	23.8 (0.6 %)	22.9 (1.5 %)	23.7 (0.7 %)	23.8 (0.6 %)	24.3 (0.1 %)
VM 6	2.7	2.6 (0.1 %)	1.1 (1.6 %)	2.5 (0.2 %)	2.6 (0.1 %)	2.6 (0.1 %)
VM 7	32.5	32.1 (0.4 %)	26.1 (6.4 %)	32.3 (0.2 %)	32.7 (0.2 %)	38.1 (5.6 %)
VM 9	44.4	44.2 (0.2 %)	83.9 (39.5 %)	44.7 (0.3 %)	44.6 (0.2 %)	50.0 (5.6 %)

(d) Transaction rate 80.

Entity	Meas.	LO	RR	UR	SDL	KF
VM 2	81.2	84.9 (3.7 %)	85.7 (4.5 %)	85.9 (4.7 %)	86.4 (5.2 %)	84.4 (3.2 %)
VM 3	90.9	95.5 (4.6 %)	62.7 (28.2 %)	96.2 (5.3 %)	96.2 (5.3 %)	99.3 (8.4 %)
VM 4	34.2	35.3 (1.1 %)	11.4 (22.8 %)	35.5 (1.3 %)	35.5 (1.3 %)	35.3 (1.1 %)
VM 5	29.3	29.6 (0.3 %)	27.5 (1.8 %)	29.5 (0.2 %)	29.5 (0.2 %)	29.9 (0.6 %)
VM 6	3.2	3.2 (0.0 %)	1.3 (1.9 %)	3.1 (0.1 %)	3.2 (0.0 %)	3.1 (0.1 %)
VM 7	40.0	40.2 (0.2 %)	31.0 (9.0 %)	40.3 (0.3 %)	40.8 (0.8 %)	47.1 (7.1 %)
VM 9	54.2	55.1 (0.9 %)	99.7 (45.5 %)	55.1 (0.9 %)	55.2 (1.0 %)	61.2 (7.0 %)

(e) Transaction rate 100.

Table 7.7: Utilization predictions (absolute error in brackets).

Entity	Meas.	LO	RR	UR	SDL	KF
Purchase	36.7	29.1 (20.7%)	31.7 (13.6%)	31.1 (15.3%)	39.6 (7.9%)	31.2 (15.0%)
Manage	30.0	26.6 (11.3%)	27.9 (7.0%)	20.8 (30.7%)	24.6 (18.0%)	27.6 (8.0%)
Browse	49.3	42.0 (14.8%)	45.2 (8.3%)	47.8 (3.0%)	44.3 (10.1%)	46.5 (5.7%)
Mfg EJB	17.3	16.6 (4.0%)	17.3 (0.0%)	27.5 (59.0%)	12.0 (30.6%)	18.1 (4.6%)
Mfg WS	17.2	16.8 (2.3%)	17.4 (1.2%)	5.5 (68.0%)	12.1 (29.7%)	18.3 (6.4%)

(a) Transaction rate 20.

Entity	Meas.	LO	RR	UR	SDL	KF
Purchase	38.3	36.2 (5.5%)	37.6 (1.8%)	39.4 (2.9%)	46.9 (22.5%)	38.6 (0.8%)
Manage	31.0	29.6 (4.5%)	32.6 (5.2%)	23.0 (25.8%)	28.3 (8.7%)	32.4 (4.5%)
Browse	57.6	54.1 (6.1%)	55.0 (4.5%)	62.5 (8.5%)	56.5 (1.9%)	60.0 (4.2%)
Mfg EJB	18.6	17.5 (5.9%)	20.1 (8.1%)	29.0 (55.9%)	12.6 (32.3%)	19.3 (3.8%)
Mfg WS	18.7	17.8 (4.8%)	20.3 (8.6%)	5.8 (69.0%)	12.7 (32.1%)	19.5 (4.3%)

(b) Transaction rate 40.

Entity	Meas.	LO	RR	UR	SDL	KF
Purchase	44.3	46.0 (3.8%)	48.8 (10.2%)	51.1 (15.3%)	58.7 (32.5%)	50.8 (14.7%)
Manage	34.0	33.8 (0.6%)	42.9 (26.2%)	25.0 (26.5%)	33.4 (1.8%)	38.5 (13.2%)
Browse	73.2	75.4 (3.0%)	73.5 (0.4%)	84.9 (16.0%)	77.3 (5.6%)	86.0 (17.5%)
Mfg EJB	19.3	19.0 (1.6%)	27.0 (39.9%)	31.0 (60.6%)	13.6 (29.5%)	21.2 (9.8%)
Mfg WS	19.5	19.3 (1.0%)	27.1 (39.0%)	6.2 (68.2%)	13.7 (29.7%)	21.3 (9.2%)

(c) Transaction rate 60.

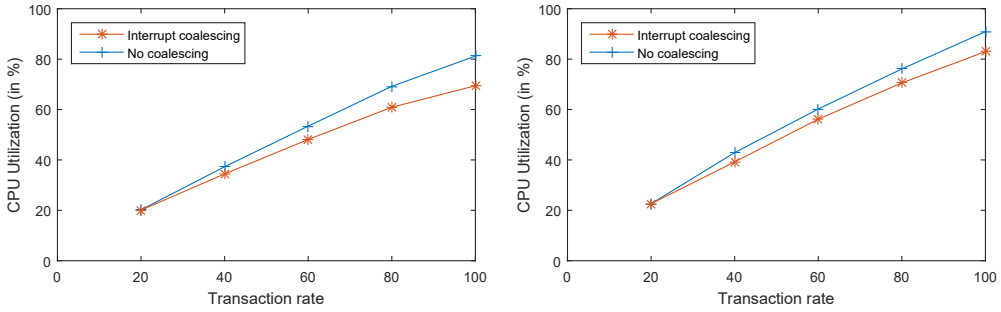
Entity	Meas.	LO	RR	UR	SDL	KF
Purchase	59.5	68.0 (14.3%)	82.0 (37.8%)	78.0 (31.1%)	83.6 (40.5%)	81.8 (37.5%)
Manage	38.5	39.8 (3.4%)	78.5 (103.9%)	27.7 (28.1%)	42.9 (11.4%)	49.5 (28.6%)
Browse	112.2	126.3 (12.6%)	125.8 (12.1%)	140.2 (25.0%)	130.7 (16.5%)	162.1 (44.5%)
Mfg EJB	20.8	21.0 (1.0%)	53.2 (155.8%)	34.3 (64.9%)	15.1 (27.4%)	24.3 (16.8%)
Mfg WS	21.0	21.3 (1.4%)	53.2 (153.3%)	6.7 (68.1%)	15.2 (27.6%)	24.4 (16.2%)

(d) Transaction rate 80.

Entity	Meas.	LO	RR	UR	SDL	KF
Purchase	144.9	157.1 (8.4%)	420.2 (190.0%)	192.6 (32.9%)	194.3 (34.1%)	285.8 (97.2%)
Manage	57.0	53.8 (5.6%)	561.0 (884.2%)	30.8 (46.0%)	68.9 (20.9%)	73.6 (29.1%)
Browse	342.3	364.4 (6.5%)	609.0 (77.9%)	424.5 (24.0%)	410.8 (20.0%)	771.2 (125.3%)
Mfg EJB	23.5	24.5 (4.3%)	446.0 (1797.9%)	38.5 (63.8%)	17.5 (25.5%)	29.1 (23.8%)
Mfg WS	23.6	24.6 (4.2%)	443.5 (1779.2%)	7.4 (68.6%)	17.6 (25.4%)	29.2 (23.7%)

(e) Transaction rate 100.

Table 7.8: Response time predictions (relative error in brackets).



(a) VM 2. **(b)** VM 3.

Figure 7.5: Impact of interrupt coalescing.

the receiving process or VM. High network traffic therefore can cause a high interrupt load on a machine resulting in additional overheads due to context switches.

Interrupt coalescing is an optimization that aims at processing multiple network packets with a single hardware interrupt. Incoming network packets are enqueued and a hardware interrupt is triggered if either the queue is full or after a certain timeout. For instance, the VMware ESXi 5.5 hypervisor by default queues up to 64 network packets and has a timeout of 4 milliseconds (VMware, Inc., 2014). In addition, VMware states that "other events, such as the virtual machine being idle, can also trigger virtual machine interrupts or packet transmission, so packets are rarely delayed the full 4 milliseconds" (VMware, Inc., 2014, page 38).

Interrupt coalescing can be turned off in the VMware ESXi hypervisor by setting the `ethernetX.coalescingScheme` parameter to `disabled`. In our distributed setup of SPECjEnterprise2010, we ran experiments with or without interrupt coalescing and at different transaction rates. We then compared the CPU utilization of each VM to analyze the impact of interrupt coalescing.

Figure 7.5 shows the CPU utilization of the VM 2 (see Figure 7.5a) and VM 3 (see Figure 7.5b). These are the VMs with the highest CPU utilization in the experiments. While the transaction rate increases linearly, the CPU utilization exhibits a non-linear curve, especially when interrupt coalescing is activated in VMware ESXi hypervisor. The results show that the impact of interrupt coalescing increases with higher CPU utilizations While the CPU utilization is almost equal at a transaction rate $tx = 20$, there are significant differences when $tx = 100$: The CPU utilization of VM 2 is 11.78%, and the one of VM 3 is

7.93% higher when interrupt coalescing is deactivated.

We explain this distinct load-dependent impact of interrupt coalescing with its timeout setting. At lower transaction rates the timeout triggers more often before the queue of network packets is filled up resulting in less network packets being processed per hardware interrupt. In consequence, the relative overhead due to interrupt handling is higher at lower utilizations.

The current version of DML does not provide means to describe and solve models with load-dependent resource demands. Kumar et al. (2009b) describe a technique to estimate such load-dependent resource demands assuming the functional form of the dependency as an input. LibReDE can be extended for such scenarios by applying transformations to the state model (e.g., logarithm or square root transformations) as it is often done with regression techniques. This is left for future work as it requires also corresponding modeling and solution support in DML which is out of scope of this thesis.

7.4.4 Discussion

In this case study, we evaluated the degree of automation achieved for the model learning step as well as the fitting and prediction accuracy of the resulting performance models. For the SPECjEnterprise2010 application, we achieved a degree of automation of 85.5%. By integrating existing techniques for usage profile extraction, this number could be increased to 97.4%. As a result, a system administrator only needs to provide high-level information on the control flow between applications to obtain a complete model. The internal architecture of the application could be completely extracted including all platform layers. It is noteworthy, that the model learning capabilities in the Wildfly VAs only assume that the application adheres to the Java EE standard and does not include any prior knowledge of the SPECjEnterprise2010 application. Therefore, it can be reused to create performance models of other Java EE applications. For instance, Bauer (2016) has already used our Wildfly VA with a different application.

We compared the fitting accuracy achieved with different approaches to resource demand estimation. LibReDE recommends the optimization-based approach using end-to-end response times (see Liu et al., 2006) that can provide the best results with a total cross-validation error of 1.7%. We then use the fully parameterized model to predict the performance for scaling scenarios. When using the resource demand estimation approach as recommended by LibReDE, we obtained model predictions with an absolute error of less than 4% for the CPU utilization and a relative error of less than 21% for the end-to-end response time. Given the SPECjEnterprise2010 deployment with seven different VMs and 80 different resource demands to be estimated, it poses a considerable

problem size. LibReDE is able to solve the problem in less than 10 seconds demonstrating the feasibility to continuously update the resource demands in an online manner.

7.5 Case Study: Zimbra Collaboration Server

This case study is based on the Zimbra VA described in Section 7.2.2. The goal of this case study is to evaluate the effectiveness and the resource efficiency of the controllers proposed in Chapter 6. We consider the vertical scaling of virtual CPUs and memory of VMs.

7.5.1 Experiment Setup

We use a two server setup of Zimbra, where the mailbox server (including the LDAP server) is deployed in one VM and the MTA in another one. Users directly interact with the mailbox server having stringent requirements regarding the response times. The mailbox server and the MTA only communicate with each other asynchronously, i.e., the processing time at the MTA is excluded in the end-to-end response time experienced by a user. However, in order to ensure reliable and fast delivery of mails to a recipient, the MTA is also subject to goals with regards to the throughput and the maximum time for delivery. In the following, we assume a maximum acceptable response time for requests to the mailbox server of one second and a maximum delivery time for mails of two minutes.

Workload We use an adapted version of a load driver used by the Zimbra developers to generate our workloads. The driver simulates a session-based, closed workload for a configured number of users. A session consists of several tasks sending requests to the server. Tasks represent atomic user actions (e.g., reading, writing, moving, and deleting mails). The sleep times between the tasks of a session are exponentially distributed. We modified the driver to support dynamic workloads so that it can vary the number of concurrent sessions of users over time according to a given time series. Each session randomly chooses a mailbox on the server from a uniform distribution.

The mailbox server contained 2500 mailboxes, each with 10MB mail content. Each mailbox contains approximately 5000 messages with content and different types of attachments from a dump of a set of mailboxes from a production mail server. The number of concurrent sessions is dynamically varied over the duration of an experiment. Given that we could not obtain suitable arrival

traces from a production instance, we extracted the load intensities from the FIFA'98 traces (Arlitt, 2000) as a time series and scaled it to match the capacity of our system.

Deployment Our experiment setup consists of three physical hosts: ESX1, ESX2, and ESX3. The physical hosts are connected by a 1 Gbit/s network. ESX1 runs a vCenter Server instance, the Zimbra load generator, and our controller. ESX2 runs a VM with the mailbox server and ESX3 runs one with the MTA. We used two different resource environments for our experiments:

- *Setup 1*: Each physical host has two 2.6 GHz Intel Xeon E5430 processors with 4 cores, 32 GB RAM, and a 150 GB HDD. The hosts run a VSphere 5.1 hypervisor; the VMs are allocated 4 GB RAM and initially 1 CPU. They run Linux CentOS 6.4 as operating system with a Zimbra 8.0.5 server.
- *Setup 2*: Each physical host is equipped with a 3.2 GHz Intel Xeon E3-1230 CPU with 4 cores, 16 GB RAM and a 500 GB HDD. The hosts run a VSphere 5.5 hypervisor; the VMs are configured with 2 CPUs and 4 GB RAM. Inside is running a CentOS 7.0 64-bit OS and a Zimbra 8.5 server.

The MTA is not connected to the Internet so that only mails from the local mailbox server are processed. In *Setup 2*, we adapted the Zimbra database configuration to store and reload the MySQL buffer pool when restarting in order to reduce cache warm up times. We use *Setup 2* only in Section 7.5.5, and *Setup 1* otherwise.

7.5.2 Application Scalability

First experiments with Zimbra showed that in our experiment setup the scalability of the mailbox server is I/O-bound whereas the MTA is CPU-bound. Each incoming mail at the mailbox server needs to be persisted to the hard disk. Furthermore, each user request may trigger queries in the MySQL database running on the mailbox server potentially leading to even more hard drive accesses. The MTA is on the other hand CPU-bound as the anti-virus and spam checks performed for each mail requires considerable CPU time.

Limits on the Scaling of virtual CPUs In this experiment, we evaluate the scalability of the application with an increasing workload and number of virtual CPUs. The workload consists of a fixed number of 500 active users. The session intensity is step-wise increased every hour starting from 2.5 up to 10 sessions per user and per hour. The complete experiment ran over 9 hours. The mailbox

server is initially configured with 2 CPUs and the utilization of both cores is below 30%. The bottleneck is the MTA server which is automatically scaled from 1 to 6 CPUs by our model-based controller in response to the increasing workload.

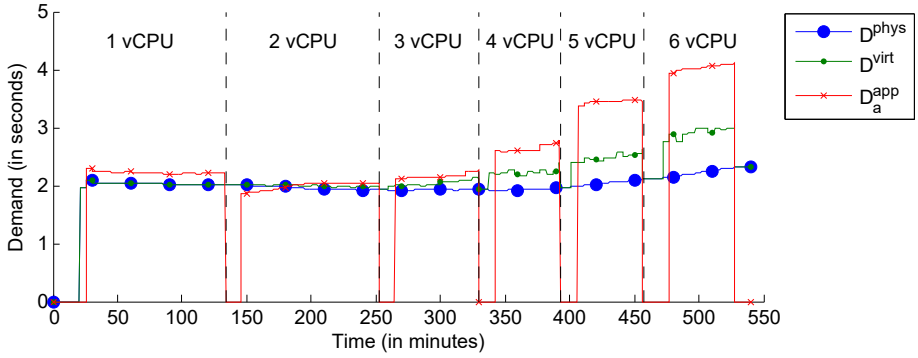


Figure 7.6: Model estimates with increasing workload.

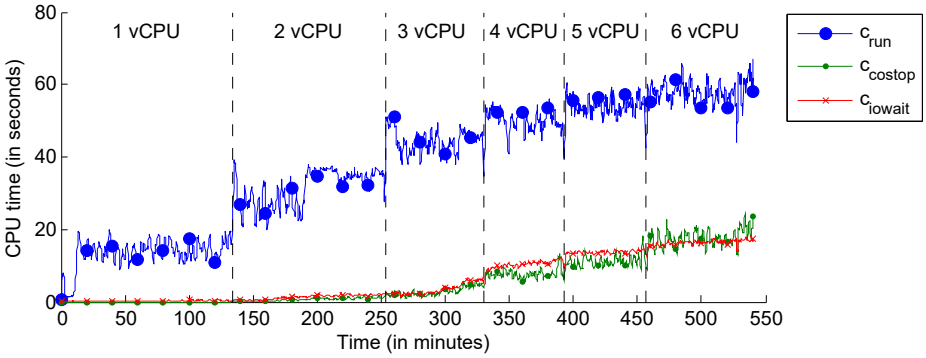


Figure 7.7: Hypervisor scheduling statics during increasing workload.

Fig. 7.6 shows the estimated physical resource demand D^{phys}, the virtual resource demand D^{virt}, and the application demand D_a^{app} depending on the number of CPUs a. D^{phys} is relatively constant and shows only a slight load-dependency with 5 and 6 CPUs (rising from approximately 2 to 2.3 seconds). This load-dependency is not explicitly captured in the model. We rely on the model estimator to adapt to these changes in the physical resource demand.

The virtual resource demand D^{virt} approximately matches the physical resource demand for one and two CPUs, however, it increases significantly for higher numbers of CPUs. This increase can be explained with the CPU scheduling by the ESX hypervisor Fig. 7.7 shows the scheduling statistics as reported

by ESX for the MTA VM. The statistics are reported every 20 seconds and show total CPU time during which the VM was in a certain scheduling state. The increase of D^{virt} is caused by delays at the hypervisor due to co-scheduling effects. Although there are no additional VMs contending for the physical resources with the MTA server, its co-stop time increases. With increasing workload the I/O wait time as reported by the hypervisor also increases. We explain the co-stop time with the observed I/O wait: due to individual virtual CPUs which are delayed by I/O, other CPUs are slowed down by the hypervisor such that the CPU time of the different CPUs of a VM do not diverge.

However, the hypervisor co-scheduling overhead is not the only reason for the limited scalability of the MTA. Figure 7.7 also shows that, the used CPU time (reported by the c_{run} statistic) stagnates with the increasing number of CPUs. With 6 virtual CPUs the application could get 120 seconds CPU time in each 20 second interval. However, it is only able to use about 60 seconds of that. To mitigate the bottleneck, a reconfiguration of the application would be required (e.g., changing the number of processing threads in the MTA).

Influence of Memory Size In this experiment, we evaluate the potential for the scaling of the memory size during system operation. In the previous experiment, we focused on the cpu-bound MTA server. When varying different workload parameters, the MTA yielded a constant memory allocation.

In contrast, the mailbox server is I/O-bound and uses different application caches to reduce the number of hard disk accesses. The following three major caches were identified: (a) the mailbox cache keeping the meta data of the mails, contacts, calenders, etc., of a user in memory, (b) the message blob cache retaining the content of last read mails in memory, and (c) the buffer pool of the underlying MySQL database. Table 7.9 shows the application performance of the mailbox server with 4 GB and 8 GB of main memory when the number of actively used mailboxes is increased to from 500 to 2000. We conclude that the buffer pool hit rate has a significant influence on the application performance.

	Mean latency (in ms)	Std latency	Buffer pool hit rate
4GB	106	55	95.6%
8GB	50	26	99%

Table 7.9: Impact of mailbox server memory configuration.

However, when adding main memory dynamically to the VM, it turned out that the mailbox server cannot benefit from the additional memory directly. The reason is that the buffer pool of the MySQL database is allocated statically and

cannot be changed at runtime. The same applies to the Jetty application server of the mailbox server which is a Java-based application with a maximum heap size. While the buffer pool size and the Java heap size can be over-provisioned at application startup to accommodate a future memory scale-up, this is not an option in practice, because if these sizes are larger than the physical memory size, it can result in heavy swapping at the operating system layer. In order to leverage dynamic changes to the memory configuration of a VM, additional application support would be required.

7.5.3 Physical Resource Contention

In order to evaluate the impact of physical resource contention on the model estimation, we added additional load VMs to the host where the MTA VM (configured with 2 CPUs) is running. We used 8 load VMs with one CPU each running a micro-benchmark calculating Fibonacci numbers. These load VMs demand all the physical CPU resources of the host. As a result, the VM running the MTA server is constantly competing for the physical CPU resources and the processing of mails is therefore slowed down in the MTA.

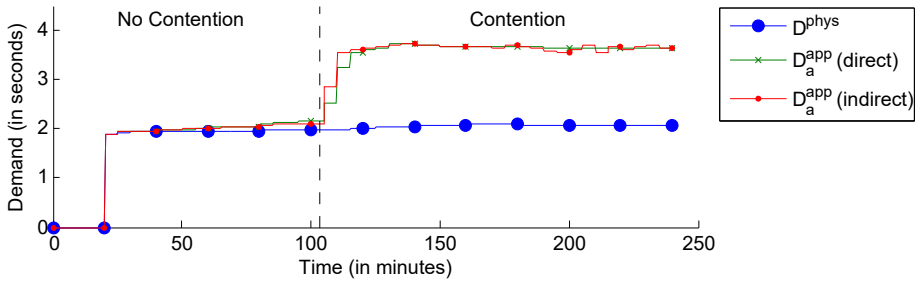


Figure 7.8: Demand estimates under physical resource contention.

Fig. 7.8 shows the estimated physical demand D^{phys} and the application demand D_a^{app} depending on the number of vCPUs a estimated using two different approaches. The *indirect* approach is based on a least-squares regression using the following equation to estimate the application demand based on the virtual resource demand determined using the observed hypervisor scheduling statistics:

$$R_{s,c} - D_{v,cpu}^{virt}(1 + \frac{Q_{v,a}}{a}B_{v,a}) = D_{v,a}^{other}(1 + \frac{Q_{v,a}}{a}B_{v,a}). \tag{7.1}$$

For comparison, we also estimated application demand *directly* based on the residence time equation (see Equation 6.2 on page 158) performing a linear

regression on the observed mean latency and the average queue length on arrival.

In the first part of the experiment, the micro-benchmarks in all load VMs are not running, i.e., the VM running the MTA receives all requested resources. When the CPU-heavy computation begins in the load VMs after approximately 100 minutes, the VM running the MTA experiences physical resource contention. The application processing rate is slowed down, and the application demand increases as one would expect. In contrast, the physical resource demand (i.e., the CPU time on the physical CPU to process one request) is not influenced by the physical resource contention. The comparison between the directly and indirectly estimated application demands show that the difference between both is negligible. See Table 7.10 for the exact values. This experiments shows that the scheduling statistics reported by the ESX hypervisor can be used to estimate the requests in the application are delayed to the contention effects at the virtual resource layer.

	Direct	Indirect	Relative Error
No contention	2.16	2.11	1.96%
Contention	3.65	3.65	<1%

Table 7.10: Comparison of direct to indirect estimation of D_a^{app} for a given vCPU configuration.

7.5.4 Short-term CPU Scaling

To evaluate the behavior of the resource controller under a dynamic workload, we ran a workload with a typical pattern for an application in the course of a week. We used the access logs from the FIFA'98 world cup web servers (Arlitt, 2000) and extracted the session intensities of a complete week (06/01-06/06/1998). We scaled the session intensities down to adjust the workload to the computing capacity of our system. Fig. 7.9 shows the workload at the Mailbox server and at the MTA. Only a fraction of the requests to the Mailbox server result in a mail being sent through the MTA. We can observe a workload pattern typical for many real-world applications, where the demand during the day is significantly higher than in the night. Additionally, there are differences in the workload between weekdays and weekends. This results in varying resource requirements of the application in the course of a week.

We refer to the model-based controller described in Section 6.1.4 as the *demand controller*, and compare it with a threshold-based *utilization controller*

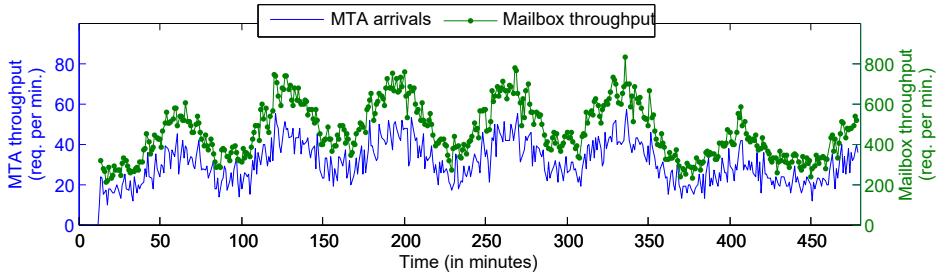


Figure 7.9: Dynamic workload used for the experiment.

and a static allocation of one vCPU. We used two variations of the utilization controller with different control intervals. The utilization controller checks the CPU utilization of the MTA VM every minute (or every 5 minutes). If the average utilization of the control interval is above 90%, it adds an additional vCPU to the VM; if the current CPU usage is below 40%, it removes one CPU.

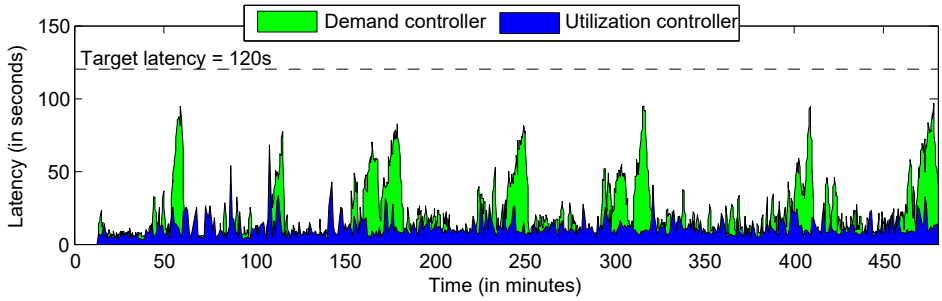


Figure 7.10: Comparison of the MTA latency between the demand controller and the utilization controller.

Fig. 7.10 shows the observed latency of the MTA with the demand controller and the utilization controller (1 minute control interval). The latency of the MTA is the time from receiving a mail until it is delivered to the recipient's mailbox server. For the MTA we chose a target latency of two minutes. When statically allocating a single CPU, the server gets overloaded during load spikes, so the mails queue up and we observe maximum latencies of over 45 minutes (see Table 7.11). Both controllers can avoid the overload situation by adding additional CPUs during phases of high workloads. We conclude that both controllers can effectively maintain the mail delivery latency below the target value, therefore fulfilling the application SLO.

On the other hand, Fig. 7.11 shows that the demand controller is significantly

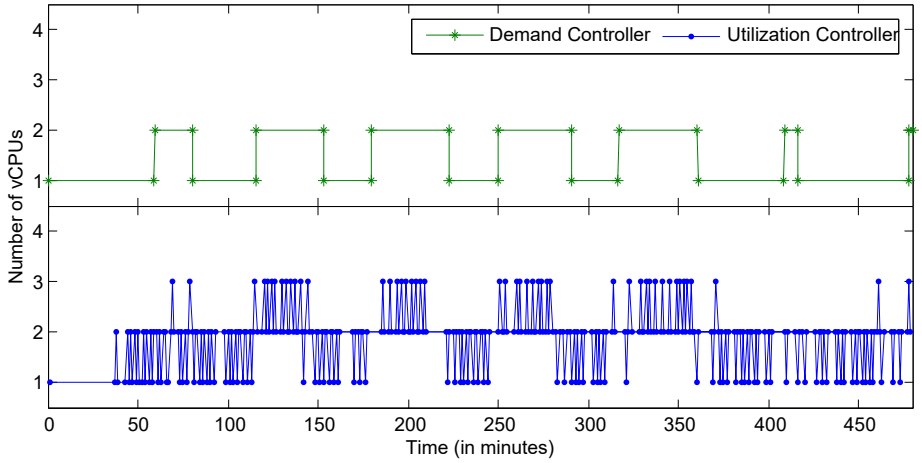


Figure 7.11: Number of reconfigurations with the demand and the utilization controller (1 minute control interval).

more stable than the tested variants of the utilization controller. The utilization controller exhibits an oscillating behavior with 273 (or 72) reconfigurations with the one (or five) minute control interval compared to the 13 reconfigurations of the demand controller. At the same time the total CPU allocation of the demand controller is lower compared to those of the utilization controllers. Even though additional optimizations of the control interval and the thresholds of the utilization controller may improve its efficiency and stability further, the optimal setting for the utilization controller heavily depends on the application and its workload, and it may change as the application state and the execution environment evolve. In contrast, our demand controller only requires the target application latency, which is usually available for a business critical application.

	Latency [s]		Reconfigurations	CPUs	
	mean	max		mean	max
Demand controller	20.48	95.99	13	1.4	2
Util. controller (1 min)	10.82	67.86	273	1.83	3
Util. controller (5 min)	25.97	92.1	72	1.46	3
Static allocation	1385	2842	0	1	1

Table 7.11: Comparison of controller performance.

7.5.5 Mid- to Long-term Memory Scaling

In this section, we evaluate how our proactive controller (see Section 6.2) can help to reduce the impact of the reconfiguration on the application availability and performance by comparing it with a reactive approach. The results of the following experiments were published in Spinner et al. (2015b).

Proactive vs. Reactive Controllers We compare our proactive controller to a threshold-based, reactive controller. In addition, we performed one baseline experiment without vertical memory scaling.

As the application does not directly support the observation of the incoming load, we use the monitored throughput as an approximation for the number of arriving requests, and learn the forecasting model based on the throughput. Given that we aggregate the collected statistics over a complete hour, we argue that this is a safe approximation. The proactive controller is configured to do a forecast every night at 3 AM when a minimum load on the system is expected. It predicts the number of requests for the next day resulting in 24 hourly arrival rates of which it takes the maximum. As thresholds, we use the maximum sustainable throughput of the system for a given memory size. We determined these thresholds in an offline profiling experiment. The controller reconfigures the memory in 4096 MB steps.

The reactive controller monitors the availability and response time of the mailbox server. It is triggered if the server is unavailable or if the observed average response times are above one second for over a period of three minutes. The controller has a quiet time of one hour, i.e., after a memory reconfiguration the trigger will not fire again in this period.

Impact of reconfiguration In this experiment, we evaluate the impact of memory scaling on the availability and performance of the application. We used *Setup 2* described in Section 7.5.1 for the experiments. The FIFA'98 trace covers a period of 92 days making it infeasible to run an experiment over the complete length. Therefore, we extracted a subset of four days from the trace, scaled its length to an experiment length of 16 hours, and used it as an input to our load generator. The subset covers the period from Saturday, June 30th 3:00 AM to Wednesday, July 4th, 3:00 AM. Figure 7.12 shows the resulting number of active users over the experiment length. It is expected that the initially configured memory size of 4 GB is insufficient to serve the workload on Monday and Tuesday.

In order to reduce the experiment duration, we used four weeks of historic data from the FIFA'98 trace to learn the forecast model. As the experiment begins, the proactive controller automatically switches to the live monitoring data

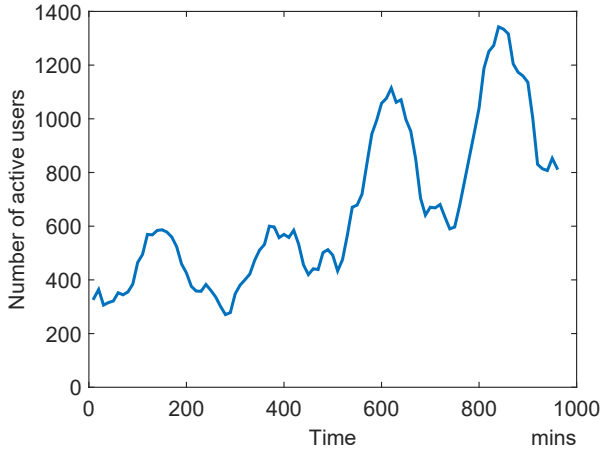


Figure 7.12: Number of active users in a four-day workload trace.

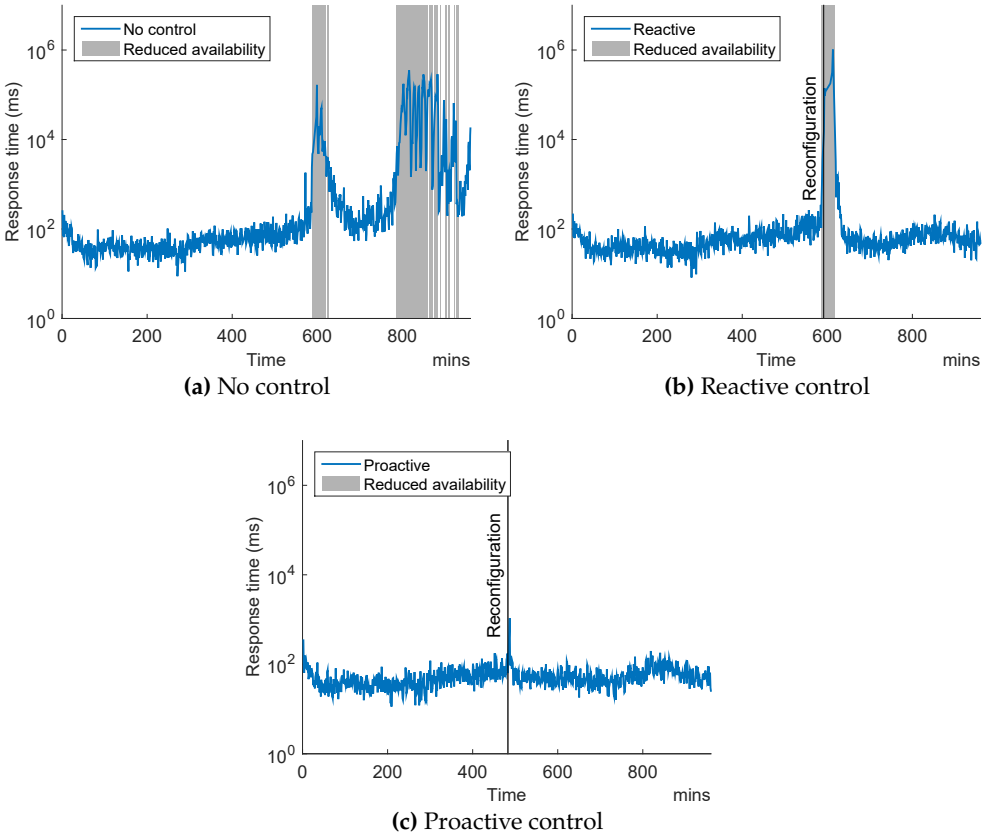
from the Zimbra server. The live data is continuously appended to the historic data and the forecast model is updated according to the new observations.

In the following, we define the term “availability" as the number successful requests divided by the total number of requests during a time interval. If the availability is below 100%, we consider it *reduced availability*. Here the success of each request is measured from the client perspective, i.e., as observed by the load generator. If a request times out due to an overloaded application, we consider the request unsuccessful.

	No control	Reactive	Proactive
Mean response time	7,567 ms	1,211 ms	52 ms
Maximum response time	349,830 ms	1,023,100 ms	1,077 ms
Timeouts	84	285	0
Errors	8493	1485	337
Time of reduced availability	176 min	33 min	4 min

Table 7.12: Comparison of controllers.

Figure 7.13 shows the observed average response time for three different experiments including the time of reduced availability during which not all requests can be successfully served. In the first experiment in Figure 7.13a, no additional memory is added to the VM. On the third and forth day, the application is overloaded as the memory becomes a bottleneck resulting in high read load on the hard disk. The response times of the application therefore



(a) No control **(b)** Reactive control

(c) Proactive control

Figure 7.13: Observed response times of the mailbox server.

increase significantly (see also Table 7.12). During peak periods the application is not able to serve all requests. In total, the availability is below 100% for a period of 176 minutes due to timeouts and connection errors (see Table 7.12).

The reactive controller in Figure 7.13b is triggered by the unavailability of the application as the response times increase too abruptly for the controller to react. As the application is overloaded, the steps described in Section 6.2.2.2 take a long time to complete and afterwards the application also needs to reload its caches under a high workload causing additional overhead. In total, the application is only partially available over a period of 33 minutes due to the overload situation and the reconfiguration (see Table 7.12). After that period, the reconfiguration effectively mitigated the memory bottleneck and the application is able to serve the workload with an acceptable performance

again.

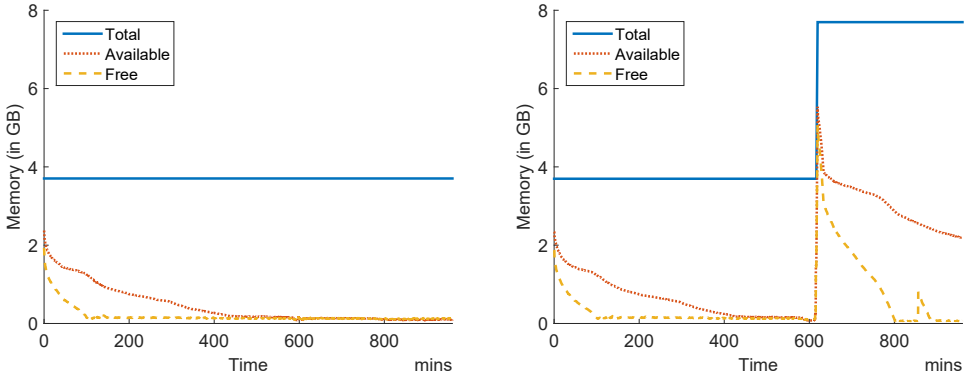
The results from the proactive controller are shown in Figure 7.13c. The proactive controller correctly detects the future memory bottleneck in the night between the second and third day and proactively triggers the reconfiguration during a phase of low load. This results in a much lower impact of the reconfiguration on the availability and performance of the application. Given that the application needs to be restarted, it is unavailable for a period of 4 minutes (see Table 7.12). After this period, it reloads its caches and serves user requests in parallel without overloading the VM. Compared to the reactive controller, our approach reduces both the time of lower availability and the number of errors during the reconfiguration by more than 80%. We conclude that using a proactive control approach to memory scaling can effectively reduce the impact of the required reconfigurations on the application.

Memory usage Figure 7.14 shows the observed memory usage of the application as reported by the guest operating system in the VM. The free counter reports the unallocated memory and corresponds to the memory usage as seen by the hypervisor. The available counter does not include the memory reserved for the operating system buffer caches. As the Linux operating system greedily allocates memory for its buffer caches, the free counter does not reflect the actual memory demand of the VM. The available counter is a better indicator of the memory pressure within the VM. However, it does not show a clear correlation to the workload intensity. Therefore, it is necessary in our approach to benchmark the application in advance to determine the maximum number of session that can be served with a given memory size.

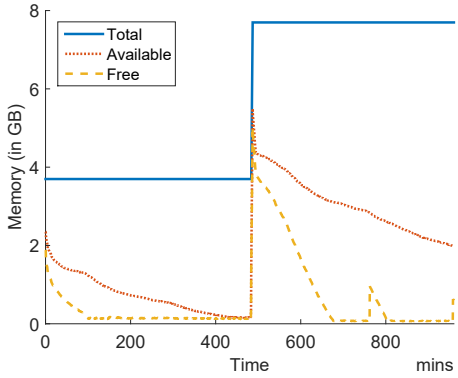
7.5.5.1 Allocation decisions

Using the same thresholds as in the reconfiguration experiments, we also analyzed the allocation decisions of the proactive controller for the complete first 5 weeks of the FIFA'98 trace. Excluding the training phase of the forecaster, this results in 30 days for which the proactive controller can predict the required memory size. We compare the memory allocation resulting from the forecast arrival rate to the memory allocation that would be required for the actual arrival rate. As the memory allocation can only be a multiple of a certain step size in our approach, we calculated the over- and under-provisioning ratios using step sizes of 1024 MB, 2048 MB, and 4096 MB. The results are shown in Table 7.13.

In summary, the proactive controller using our splitTs method (see Section 6.2) results in a lower chance of over- and under-provisioning ($< 11\%$)



(a) No control **(b)** Reactive control



(c) Proactive control

Figure 7.14: Observed memory usage of the mailbox server VM.

compared to the WCF method (Herbst et al., 2014). However, both methods tend to underestimate the resource allocation on the FIFA'98 trace. This is also visible in Figure 6.4. Although splitTs correctly captures the long-term increasing trend of the trace, more sophisticated methods to extract overlapping trends and seasonal patterns may improve the results. This will be part of our future work on combining splitTs with the load intensity modeling framework DLIM (Kistowski et al., 2015). In order to obtain more conservative forecasts and reduce under-provisioning, we recommend to use the upper confidence level of the forecast. Compared to a constant factor as proposed in (Shen et al., 2011), the confidence interval has the advantage that its width also depends on the fitting quality of the forecast model.

	Step size	Overprovisioned		Underprovisioned	
		Days	Amount	Days	Amount
splitTs	1024 MB	3	1.08%	14	10.79%
	2048 MB	0	0%	9	10.07%
	4096 MB	0	0%	6	8.54%
WCF	1024 MB	6	3.6%	18	14.03%
	2048 MB	4	3.36%	14	13.42%
	4096 MB	2	2.44%	8	10.98%

Table 7.13: Allocation decisions of the proactive controller using splitTs and the WCF forecasters.

7.5.6 Discussion

The short-term model-adaptive controller for vertical CPU scaling reduced the number of allocated virtual CPUs by up to 23% compared to two variants of a trigger-based controller. The latter requires the manual specification of scaling rules including thresholds. These rules typically depend on the architecture and current state of the application. In contrast, our controller can directly use the performance model learned automatically at system run-time to adapt the number of virtual CPUs in accordance with application-level SLOs with regards to response time and throughput. The performance model also provides deeper insights into the performance behavior of an application for a more fine-grained detection of bottlenecks (e.g., also considering contention effects). In our scenario, trigger-based approaches did not consider the current state of the application resulting in many short-term reconfigurations. Our model-adaptive controller was able to incorporate the current application state (i.e., the work queued at the mail transfer agent) into its scaling decisions, resulting in a reduction of reconfigurations by up to 95%.

Our proactive controller could reduce the impact of memory reconfigurations on the application availability and performance of the application by more than 80% compared to a reactive controller. Using our splitTs forecaster, we were also able to reduce the over- and under-provisioning of memory compared to other state-of-the-art forecasting techniques to under 11%.

Chapter 8

Conclusions

This chapter concludes the thesis and provides a summary of its contributions. We discuss the benefits of our work and give an overview of potential future work.

8.1 Summary

In this thesis, we proposed an approach to self-aware resource management in virtualized data centers. Self-aware means that it is based on models that are learned automatically at system run-time and that are used to reason about the system in order to fulfill high-level goals. In the context of virtualized data centers, the high-level goals are the fulfillment of application SLAs with regards to performance and availability objectives while continuously improving resource efficiency. Our approach is based on an existing meta-model – called DML (Huber et al., 2017) – to represent the knowledge about the system performance behavior. We made contributions in the following two areas: *model learning* (covering model structure and parameterization), and *model reasoning* (using models for vertical scaling of applications).

Reference Architecture for Online Model Learning in Virtualized Systems

The reference architecture lays the foundation to integrate different model learning techniques into virtualization platforms in order to obtain end-to-end performance models of the complete system. We proposed to bundle the logic for online model learning into VAs that can be then be shared with others (e.g., through online marketplaces) increasing reuse of model learning logic. When such a VA is deployed in a virtualized environment, the model learning logic will start to monitor the system under real production workloads and will automatically built a performance model of the system. A performance engineer, who has expertise in performance modeling, can specifically design the model learning logic in a VA for a given software stack. Ideally, vendors of

hypervisors, middleware platforms, or standardized enterprise applications directly design and integrate model learning capabilities into their systems and provide ready-to-use VAs.

Given the layered nature of software systems in virtualized data centers, the complete software stack is typically only known at system run-time. Therefore, our reference architecture is *agent-based* where each agent is responsible for creating and maintaining a certain subset – called model skeleton – of the complete performance model. We chose the DML meta-model as the performance model formalism for our reference architecture. Based on an analysis of the meta-model and a literature review of existing techniques for model learning, we derived different roles an agent may fulfill in our reference architecture and specified the collaborations between these agent roles. Furthermore, we proposed an algorithm for merging the different model skeletons into a complete performance model in a central repository. We discussed the constraints that model skeletons need to fulfill in order to allow for conflict-free merging.

We developed proof-of-concept implementations of our reference architecture and model extraction agents for representative systems. The reference implementation of our architecture is based on state-of-the-art technologies from MDE for maintaining the performance model in a central model repository. We built agents for the VMware vSphere virtualization platform – the market leader for x86 hypervisors –, for the WildFly application server – a widely used Java EE-compliant middleware platform –, and for Zimbra collaboration server – a standard enterprise collaboration application –. The idea of our reference architecture was first published in Spinner et al. (2013) and was later refined in Spinner et al. (2016). Furthermore, our work laid the foundation for a research project funded by the German Research Foundation (DFG) under grant No. KO 3445/11-1.

Online Method for the Statistical Estimation of Resource Demands

We provided the first systematization of of the state-of-the-art on resource demand estimation. Existing estimation approaches are categorized according to their required input parameters, their provided output metrics, and their measures to improve their robustness to anomalies in the measurement data. Furthermore, we evaluated the influence of different factors (sampling interval, number of samples, number of workload classes, load level, collinear workload classes, background jobs, and delayed processing) on the estimation accuracy of different estimation approaches. The results show, that using response times can improve the accuracy of the estimated resource demands significantly compared to a approaches solely based on utilization measurements, espe-

cially in cases with multiple workload classes. The systematization and the experimental comparison were published in Spinner et al. (2015a).

However, approaches employing response time measurements are very sensitive if assumptions of the underlying mathematical model are violated (e.g., wrong scheduling strategy, or delays due to other resources). To address this limitation, we described an online method for the statistical estimation of resource demands – called LibReDE – which is the first to apply multiple statistical techniques at run-time (in total eight) automatically combining their results. LibReDE checks the applicability of a statistical technique according to its the pre-conditions and executes only the applicable ones. We proposed a cross-validation scheme to automatically select the statistical technique providing the best results for a system under study. We implemented LibReDE as a Java library and published it under an open-source license to be available to other performance engineers. The tool is described in Spinner et al. (2014a). Furthermore, we integrated LibReDE into our reference architecture for online model learning to fully automate the estimation of resource demands.

LibReDE has been successfully used in a number of case studies. In a case study together with our industrial partner Google, we used LibReDE to estimate the resource demands of the SPECjEnterprise2010 full system benchmark, designed by the SPEC consortium to be representative of many Java EE workloads. We demonstrated the scalability of LibReDE which is able to estimate 80 different resource demand values with a total cross-validation error of 1.7% in less than 10 seconds. In a collaboration with the Fortiss research institute (see joint our publication Willnecker et al., 2015), we compared the results from LibReDE with the ones from the state-of-the-art Dynatrace APM tool. LibReDE was able to achieve a similar level of accuracy compared to the directly measured resource demands. In a case study in collaboration with SAP Research, we implemented a resource usage control mechanism for the SAP HANA cloud based on LibReDE successfully providing performance isolation in a multi-tenant environment (see our joint publication Krebs et al., 2014a). Finally, LibReDE also served as the basis for our model-based controllers described in this thesis.

Model-based Controllers for Autonomic Vertical Scaling of Virtualized Applications

We described two autonomic controllers for vertical scaling of virtualized applications using resource hot-plug mechanisms provided by modern hypervisors. The first controller is based on a model-adaptive control loop for short-term scaling – in the range of seconds to minutes – of virtual CPUs. We proposed a layered performance model based on queuing-theory reasoning on the impact

of changes to the resource allocation on the application performance. Furthermore, we showed how to estimate the resource demands explicitly capturing any scheduling delays in the hypervisor. We used the layered performance model to adapt the number of configured virtual CPUs of a VM in accordance with the application SLAs.

The second controller is based on a proactive control loop for vertical scaling of applications with limited elasticity on a mid- to long-term scale (i.e., hours or days). We proposed a method to workload forecasting based on multiple time series analysis methods incorporating meta-knowledge about the expected workloads. In our evaluation based on real-world traces, we showed that splitTs can improve the forecast accuracy quantified with the MASE error metric by 11 to 59%. We integrated splitTs in our proactive controller to schedule application reconfigurations, which potentially impact the user experience, during phases of low load in order to improve application availability and performance.

We evaluated both controllers in the context of case study together with our industrial partner VMware using the Zimbra collaboration server. We showed that the short-term controller can reduce the required number of virtual CPUs by up to 23% while reducing the number of reconfigurations by up to 95% when compared to a trigger-based controller. In the same case study, our proactive controller could reduce the impact of memory reconfigurations on the application availability and performance of the application by more than 80% compared to a reactive controller. Using splitTs, we were also able to reduce the over- and under-provisioning of memory compared to other state-of-the-art forecasting techniques to under 11%. The short-term controller was published in Spinner et al. (2014b) and the mid- to long-term controller in Spinner et al. (2015b).

8.2 Benefits

We see the following benefits of our self-aware approach to resource management in virtualized data centers:

- Our approach provides a high degree of automation through deep integration into the virtualization platform. System administrators do not need to create and maintain performance models of applications manually. Experts can tailor the model learning logic to a given platform technology or application and share it in the form of VA with a broader audience. System administrators can easily deploy such VAs and benefit from the automatically generated performance models without requiring the manual effort and the expertise typically involved with performance

modeling. Furthermore, the deep integration enables a tighter coupling of the models to the actual system in order to ensure that the models always reflect the current system architecture.

- Data center operators can optimize their resource allocations for greater efficiency and elasticity without negative impacts on the performance and availability services hosted in a data center. Performance models provide the missing link between low-level resource allocations and application-level SLAs regarding response time and throughput. Our autonomous controllers can exploit these models in order to determine the minimal amount of resources necessary to meet the application SLAs. This helps to significantly increase elasticity of applications and thus helps to reduce the over-provisioning of resources.

- Horizontal scaling requires additional architectural components in applications (e.g., load balancers) to work. Furthermore, the provisioning of additional VM instances can take significant time. Our controllers to vertical scaling also make it possible to provide the elasticity for applications not well suited for horizontal scaling (e.g., relational databases).

8.3 Future Work

The results of this thesis provide a basis for several topics of future work. In the following, we provide an overview of research topics extending our work.

- *Model learning capabilities for storage and network infrastructure*. In Section 4.2.1 we do not elaborate the roles for extracting the storage and network infrastructure in a data center in detail. The current version of DML does not include meta-models for these aspects so far, however, extensions are planned for future versions. When these extensions are available, the agent roles in Section 4.2.1 need to be updated accordingly. Descriptions of the structure and behavior (e.g., to determine network delays) need to be learned. Initial work in this direction can be found in Noorshams (2015) for storage resources and in Rygielski and Kounev (2014) for the network infrastructure.

- *Load-dependent resource demands*. The resource demands in a system may change with its load level. For instance, in Section 7.4.3 we show that interrupt coalescing – a common optimization for network communication – causes a resource demands to vary with the load level. Given that micro-service architectures become increasingly popular resulting in highly

distributed setups, we need to be able to reflect the load dependencies in our performance models. DML currently does not provide modeling elements and analysis techniques that support load-dependent resource demands. The modeling formalism needs to be extended accordingly in the future. At the same time, we also need to be able to include the load-dependency during resource demand estimation. Kumar et al. (2009b) are the only authors addressing this challenge so far. However, they assume a fixed functional shape that needs to be provided in advance. Future work may use our feedback loop proposed as part of LibReDE to also learn the functional shape of a load dependency.

- *Meta-learning techniques for selecting estimation approaches.* LibReDE currently selects the approach to resource demand estimation based on fixed pre-conditions and the results of the cross-validation. However, running many estimation approaches in parallel may be too expensive in an online setting. An alternative is the use of meta-learning approaches to derive decision rules based on historic data. These decision rules select estimation approaches based on certain features of the input data, such as, number of workload classes, load level, auto-correlation. Machine learning techniques (e.g., neuronal networks) may be used to determine the decision rules. The opportunities of meta-learning techniques is currently investigated as part of a research project funded by a Google faculty research award. Initial results are available in Grohmann (2016b).

- *Auto-tuning of estimation approaches*. The accuracy of many approaches to resource demand estimation is influenced by parameters (e.g., sampling interval length, or size of sliding window) that a user needs to provide. However, good values for these parameters depend on the system under study. Auto-tuning techniques to automatically find good values for these parameters during resource demand estimation promise to relieve the user from having to provide these values. The opportunities of meta-learning techniques is currently investigated as part of a research project funded by a Google faculty research award. Initial results are available in Grohmann (2016b).

- *Declarative performance engineering*. Self-aware computing systems explicitly consider the possibility for user interaction in order to reach high-level goals. It is not always reasonable to fully automate the act step in the LRA-M loop described in Section 2.1. For instance, the knowledge captured in automatically learnt performance models may support system administrators when performing manual one-time reconfigurations in a

data center. Walter et al. (2016) envision a declarative approach to performance engineering for automatically answering performance-related what-if questions (e.g., which is the bottleneck resource if the application workload doubles?). Approaches to declarative performance engineering may be integrated with our performance model repository in order to support system administrators with no or little experience in system performance analysis techniques to exploit the knowledge of automatically learnt models. Approaches to declarative performance engineering are currently investigated in the DFG Priority Programme "DFG-SPP 1593: Design For Future—Managed Software Evolution" as part of project grant KO 3445/15-1.

- *Multi-dimensional reconfiguration spaces*. In this thesis, we focus on vertical scaling of virtualized applications as a new mechanism for on-demand application scaling at run-time. However, vertical scaling is subject to limits prescribed by the application architecture and the physical hardware. A combination with horizontal scaling mechanisms as well as global and local resource scheduling techniques is necessary for optimal results. While our techniques for model learning result in performance models that help to answer different performance-related questions, future work may consider combined algorithms to multi-dimensional optimization based on these models.

- *Memory resource demands*. Main memory is a limiting resource for many enterprise applications. However, the quantification of memory resource demands is an open research question. While fine-grained instrumentation of applications is feasible (e.g., intercepting each allocation of memory), it is typically too expensive to be applied to production systems. Statistical estimation techniques described in Section 5.1 are designed for processing resources where the processing time can be directly associated to individual requests. Memory allocations are however often state-based and survive individual requests (e.g., in a session state). We require new estimation approaches to cope with such state-based, passive memory resources.

- *Identification of parameter dependencies*. The value of model variables (e.g., resource demands, or branching probabilities) may depend on the value of input parameters of requests to the system. In order to be able to predict the impact of variations in the input parameters on the application performance, these dependencies need to be captured explicitly. While Krogmann (2010) and Brosig (2014) propose abstractions to model these

parameter dependencies in architecture-level performance models, they require a user to specify them manually. Techniques to automatically identify these dependencies in an online manner could help to further improve the prediction power and accuracy of the extracted performance models.

List of Figures

2.1 LRA-M loop for self-aware computing systems (excerpt from Kounev et al., 2017b). 17
2.2 Ecore meta-metamodel. 19
2.3 DML meta-model overview (excerpt from Kounev et al., 2014). . 20
2.4 DML resource landscape (excerpt from Kounev et al., 2016). . . 21
2.5 DML application architecture (simplified). 23
2.6 Single service center 32
2.7 Example of an LQN model. 38

4.1 Conceptual overview of the reference architecture. 68
4.2 Communication diagram of agent interactions. 72
4.3 Model skeleton meta-model. 75
4.4 Sensor meta-model. 75
4.5 Notation. 80
4.6 Overview of the data center scope. 82
4.7 Overview of the usage scope. 83
4.8 Overview of the platform scope. 86
4.9 Overview of the application scope. 88
4.10 Overview of the model variable scope. 92
4.11 Example of overlapping model skeletons. 94

5.1 Types of input parameters. 113
5.2 Boxplot of demand estimation error E_d for different sampling intervals (dataset D1, load level $U = 50\%$, number of workload classes $C = 1$). 126
5.3 Overview of estimation method. 140
5.4 Workload description meta-model. 142

6.1 Overview of the model-adaptive control loop 155
6.2 Overview of layered queueing model. 156
6.3 Overview of proactive control loop. 170
6.4 Forecasts using splitTs on the FIFA'98 traces (first day is Friday). 174
6.5 Cumulative distribution function of absolute scaled errors. . . . 176

7.1 Excerpt of the vCenter object model (VMware, Inc., 2013). 184
7.2 Accuracy of resource demands with different numbers of tenants and requests. 194
7.3 Distributed deployment of SPECjEnterprise2010. 199
7.4 Mean relative error for different transaction rates. 206
7.5 Impact of interrupt coalescing. 209
7.6 Model estimates with increasing workload. 213
7.7 Hypervisor scheduling statics during increasing workload. . . . 213
7.8 Demand estimates under physical resource contention. 215
7.9 Dynamic workload used for the experiment. 217
7.10 Comparison of the MTA latency between the demand controller and the utilization controller. 217
7.11 Number of reconfigurations with the demand and the utilization controller (1 minute control interval). 218
7.12 Number of active users in a four-day workload trace. 220
7.13 Observed response times of the mailbox server. 221
7.14 Observed memory usage of the mailbox server VM. 223

List of Tables

4.1 Existing approaches to usage model extraction. 85
4.2 Existing approaches to application model extraction 91
4.3 Existing approaches to model parameterization (E stands for estimation and M for measurement). 93

5.1 Explanation of variables. 107
5.2 Overview of estimation approaches categorized according to statistical techniques. 108
5.3 Input parameters of estimation approaches (utilization U_i, response time R_c, throughput X_c, arrival rate λ_c, queue length $A_{i,c}$, visit counts $V_{i,c}$, demands $D_{i,c}$, think time Z, scheduling policy P). 115
5.4 Output metrics of estimation approaches. 118
5.5 Estimation approaches considered in the experimental evaluation. 124
5.6 Mean and standard deviation of demand estimation error E_d for different sampling intervals (dataset D1 and D2, load level $U = 50\%$, number of workload classes $C = 1$). N denotes the average number of requests observed in one sampling interval. 127
5.7 Mean demand estimation error E_d for different number of samples N (dataset D1, load level $U = 50\%$, number of workload classes $C = 1$). 127
5.8 Mean relative demand error E_d for number of workload classes $C = \{1, 2, 5\}$ (load level $U = 50\%$). 128
5.9 Correlation analysis results (dataset D1, load level $U = 50\%$). Entries with $\rho > 0.7$ are in bold letters. 130
5.10 Mean relative demand error E_d for high numbers of workload classes $C = \{5, 10, 15, 20\}$ (dataset D2, load level $U = 50\%$). . . . 131
5.11 Mean demand estimation error E_d for different load levels U and number of workload classes $C = 1$). 132
5.12 Sensitivity to collinearity in throughput observations. 133
5.13 Demand error E_d, utilization error E_u and response time error E_r when system executes background job with intensity U_b. . . 134

5.14 Demand error E_d, utilization error E_u, and response time error E_r when the jobs are interrupted by wait periods. 135
5.15 Mean execution time T (in milliseconds) partitioned by number of workload classes C and load level U. 136
5.16 List of estimation approaches including the underlying statistical techniques and the partitioning scheme (R: per-resource, T: per-tier, S: system). 147

6.1 Forecast accuracy. 175

7.1 Instrumentation points. 191
7.2 Comparison of Dynatrace and LibReDE (the values are the mean relative errors between the predicted and measured values). . . 196
7.3 List of extracted model entities. 201
7.4 Absolute errors between the calculated utilization and the simulation results. 203
7.5 Calculated (Calc.) and simulated (Sim.) response times including 95% c.i. Calculated values outside of the c.i. are bold. 203
7.6 Cross-validation errors and execution time. 204
7.7 Utilization predictions (absolute error in brackets). 207
7.8 Response time predictions (relative error in brackets). 208
7.9 Impact of mailbox server memory configuration. 214
7.10 Comparison of direct to indirect estimation of D_a^{app} for a given vCPU configuration. 216
7.11 Comparison of controller performance. 218
7.12 Comparison of controllers. 220
7.13 Allocation decisions of the proactive controller using splitTs and the WCF forecasters. 224

Acronyms

APM Application Performance Management. 54, 59, 91, 193, 225

ARMA Auto-Regressive-Moving-Average. 46

CBMG Customer Behavior Model Graph. 57, 85

CIM Common Information Model. 83

DML Descartes Modeling Language. 18, 20–22, 24, 25, 31, 51, 52, 64, 70, 73, 74, 76, 77, 79, 81, 94, 96, 97, 101, 103, 105, 139, 140, 143–145, 156, 182, 183, 188, 197, 198, 200–204, 208, 223, 224, 227, 228

DNI Descartes Network Infrastructure. 22

DRS Distributed Resource Scheduler. 47

DVFS Dynamic Voltage and Frequency Scaling. 117

EJB Enterprise Java Bean. 187–189, 195

EKF Extended Kalman filter. 27, 28

EMF Eclipse Modeling Framework. 19, 20, 94, 95, 141

FCFS First-Come-First-Serve. 32–34, 36, 38, 109, 112, 158, 165, 192

GPS Generalized Processor Sharing. 46, 160

HTML Hypertext Markup Language. 187

ICA Independent Component Analysis. 84

IIOP Internet Inter-ORB Protocol. 187, 189

IS Infinite-Server. 32, 34, 36, 158

Java EE Java Enterprise Edition. 7, 11, 56, 57, 67, 91, 102, 179, 185, 187, 188, 190, 195, 208, 224, 225

JCA Java EE Connector Architecture. 189, 190

JDBC Java Database Connectivity. 189, 196, 198

JMS Java Message Service. 189

JNDI Java Naming and Directory Interface. 188, 189

JSF Java Server Faces. 187–189

JSP Java Server Pages. 187–189

LAD Least Absolute Differences. 25

LCFS Last-Come-First-Serve. 34–36

LQN Layered Queueing Network. 37, 38, 47, 51, 56, 90, 157, 160, 163, 231

LSQ Least Squares. 25, 26

MARS Multivariate Adaptive Regression Splines. 54

MCMC Markov-chain Monte Carlo. 30

MDD Model-Driven Deveoplment. 18, 102, 140

MDE Model-Driven Engineering. 18, 224

MLE Maximum Likelihood Estimation. 112, 117

MOF Meta-Object Facility. 18, 19

MTA Mail Transfer Agent. 183–185, 209

MVA Mean Value Analysis. 37, 39, 113, 165

OMG Object Management Group. 18

OS Operating System. 21, 39–41, 43, 152, 159

OVF Open Virtualization Format. 41, 184

PCM Palladio Component Model. 56, 57, 70, 90, 91, 193

PS Processor-Sharing. 32, 34–36, 109, 112, 159, 160, 162

QN Queueing Network. 5, 8, 20, 31, 33, 34, 37, 38, 46, 47, 49, 51, 70, 90, 103, 110, 112, 113, 141, 145, 158, 159, 192, 201, 202

QoS Quality of Service. 46

QPN Queueing Petri Network. 20, 141, 201, 204

RMI Remote Method Invocation. 187, 189, 196

SaaS Software-as-a-Service. 191

SLA Service-level Agreement. 1–3, 47, 48, 52, 154, 223, 226, 227

SLO Service-level Objective. iii, v, 9–11, 151, 153, 163, 165–167, 180, 222

SMTP Simple Mail Transfer Protocol. 189

SNMP Simple Network Management Protocol. 83

SOAP Simple Object Access Protocol. 181, 187, 189, 196

UML Unified Modeling Language. 18, 55, 56, 80

URL Uniform Resource Locator. 82, 190

UUID Universally Unique Identifier. 97

VA Virtual Appliance. 7, 41, 65, 67–72, 78, 102, 179–181, 184–186, 190, 191, 198, 208, 209, 223, 224, 226

VM Virtual Machine. 2–4, 7, 9, 21, 39–51, 64–67, 69, 71, 77, 86, 87, 151–153, 155, 159, 160, 163, 165, 166, 176, 180–185, 193, 196, 197, 201, 207–210, 227

VMM Virtual Machine Monitor. 39

XML Extensible Markup Language. 41

Bibliography

Adams, K. and O. Agesen (2006). "A comparison of software and hardware techniques for x86 virtualization". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pp. 2–13 (see pages 40, 41).

Arlitt, M. F. (2000). "Characterizing Web user sessions". In: *SIGMETRICS Performance Evaluation Review* 28.2, pp. 50–63 (see pages 171, 173, 212, 216).

Awad, M. and D. A. Menascé (2014). "Dynamic Derivation of Analytical Performance Models in Autonomic Computing Environments". In: *Proceedings of the 2014 Computer Measurement Group Conference* (see pages 90, 91).

Balsamo, S. (2000). "Product Form Queueing Networks". In: *Performance Evaluation: Origins and Directions*, pp. 377–401 (see page 34).

Bard, Y. and M. Shatzoff (1978). "Statistical Methods in Computer Performance Analysis". In: *Current Trends in Programming Methodology* III (see pages 58, 108, 109, 115, 118).

Barham, P., A. Donnelly, R. Isaacs, and R. Mortier (2004). "Using Magpie for Request Extraction and Workload Modelling". In: *Proceedings of the 6th Symposium on Operating System Design and Implementation OSDI*, pp. 259–272 (see pages 58, 104, 145).

Barham, P. et al. (2003). "Xen and the art of virtualization". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pp. 164–177 (see pages 40, 41).

Barrett, E., E. Howley, and J. Duggan (2013). "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud". In: *Concurrency and Computation: Practice and Experience* 25.12, pp. 1656–1674 (see page 49).

Baskett, F., K. M. Chandy, R. R. Muntz, and F. G. Palacios (1975). "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers". In: *J. ACM* 22.2, pp. 248–260 (see pages 34, 35, 149).

Becker, S., H. Koziolek, and R. H. Reussner (2009). "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1, pp. 3–22 (see pages 53, 70).

Bellard, F. (2005). "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pp. 41–46 (see page 39).

Bennani, M. N. and D. A. Menascé (2005). "Resource Allocation for Autonomic Data Centers using Analytic Performance Models". In: *Second International Conference on Autonomic Computing (ICAC 2005), 13-16 June 2005, Seattle, WA, USA*, pp. 229–240 (see page 47).

Bittman, T. J., P. Dawson, and M. Warrilow (2016). *Magic Quadrant for x86 Server Virtualization Infrastructure* (see pages 181, 183).

Blagodurov, S., D. Gmach, M. F. Arlitt, Y. Chen, C. Hyser, and A. Fedorova (2013). "Maximizing server utilization while meeting critical SLAs via weight-based collocation management". In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27-31, 2013*, pp. 277–285 (see page 46).

Bodík, P., R. Griffith, C. A. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson (2009). "Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters". In: *Workshop on Hot Topics in Cloud Computing, HotCloud'09, San Diego, CA, USA, June 15, 2009* (see page 50).

Bolch, G., S. Greiner, H. de Meer, and K. S. Trivedi (2006). *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons (see pages 31, 33, 34, 36, 37, 125).

Box, G., G. Jenkins, and G. Reinsel (2008a). *Time Series Analysis : Forecasting and Control*. 4. ed. Wiley (see page 120).

– (2008b). *Time Series Analysis : Forecasting and Control*. 4. ed. Wiley (see page 175).

Briand, L. C., Y. Labiche, and J. Leduc (2006). "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software". In: *IEEE Trans. Software Eng.* 32.9, pp. 642–663 (see page 56).

Brosig, F. (July 2014). "Architecture-Level Software Performance Models for Online Performance Prediction". PhD thesis. Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany (see pages 31, 63, 64, 73, 77, 85, 143, 202, 206, 231).

Brosig, F., N. Huber, and S. Kounev (2011). "Automated extraction of architecture-level performance models of distributed component-based systems". In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pp. 183–192 (see pages 7, 56, 57, 65, 71, 85, 91, 93, 194, 197).

Brosig, F., S. Kounev, and K. Krogmann (2009). "Automated extraction of palladio component models from running enterprise Java applications". In: *Proceedings of the 4th International Conference on Performance Evaluation Method-*

ologies and Tools, VALUETOOLS, p. 10 (see pages 58, 107, 108, 115, 118, 124, 147, 202).

Brunnert, A., C. Vögele, and H. Krcmar (2013). "Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications". In: *Proceedings of the 10th European Workshop Computer Performance Engineering, EPEW*, pp. 74–88 (see pages 7, 56–58, 65, 71, 91, 93, 104, 145, 195, 197).

Carrera, D., J. Guitart, J. Torres, E. Ayguadé, and J. Labarta (2003). "Complete instrumentation requirements for performance analysis of Web based technologies". In: *2003 IEEE International Symposium on Performance Analysis of Systems and Software, March 6-8, 2003, Austin, Texas, USA, Proceedings*, pp. 166–175 (see page 54).

Casale, G., P. Cremonesi, and R. Turrin (2008). "Robust Workload Estimation in Queueing Network Performance Models". In: *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP*, pp. 183–187 (see pages 58, 108, 109, 115, 118–121).

Casale, G., P. Cremonesi, and R. Turrin (2007). "How to Select Significant Workloads in Performance Models". In: *CMG Conference Proceedings* (see pages 58, 108, 109, 115, 116, 118–121).

Chandra, A., W. Gong, and P. J. Shenoy (2003). "Dynamic Resource Allocation for Shared Data Centers Using Online Measurements". In: *Quality of Service - IWQoS 2003, 11th International Workshop, Berkeley, CA, USA, June 2-4, 2003, Proceedings*, pp. 381–400 (see page 46).

Chardigny, S. and A. Seriai (2010). "Software Architecture Recovery Process Based on Object-Oriented Source Code and Documentation". In: *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings*, pp. 409–416 (see page 55).

Chatterjee, S. and B. Price (1995). *Praxis der Regressionsanalyse*. Oldenbourg (see pages 25, 26, 121).

Chouambe, L., B. Klatt, and K. Krogmann (2008). "Reverse Engineering Software-Models of Component-Based Systems". In: *12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece*, pp. 93–102 (see page 55).

Courtois, M. and C. M. Woodside (2000). "Using regression splines for software performance analysis". In: *Workshop on Software and Performance*, pp. 105–114 (see pages 54, 93).

Cox, D. R. (Apr. 1955). "A use of complex probabilities in the theory of stochastic processes". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 51 (02), pp. 313–319. ISSN: 1469-8064 (see page 35).

Cremonesi, P., K. Dhyani, and A. Sansottera (2010). "Service Time Estimation with a Refinement Enhanced Hybrid Clustering Algorithm". In: *Proceedings of the 17th International Conference on Analytical and Stochastic Modeling Techniques and Applications, ASMTA*, pp. 291–305 (see pages 58, 108, 111, 115, 118, 121).

Cremonesi, P. and A. Sansottera (Sept. 2012). "Indirect Estimation of Service Demands in the Presence of Structural Changes". In: *Proceedings of the 2012 Ninth International Conference on Quantitative Evaluation of Systems, QEST*, pp. 249–259 (see pages 108, 111, 115, 118).

– (2014). "Indirect estimation of service demands in the presence of structural changes". In: *Performance Evaluation* 73, pp. 18–40 (see pages 108, 111, 115, 118).

Crnkovic, I., H. W. Schmidt, J. A. Stafford, and K. C. Wallnau (2005). "Automated Component-Based Software Engineering". In: *Journal of Systems and Software* 74.1, pp. 1–3 (see page 53).

De Livera, A. M., R. J. Hyndman, and R. D. Snyder (2011). "Forecasting Time Series With Complex Seasonal Patterns Using Exponential Smoothing". In: *Journal of the American Statistical Association* 106.496, pp. 1513–1527 (see page 175).

Ding, L. and N. Medvidovic (2001). "Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution". In: *2001 Working IEEE / IFIP Conference on Software Architecture (WICSA 2001), 28-31 August 2001, Amsterdam, The Netherlands*, p. 191 (see page 55).

DMTF (Feb. 2009). *Open Virtualization Format Specification, Version 1.0.0* (see page 41).

Dostl, Z. (2009). *Optimal Quadratic Programming Algorithms: With Applications to Variational Inequalities*. Springer Publishing Company, Incorporated (see pages 28, 29).

Dutreilh, X., S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck (May 2011). "Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: Towards a Fully Automated Workflow". In: *7th International Conference on Autonomic and Autonomous Systems, ICAS 2011*, pp. 67–74 (see page 49).

Förtsch, S. and B. Westfechtel (2007). "Differencing and Merging of Software Diagrams - State of the Art and Challenges". In: *Proceedings of the Second International Conference on Software and Data Technologies, INSTICC*. Ed. by J. Filipe, B. Shishkow, and M. Helfert (see pages 95, 97).

Franks, G. (1999). "Performance Analysis of Distributed Server Systems". PhD thesis. Carlton University (see page 36).

Franks, G., T. Omari, C. M. Woodside, O. Das, and S. Derisavi (2009). "Enhanced Modeling and Solution of Layered Queueing Networks". In: *IEEE Trans. Software Eng.* 35.2, pp. 148–161 (see pages 38, 160).

Gandhi, A., Y. Chen, D. Gmach, M. F. Arlitt, and M. Marwah (2011). "Minimizing data center SLA violations and power consumption via hybrid resource provisioning". In: *2011 International Green Computing Conference and Workshops, IGCC 2012, Orlando, FL, USA, July 25-28, 2011*, pp. 1–8 (see pages 5, 51, 52).

Gandhi, A., P. Dube, A. Karve, A. Kochut, and L. Zhang (2014). "Adaptive, Model-driven Autoscaling for Cloud Applications". In: *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014.* Pp. 57–64 (see page 51).

Gartner, Inc. (2015). *Magic Quadrant for x86 Server Virtualization Infrastructure* (see page 1).

Geman, S. and D. Geman (Nov. 1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6, pp. 721–741 (see page 30).

Godard, S. (2014). *SYSSTAT utilities*. Website. Online available at `http://sebastien.godard.pagesperso-orange.fr/`. Last accessed on 07-07-2014 (see page 123).

Gong, Z., X. Gu, and J. Wilkes (2010). "PRESS: PRedictive Elastic ReSource Scaling for cloud systems". In: *Proceedings of the 6th International Conference on Network and Service Management, CNSM 2010, Niagara Falls, Canada, October 25-29, 2010*, pp. 9–16 (see page 50).

Graham, S. L., P. B. Kessler, and M. K. Mckusick (1982). "Gprof: A Call Graph Execution Profiler". In: *SIGPLAN Not.* 17.6, pp. 120–126 (see pages 57, 103).

Grassi, V., R. Mirandola, and A. Sabetta (2007). "Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach". In: *Journal of Systems and Software* 80.4, pp. 528–558 (see page 53).

Grohmann, J. (Oct. 2016a). "Reliable Resource Demand Estimation". Master Thesis. Am Hubland, Informatikgebäude, 97074 Würzburg, Germany: University of Würzburg (see page 148).

Gulati, A., G. Shanmuganathan, A. Holler, C. Waldspurger, M. Ji, and X. Zhu (2012). "VMware Distributed Resource Management: Design, Implementation, and Lessons Learned". In: *VMware Technical Journal* 1.1, pp. 45–64 (see pages 4, 42, 47, 166).

Gupta, R., S. K. Bose, S. Sundarrajan, M. Chebiyam, and A. Chakrabarti (2008). "A Two Stage Heuristic Algorithm for Solving the Server Consolidation

Problem with Item-Item and Bin-Item Incompatibility Constraints". In: *2008 IEEE International Conference on Services Computing (SCC 2008), 8-11 July 2008, Honolulu, Hawaii, USA*, pp. 39–46 (see page 47).

Hall, R. J. (1992). "Call Path Profiling". In: *Proceedings of the 14th International Conference on Software Engineering (ICSE '92)*. Melbourne, Australia: ACM, pp. 296–306 (see pages 57, 103).

Harchol-Balter, M. (2013). *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press (see pages 31, 33–35, 130).

Hastings, W. K. (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". In: *Biometrika* 57.1, pp. 97–109. ISSN: 00063444 (see page 30).

Hauck, M., M. Kuperberg, N. Huber, and R. H. Reussner (2014). "Deriving performance-relevant infrastructure properties through model-based experiments with Ginpex". In: *Software and System Modeling* 13.4, pp. 1345–1365 (see page 123).

Herbst, N. R., N. Huber, S. Kounev, and E. Amrehn (2014). "Self-adaptive workload classification and forecasting for proactive resource provisioning". In: *Concurrency and Computation: Practice and Experience* 26.12, pp. 2053–2078 (see pages 45, 171–174, 223).

Herbst, N. R., S. Kounev, and R. H. Reussner (2013). "Elasticity in Cloud Computing: What It Is, and What It Is Not". In: *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, pp. 23–27 (see pages iii, vii, 2).

Hoorn, A. van, C. Vögele, E. Schulz, W. Hasselbring, and H. Krcmar (2014). "Automatic Extraction of Probabilistic Workload Specifications for Load Testing Session-Based Application Systems". In: *8th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2014, Bratislava, Slovakia, December 9-11, 2014* (see pages 85, 200).

– (2015). "Automatic Extraction of Probabilistic Workload Specifications for Load Testing Session-Based Application Systems". In: *EAI Endorsed Trans. Self-Adaptive Systems* 1.3, e5 (see pages 57, 85, 200).

Hrischuk, C. E., C. M. Woodside, J. A. Rolia, and R. Iversen (1999). "Trace-Based Load Characterization for Generating Performance Software Models". In: *IEEE Trans. Software Eng.* 25.1, pp. 122–135 (see page 56).

Huber, N. (July 2014). "Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures". PhD thesis. Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany (see pages 21, 63, 64, 81).

Huber, N., A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev (2014). "Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments". In: *Service Oriented Computing and Applications* 8.1, pp. 73–89 (see pages 51, 52).

Huber, N., M. von Quast, M. Hauck, and S. Kounev (2011). "Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments". In: *CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science, Noordwijkerhout, Netherlands, 7-9 May, 2011*, pp. 563–573 (see pages 21, 22, 86, 87).

Hyndman, R. J. and Y. Khandakar (July 29, 2008). "Automatic Time Series Forecasting: The Forecast Package for R". In: *Journal of Statistical Software* 27.3, pp. 1–22. ISSN: 1548-7660 (see page 171).

Hyndman, R. J. and A. B. Koehler (2006). "Another look at measures of forecast accuracy". In: *International Journal of Forecasting* 22.4, pp. 679–688 (see page 174).

Israr, T. A., C. M. Woodside, and G. Franks (2007). "Interaction tree algorithms to extract effective architecture and layered performance models from traces". In: *Journal of Systems and Software* 80.4, pp. 474–492 (see pages 56, 90, 91, 93).

Jennings, B. and R. Stadler (2015). "Resource Management in Clouds: Survey and Research Challenges". In: *J. Network Syst. Manage.* 23.3, pp. 567–619 (see pages 45–47).

Jung, G., M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu (2010). "Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures". In: *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pp. 62–73 (see pages 5, 51, 52).

Jung, G., K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu (2008). "Generating Adaptation Policies for Multi-tier Applications in Consolidated Server Environments". In: *2008 International Conference on Autonomic Computing, ICAC 2008, June 2-6, 2008, Chicago, Illinois, USA*, pp. 23–32 (see page 47).

Kalbasi, A., D. Krishnamurthy, J. Rolia, and S. Dawson (2012). "DEC: Service Demand Estimation with Confidence". In: *IEEE Trans. Software Eng.* 38.3, pp. 561–578 (see pages 108, 113, 115, 117, 118).

Kalbasi, A., D. Krishnamurthy, J. Rolia, and M. Richter (2011). "MODE: Mix Driven On-line Resource Demand Estimation". In: *Proceedings of the 7th International Conference on Network and Service Management, CNSM*, pp. 1–9 (see pages 58, 108, 111, 115, 118).

Kalyvianaki, E., T. Charalambous, and S. Hand (2014). "Adaptive Resource Provisioning for Virtualized Servers Using Kalman Filters". In: *TAAS* 9.2, 10:1–10:35 (see pages 5, 50).

Kebir, S., A. Seriai, S. Chardigny, and A. Chaoui (2012). "Quality-Centric Approach for Software Component Identification from Object-Oriented Code". In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA 2012, Helsinki, Finland, August 20-24, 2012*, pp. 181–190 (see page 55).

Kelly, F. P. (1975). "Networks of Queues with Customers of Different Types". In: *Journal of Applied Probability* 12.3, pp. 542–554. ISSN: 00219002 (see page 34).

– (1976). "Networks of Queues". In: *Advances in Applied Probability* 8.2, pp. 416–432. ISSN: 00018678 (see page 34).

Kelly, T. and A. Zhang (2006). *Predicting performance in distributed enterprise applications*. Tech. rep. HP Labs Tech Report (see pages 108, 109, 115).

Kendall, D. G. (Sept. 1953). "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain". In: *The Annals of Mathematical Statistics* 24.3, pp. 338–354 (see page 32).

Kephart, J. O. (2011). "Autonomic computing: the first decade". In: *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011*, pp. 1–2 (see page 15).

Kephart, J. O. and D. M. Chess (2003). "The Vision of Autonomic Computing". In: *IEEE Computer* 36.1, pp. 41–50 (see page 15).

Kistowski, J. von, N. R. Herbst, D. Zöller, S. Kounev, and A. Hotho (2015). "Modeling and Extracting Load Intensity Profiles". In: *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, pp. 109–119 (see pages 57, 85, 223).

Kounev, S. (2008). "Software Performance Evaluation". In: ed. by B. W. Wah. Wiley Encyclopedia of Computer Science and Engineering, Wiley-Interscience, John Wiley & Sons Inc. (see page 1).

Kounev, S., F. Brosig, and N. Huber (Oct. 2014). *The Descartes Modeling Language*. Tech. rep. Department of Computer Science, University of Wuerzburg (see page 20).

Kounev, S. and A. P. Buchmann (2006). "SimQPN - A tool and methodology for analyzing queueing Petri net models by means of simulation". In: *Perform. Eval.* 63.4-5, pp. 364–394 (see pages 203, 206).

Kounev, S., N. Huber, F. Brosig, and X. Zhu (2016). "Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures". In: *IEEE Computer*

Magazine. Accepted for Publication, To appear in 2016 (see pages 18, 20, 21, 70, 74).

Kounev, S., X. Zhu, J. O. Kephart, and M. Kwiatkowska (2015). "Model-driven Algorithms and Architectures for Self-Aware Computing Systems (Dagstuhl Seminar 15041)". In: *Dagstuhl Reports* 5.1. Ed. by S. Kounev, X. Zhu, J. O. Kephart, and M. Kwiatkowska, pp. 164–196. ISSN: 2192-5283 (see pages 15, 16, 63).

Koziolek, H. (2010). "Performance evaluation of component-based software systems: A survey". In: *Perform. Eval.* 67.8, pp. 634–658 (see page 53).

Kraft, S., S. Pacheco-Sanchez, G. Casale, and S. Dawson (2009). "Estimating service resource consumption from response time measurements". In: *Proceedings of the 4th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS*, p. 48 (see pages 59, 108, 109, 112, 115–118, 123, 124, 147, 160, 202).

Krebs, R. (Aug. 2015). "Performance Isolation in Multi-Tenant Applications". PhD thesis. Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany (see page 193).

Krebs, R., C. Momm, and S. Kounev (Feb. 2014b). "Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments". In: *Elsevier Science of Computer Programming Journal (SciCo)* Vol. 90, Part B, pp. 116–134 (see page 193).

Krebs, R., A. Wert, and S. Kounev (July 8–12, 2013). "Multi-Tenancy Performance Benchmark for Web Application Platforms". Industrial Track. In: *Proceedings of the 13th International Conference on Web Engineering (ICWE 2013)*. Aalborg University, Denmark. Aalborg, Denmark: Springer-Verlag (see page 195).

Kremien, O. and J. Kramer (1992). "Methodical Analysis of Adaptive Load Sharing Algorithms". In: *IEEE Trans. Parallel Distrib. Syst.* 3.6, pp. 747–760 (see page 48).

Krogmann, K. (2010). "Reconstruction of software component architectures and behaviour models using static and dynamic analysis". PhD thesis. Karlsruhe Institute of Technology (see pages 91, 93, 231).

Krogmann, K., M. Kuperberg, and R. H. Reussner (2010). "Using Genetic Search for Reverse Engineering of Parametric Behavior Models for Performance Prediction". In: *IEEE Trans. Software Eng.* 36.6, pp. 865–877 (see pages 56, 57).

Kumar, D., A. N. Tantawi, and L. Zhang (2009a). "Real-time performance modeling for adaptive software systems with multi-class workload". In: *Proceedings of the 17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS*, pp. 1–4 (see pages 26–28, 58, 108–110, 115, 118, 121, 124).

Kumar, D., L. Zhang, and A. N. Tantawi (2009b). "Enhanced inferencing: estimation of a workload dependent performance model". In: *Proceedings of the 4th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS*, p. 47 (see pages 58, 108, 110, 115–118, 210, 230).

Kumar, S., V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan (2009c). "vManage: loosely coupled platform and virtualization management in data centers". In: *Proceedings of the 6th International Conference on Autonomic Computing, ICAC 2009, June 15-19, 2009, Barcelona, Spain*, pp. 127–136 (see pages 10, 48).

Kuperberg, M., M. Krogmann, and R. H. Reussner (2009). "TimerMeter: Quantifying Properties of Software Timers for System Analysis". In: *Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems, QEST*, pp. 85–94 (see page 104).

Kupferman, J., J. Silverman, P. Jara, and J. Browne (2009). *Scaling into the cloud.* Tech. rep. CS270 - Advanced Operating Systems. University of California, Santa Barbara (see page 49).

Kutner, M. H., C. J. Nachtsheim, and J. Neter (2003). *Applied Linear Regression Models*. The McGraw-Hill/Irwin Series Operations and Decision Sciences. McGraw-Hill Higher Education (see page 133).

Lazowska, E. D., J. Zahorjan, G. S. Graham, and K. C. Sevcik (1984). *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. (see pages 31, 32, 46, 103, 107, 108, 115, 116, 118).

Li, B., J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong (2009). "EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments". In: *IEEE International Conference on Cloud Computing, CLOUD 2009, Bangalore, India, 21-25 September, 2009*, pp. 17–24 (see page 47).

Lim, H., S. Babu, and J. S. Chase (2010). "Automated control for elastic storage". In: *Proceedings of the 7th International Conference on Autonomic Computing, ICAC 2010, Washington, DC, USA, June 7-11, 2010*, pp. 1–10 (see page 50).

Liu, H. and V. Keselj (2007). "Combined mining of Web server logs and web contents for classifying user navigation patterns and predicting users' future requests". In: *Data Knowl. Eng.* 61.2, pp. 304–330 (see pages 84, 85).

Liu, Z., L. Wynter, C. H. Xia, and F. Zhang (2006). "Parameter inference of queueing models for IT systems using end-to-end measurements". In: *Perform. Eval.* 63.1, pp. 36–60 (see pages 58, 108, 110, 111, 115, 116, 118, 120, 124, 137, 147, 160, 202, 205, 210).

Liu, Z., C. H. Xia, P. Momcilovic, and L. Zhang (2003). *AMBIENCE: Automatic Model Building using IferENCE*. Tech. rep. IBM Research (see pages 108, 110, 115, 116, 118).

Liu, Z. et al. (2012). "Renewable and cooling aware workload management for sustainable data centers". In: *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pp. 175–186 (see page 48).

Lorido-Botran, T., J. Miguel-Alonso, and J. A. Lozano (2014). "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments". In: *J. Grid Comput.* 12.4, pp. 559–592 (see pages 45, 48, 49).

Lu, L., H. Zhang, G. Jiang, H. Chen, K. Yoshihira, and E. Smirni (2011). "Untangling mixed information to calibrate resource utilization in virtual machines". In: *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011*, pp. 151–160 (see page 87).

Lu, L., X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. Smirni (2014). "Application-driven dynamic vertical scaling of virtual machines in resource pools". In: *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, pp. 1–9 (see pages 10, 48, 166).

Menascé, D. A. (2002). "Simple analytic modeling of software contention". In: *SIGMETRICS Performance Evaluation Review* 29.4, pp. 24–30 (see page 157).

– (2008). "Computing missing service demand parameters for performance models". In: *CMG Conference Proceedings*, pp. 241–248 (see pages 58, 108, 110, 111, 114, 115, 118, 124, 137, 147, 160).

Menascé, D. A., V. Almeida, R. C. Fonseca, and M. A. Mendes (1999). "A methodology for workload characterization of E-commerce sites". In: *EC*, pp. 119–128 (see pages 57, 85).

Menascé, D. A., L. W. Dowdy, and V. A. F. Almeida (2004). *Performance by Design: Computer Capacity Planning By Example*. Upper Saddle River, NJ, USA: Prentice Hall PTR (see pages 32, 35, 46, 103, 107, 161).

Menascé, D. A. and H. Gomaa (2000). "A Method for Design and Performance Modeling of Client/Server Systems". In: *IEEE Trans. Software Eng.* 26.11, pp. 1066–1085 (see page 106).

Meyer, R. A. and L. H. Seawright (1970). "A virtual machine time-sharing system". In: *IBM Systems Journal* 9.3, pp. 199–218 (see page 39).

Miller, K. and M. Pegah (2007). "Virtualization: virtually at the desktop". In: *Proceedings of the 35th Annual ACM SIGUCCS Conference on User Services 2007, Orlando, Florida, USA, October 7-10, 2007*, pp. 255–260 (see page 39).

Neilson, J. E., C. M. Woodside, D. C. Petriu, and S. Majumdar (1995). "Software Bootlenecking in Client-Server Systems and Rendezvous Networks". In: *IEEE Trans. Software Eng.* 21.9, pp. 776–782 (see page 145).

Nemhauser, G. L. [, ed. (1989). *Optimization*. Handbooks in operations research and management science ; 1. Amsterdam [u.a.]: North-Holland (see page 29).

Nguyen, H., Z. Shen, X. Gu, S. Subbiah, and J. Wilkes (2013). "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service". In: *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, pp. 69–82 (see page 50).

Noorshams, Q. (2015). "Modeling and Prediction of I/O Performance in Virtualized Environments". PhD thesis. Karlsruhe Institute of Technology (KIT) (see pages 81, 229).

Noorshams, Q., D. Bruhn, S. Kounev, and R. H. Reussner (2013). "Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques". In: *ACM/SPEC International Conference on Performance Engineering, ICPE'13, Prague, Czech Republic - April 21 - 24, 2013*, pp. 283–294 (see pages 54, 148).

Nou, R., S. Kounev, F. Julià, and J. Torres (2009). "Autonomic QoS control in enterprise Grid environments using online simulation". In: *Journal of Systems and Software* 82.3, pp. 486–502 (see pages 107, 108, 115, 118).

OMG (May 2006). *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)* (see page 53).

– (June 2015). *OMG Meta Object Facility (MOF) Core Specification. Version 2.5* (see page 18).

Oracle America, I. (2013). *Java Platform, Enterprise Edition (Java EE) Specification, v7* (see page 189).

Ousterhout, J. K. (1982). "Scheduling techniques for concurrent systems". In: *3rd International Conference on Distributed Computing Systems*, pp. 22–30 (see page 160).

Pacifici, G., W. Segmuller, M. Spreitzer, and A. N. Tantawi (2008). "CPU demand for web serving: Measurement analysis and dynamic estimation". In: *Perform. Eval.* 65.6-7, pp. 531–553 (see pages 58, 108, 109, 115, 116, 118–121, 130, 133, 147).

Padala, P., A. Holler, L. Lu, A. Parikh, M. Yechuri, and X. Zhu (2014). "Scaling of Cloud Applications Using Machine Learning". In: 3.1 (see page 49).

Padala, P. et al. (2009). "Automated control of multiple virtualized resources". In: *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pp. 13–26 (see pages 5, 46, 48).

Parekh, A. K. and R. G. Gallager (1994). "A generalized processor sharing approach to flow control in integrated services networks: the multiple node case". In: *IEEE/ACM Trans. Netw.* 2.2, pp. 137–150 (see page 160).

Pérez, J. F. and G. Casale (2013a). "Assessing SLA Compliance from Palladio Component Models". In: *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013*, pp. 409–416 (see page 160).

– (2013b). "Assessing SLA Compliance from Palladio Component Models". In: *Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC*, pp. 409–416 (see page 163).

Pérez, J. F., G. Casale, and S. Pacheco-Sanchez (2015). "Estimating Computational Requirements in Multi-Threaded Applications". In: *IEEE Trans. Software Eng.* 41.3, pp. 264–278 (see pages 115, 118).

Pérez, J. F., S. Pacheco-Sanchez, and G. Casale (2013). "An Offline Demand Estimation Method for Multi-threaded Applications". In: *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS*, pp. 21–30 (see pages 59, 108, 109, 112, 115, 116, 118, 124, 138).

Petriu, D. B. and C. M. Woodside (2007). "An intermediate metamodel with scenarios and resources for generating performance models from UML designs". In: *Software and System Modeling* 6.2, pp. 163–184 (see page 53).

Popek, G. J. and R. P. Goldberg (1974). "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7, pp. 412–421 (see pages 39, 40).

Rao, J., X. Bu, K. Wang, and C. Xu (2011). "Self-adaptive provisioning of virtualized resources in cloud computing". In: *SIGMETRICS 2011, Proceedings of the 2011 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, San Jose, CA, USA, 07-11 June 2011 (Co-located with FCRC 2011)*, pp. 129–130 (see page 46).

Rao, J., X. Bu, C. Xu, L. Y. Wang, and G. G. Yin (2009). "VCONF: a reinforcement learning approach to virtual machines auto-configuration". In: *Proceedings of the 6th International Conference on Autonomic Computing, ICAC 2009, June 15-19, 2009, Barcelona, Spain*, pp. 137–146 (see page 49).

Rathfelder, C. (2013). *Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation*. Vol. 10. The Karlsruhe Series on Software Design and Quality. Karlsruhe, Germany: KIT Scientific Publishing (see page 80).

RightScale, I. (2016). Website. Online available at `http://docs.rightscale.com/faq/What_is_auto-scaling.html`. Last accessed: 04-11-2016 (see page 49).

Rohr, M., A. van Hoorn, S. Giesecke, J. Matevska, W. Hasselbring, and S. Alekseev (2008a). "Trace-Context Sensitive Performance Profiling for Enterprise

Software Applications". In: *Performance Evaluation: Metrics, Models and Benchmarks, SPEC International Performance Evaluation Workshop, SIPEW 2008, Darmstadt, Germany, June 27-28, 2008. Proceedings*, pp. 283–302 (see pages 54, 57, 91).

Rohr, M., A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring (Feb. 2008b). "Kieker: Continuous Monitoring and on demand Visualization of Java Software Behavior". In: *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08)*. Ed. by C. Pahl. Anaheim, CA, USA: ACTA Press, pp. 80–85 (see page 71).

Rolia, J. A. and K. C. Sevcik (1995). "The Method of Layers". In: *IEEE Trans. Software Eng.* 21.8, pp. 689–700 (see pages 39, 157, 158, 163).

Rolia, J., A. Kalbasi, D. Krishnamurthy, and S. Dawson (2010). "Resource demand modeling for multi-tier services". In: *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering, ICPE*, pp. 207–216 (see pages 108, 113, 115, 117, 118, 121).

Rolia, J. and B. Lin (1994). "Consistency issues in distributed application performance metrics". In: *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON*, p. 62 (see page 126).

Rolia, J. and V. Vetland (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*, p. 54 (see pages 58, 108, 109, 115–118, 124, 126, 147, 194, 202).

– (1998). "Correlating resource demand information with ARM data for application services". In: *Proceedings of the first International Workshop on Software and Performance, WOSP*, pp. 219–230 (see pages 108, 109, 115, 118).

Rometsch, F. and H. Sauer (2008). "Dynatrace Diagnostics: Performance-Management und Fehlerdiagnose vereint". In: *iX* 9/2008. Ed. by R. Hülsenbusch (see pages 54, 58, 71).

Rygielski, P. and S. Kounev (2013). "Network Virtualization for QoS-Aware Resource Management in Cloud Data Centers: A Survey". In: *Praxis der Informationsverarbeitung und Kommunikation* 36.1, pp. 55–64 (see page 39).

– (Sept. 2014). *Descartes Network Infrastructures (DNI) Manual: Meta-models, Transformations, Examples*. Technical Report v.0.3. Am Hubland, 97074 Würzburg: Chair of Software Engineering, University of Würzburg (see pages 22, 81, 229).

Salomie, T., G. Alonso, T. Roscoe, and K. Elphinstone (2013). "Application level ballooning for efficient server consolidation". In: *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pp. 337–350 (see page 169).

Schmidt, D. C. (2006). "Guest Editor's Introduction: Model-Driven Engineering". In: *IEEE Computer* 39.2, pp. 25–31 (see page 18).

Seawright, L. H. and R. A. MacKinnon (1979). "VM/370 - A Study of Multiplicity and Usefulness". In: *IBM Systems Journal* 18.1, pp. 4–17 (see page 40).

Shanmuganathan, G., A. Gulati, A. Holler, S. Kalyanaraman, P. Padala, X. Zhu, and R. Griffith (2013). "Towards Proactive Resource Management in Virtualized Datacenters". In: *Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE)* (see pages 10, 48).

Sharma, A. B., R. Bhagwan, M. Choudhury, L. Golubchik, R. Govindan, and G. M. Voelker (2008). "Automatic request categorization in internet services". In: *SIGMETRICS Performance Evaluation Review* 36.2, pp. 16–25 (see pages 59, 84, 85, 108, 111, 115, 118, 123).

Shen, Z., S. Subbiah, X. Gu, and J. Wilkes (2011). "CloudScale: elastic resource scaling for multi-tenant cloud systems". In: *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, p. 5 (see pages 10, 50, 223).

Simon, D. (2006). *Optimal state estimation : Kalman, H. [infinity] and nonlinear approaches*. Hoboken, NJ: Wiley-Interscience (see pages 26, 27).

Smith, C. U., C. M. Lladó, V. Cortellessa, A. D. Marco, and L. G. Williams (2005). "From UML models to software performance results: an SPE process based on XML interchange formats". In: *Proceedings of the Fifth International Workshop on Software and Performance, WOSP*, pp. 87–98 (see page 53).

Smith, C. U. and L. G. Williams (2002). *Performance Solutions- A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley (see page 1).

Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Springer-Verlag, Wien (see page 16).

Standard Performance Evaluation Corporation (2010). *SPECjEnterprise2010 Design Document*. Website. Online available at `https://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html`. Last accessed on 02-02-2017 (see page 198).

Steinberg, D., F. Budinsky, M. Paternostro, and E. Merks (2009). *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional (see page 19).

Stewart, C., T. Kelly, and A. Zhang (2007). "Exploiting nonstationarity for performance prediction". In: *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pp. 31–44 (see pages 108, 109, 115, 116, 120, 129).

Sutton, C. and M. I. Jordan (Mar. 2011). "Bayesian inference for queueing networks and modeling of internet services". In: *The Annals of Applied Statistics* 5.1, pp. 254–282 (see pages 59, 108, 112, 115, 118).

Tesauro, G., N. K. Jong, R. Das, and M. N. Bennani (2007). "On the use of hybrid reinforcement learning for autonomic resource allocation". In: *Cluster Computing* 10.3, pp. 287–299 (see pages 5, 49).

Tumanov, A., J. Cipar, G. R. Ganger, and M. A. Kozuch (2012). "alsched: algebraic scheduling of mixed workloads in heterogeneous clouds". In: *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, p. 25 (see page 48).

Urdaneta, G., G. Pierre, and M. van Steen (2009). "Wikipedia workload analysis for decentralized hosting". In: *Computer Networks* 53.11, pp. 1830–1845 (see pages 171, 173, 174).

Urgaonkar, B., G. Pacifici, P. J. Shenoy, M. Spreitzer, and A. N. Tantawi (2007). "Analytic modeling of multitier Internet applications". In: *ACM Transactions on the Web* 1.1 (see pages 46, 58, 107, 108, 115, 118).

Urgaonkar, B., P. J. Shenoy, A. Chandra, P. Goyal, and T. Wood (2008). "Agile dynamic provisioning of multi-tier Internet applications". In: *TAAS* 3.1 (see pages 5, 51, 52).

Villela, D. A., P. Pradhan, and D. Rubenstein (2004). "Provisioning servers in the application tier for e-commerce systems". In: *Quality of Service - IWQoS 2004, 12th International Workshop, Montreal, Canada, June 7-9, 2004, Proceedings*, pp. 57–66 (see pages 5, 51, 52).

VMware, Inc. (2013). *vSphere API and SDK Documentation*. Website. Online available at `https://pubs.vmware.com/vsphere-55/index.jsp`. Last accessed on 04-02-2017 (see pages 155, 183–185).

– (May 2014). *Performance Best Practices for VMware vSphere 5.5*. Website. Online available at `https://www.vmware.com/pdf/Perf_Best_Practices_vSphere5.5.pdf`. Last accessed on 12/03/2016. (see page 209).

Voelter, M. et al. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org (see page 18).

Vögele, C., A. van Hoorn, and H. Krcmar (2015). "Automatic Extraction of Session-Based Workload Specifications for Architecture-Level Performance Models". In: *Proceedings of the 4th International Workshop on Large-Scale Testing, LT'15, Austin, TX, USA, February 1, 2015*, pp. 5–8 (see page 57).

Waldspurger, C. A. (2002). "Memory Resource Management in VMware ESX Server". In: *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002* (see page 168).

Walter, J. (2015). Website. Online available at `http://descartes.tools/pmx/`. Last accessed on 23-05-2016 (see pages 91, 93).

Walter, J., A. van Hoorn, H. Koziolek, D. Okanovic, and S. Kounev (Mar. 12, 2016). "Asking "What?", Automating the "How?": The Vision of Declarative

Performance Engineering". In: *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*. Delft, the Netherlands (see page 231).

Wang, W., X. Huang, X. Qin, W. Zhang, J. Wei, and H. Zhong (2012). "Application-Level CPU Consumption Estimation: Towards Performance Isolation of Multi-tenancy Web Applications". In: *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing, CLOUD*, pp. 439–446 (see pages 58, 108–110, 115, 118, 124, 147, 194).

Wang, W., X. Huang, Y. Song, W. Zhang, J. Wei, H. Zhong, and T. Huang (2011). "A Statistical Approach for Estimating CPU Consumption in Shared Java Middleware Server". In: *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference, COMPSAC*, pp. 541–546 (see pages 108, 115, 118).

Wang, W. and G. Casale (2013). "Bayesian Service Demand Estimation Using Gibbs Sampling". In: *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS*, pp. 567–576 (see pages 59, 108, 112, 113, 115, 116, 118, 123, 136).

Wang, W., J. F. Pérez, and G. Casale (2015). "Filling the gap: a tool to automate parameter estimation for software performance models". In: *Proceedings of the 1st International Workshop on Quality-Aware DevOps, QUDOS 2015, Bergamo, Italy, September 1, 2015*, pp. 31–32 (see page 93).

Wang, Y. and R. J. T. Morris (1985). "Load Sharing in Distributed Systems". In: *IEEE Trans. Computers* 34.3, pp. 204–217 (see page 48).

Westermann, D., J. Happe, R. Krebs, and R. Farahbod (2012). "Automated inference of goal-oriented performance prediction functions". In: *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pp. 190–199 (see pages 54, 93).

Westfechtel, B. (2014). "Merging of EMF models - Formal foundations". In: *Software and System Modeling* 13.2, pp. 757–788 (see pages 95, 96).

Wynter, L., C. H. Xia, and F. Zhang (2004). "Parameter inference of queueing models for IT systems using end-to-end measurements". In: *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS*, pp. 408–409 (see pages 108, 110, 115, 116, 118).

Xu, J., M. Zhao, J. A. B. Fortes, R. Carpenter, and M. S. Yousif (2008). "Autonomic resource management in virtualized data centers using fuzzy logic-based approaches". In: *Cluster Computing* 11.3, pp. 213–227 (see pages 46, 50).

Xue, J., F. Yan, R. Birke, L. Y. Chen, T. Scherer, and E. Smirni (2015). "PRACTISE: Robust prediction of data center time series". In: *11th International Conference*

on Network and Service Management, CNSM 2015, Barcelona, Spain, November 9-13, 2015, pp. 126–134 (see page 45).

Yazdanov, L. and C. Fetzer (2012). "Vertical Scaling for Prioritized VMs Provisioning". In: *2012 Second International Conference on Cloud and Green Computing, CGC 2012, Xiangtan, Hunan, China, November 1-3, 2012*, pp. 118–125 (see pages 10, 50).

– (2013). "VScaler: Autonomic Virtual Machine Scaling". In: *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pp. 212–219 (see pages 5, 10, 49).

Zhang, L., C. H. Xia, M. S. Squillante, and W. N. M. III (2002). "Workload Service Requirements Analysis: A Queueing Network Optimization Approach". In: *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*, pp. 23–32 (see pages 58, 108, 115–118, 137).

Zhang, L., B. Zhang, C. Pahl, L. Xu, and Z. Zhu (2013). "Personalized Quality Prediction for Dynamic Service Management Based on Invocation Patterns". In: *Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*, pp. 84–98 (see page 55).

Zhang, Q., L. Cherkasova, and E. Smirni (2007). "A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications". In: *Proceedings of the Fourth International Conference on Autonomic Computing, ICAC*, p. 27 (see pages 46, 108, 109, 115, 118, 126).

Zhang, S., H. Wu, W. Wang, B. Yang, P. Liu, and A. V. Vasilakos (2011). "Distributed workload and response time management for web applications". In: *7th International Conference on Network and Service Management, CNSM 2011, Paris, France, October 24-28, 2011*, pp. 1–9 (see page 48).

Zheng, T., C. M. Woodside, and M. Litoiu (2008). "Performance Model Estimation and Tracking Using Optimal Filters". In: *IEEE Trans. Software Eng.* 34.3, pp. 391–406 (see pages 58, 108–110, 115, 118, 121, 124, 147, 148, 202, 205).

Zheng, T., J. Yang, C. M. Woodside, M. Litoiu, and G. Iszlai (2005). "Tracking time-varying parameters in software systems with extended Kalman filters". In: *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative Research, CASCON*, pp. 334–345 (see pages 108, 115, 118, 121, 160).

Zhu, X. et al. (2009). "1000 islands: an integrated approach to resource management for virtualized data centers". In: *Cluster Computing* 12.1, pp. 45–57 (see page 47).