TECHNISCHE UNIVERSITÄT MÜNCHEN

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

Automatic Modeling and Simulating the Performance of Big Data Applications

Johannes Kroß

TECHNISCHE UNIVERSITÄT MÜNCHEN

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

Automatic Modeling and Simulating the Performance of Big Data Applications

Johannes Kroß

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Martin Bichler
Prüfer der Dissertation:	1. Prof. Dr. Helmut Krcmar
	2. Prof. Dr. Alexander Pretschner

Die Dissertation wurde am 24.02.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19.06.2020 angenommen.

Abstract

Problem Big data frameworks enable organizations to process data with high volume, velocity, and variety. Guaranteeing software performance requirements is essential to receive the expected value from big data systems. Evaluating the software performance in terms of response time and resource utilization throughout the software lifecycle involves multiple complications. Measurement-based evaluation approaches are often used in practice but require efforts, expenses, realistic test environments, and test data. Due to the extent of big data systems, it is usually not feasible to examine all different system configurations. In contrast, model-based performance evaluations do not involve these drawbacks and can be already applied very early in the software lifecycle. In the context of big data, existing approaches only support a few performance metrics, require meta-knowledge of frameworks, and need to be created and maintained manually. The goal of this dissertation is to introduce a model-based performance evaluation approach for big data systems and supports its applicability by automatically extracting and simulating models.

Research Method This dissertation follows a design science research methodology to achieve its goals. It uses a development-centered approach to apply and extend existing model-based concepts and approaches in the domain of big data software systems. Artifacts such as prototype systems are iteratively developed and enhanced. Their use is demonstrated by formulating scenarios and evaluating them in controlled experiments for their qualities (i.e., prediction accuracy) using simulations.

Results By the example of Internet of Things use cases, the increasing need of planning and managing the software performance of big data systems is emphasized. Additionally, problems of existing model-based approaches are outlined that work well in other domains but miss relevant features such as modeling distributed and parallel computing. A solution to model and simulate performance characteristics is introduced and evaluated for batch processing and stream processing in upscaling scenarios regarding input data sizes and hardware resources. In order to provide the solution in a tool-agnostic way and independent of technologies, a formalism and corresponding domain-specific language is introduced. Prototypes are presented that automate the extraction of instances of this language at the level of software execution architectures and resource demands, data models, and hardware resources. In order to monitor performance traces with low overhead and being able to derive resource demands, a sampling approach is developed. The resulting automated model-based performance approach showed to deliver accurate prediction results (i.e., response time and utilization of central processing units) for two machine learning applications in upscaling scenarios.

Research Implications Existing model-based approaches focus on predicting the metric response time. Resource demands are not modeled and predicted explicitly, but may only be assumed implicitly. This dissertation introduces a formalism and domain-specific language to describe performance-relevant characteristics and resource demands of big data systems independent of technology. While automated model extractions exists in other domains, only a few approaches exist in the domain of big data. These approaches use machine learning techniques and represent black-box approaches. This dissertation additionally demonstrates an automated white-box approach that presents procedures on how to integrate and combine measurements. Software architectures are extracted and parametric resource demands are derived and linked to software components. As a result, performance models are created on architecture-level that can be simulated and used to predict and evaluate the performance of big data systems.

Practical Implications Performance models are often not used in practice as the creation and usage is complex and tool support is not available. This dissertation contributes to the applicability and usability of model-based performance evaluations. The prototype to automatically extract models removes the need for manual creation and maintenance across software releases. It also enables performance engineers to plan required capacities and analyze the scalability of applications without expensive practical evaluations in test environments. Futhermore, performance models are abstracted from users by introducing a domain-specific language that only contains performance-relevant parameters. In this way, software engineers without meta-knowledge of big data frameworks and expert knowledge in performance models shall be able to apply the automated approach.

Limitations Resource demands are derived based on one initial execution of an application. It is essential that the test data are comparable to production data in order to ensure accurate prediction results. The presented model-based prototypes do not explicitly consider resource demands for disk drives but only implicitly. Intercepting accurate and fine-granular measurements for read and write demands and relating them to software operations was not possible without adding special instrumentation to the used data provider. As the simulation of main memory is not provided by simulation engines or only in a very limited way, demands for allocating and deallocating memory were also not modeled.

Zusammenfassung

Problem Big Data Frameworks ermöglichen es Unternehmen, Daten mit hohem Volumen, hoher Geschwindigkeit und unterschiedlicher Datentypen zu verarbeiten. Die Sicherstellung der Anforderungen an die Softwareperformance ist unerlässlich, um den erwarteten Nutzen und Mehrwert von Big Data Systemen zu generieren. Die Evaluation der Softwareperformance während des gesamten Softwarelebenszyklus hinsichtlich der Reaktionszeit und Ressourcenauslastung ist jedoch mit mehreren Komplikationen verbunden. Messbasierte Evaluationsansätze werden in der Praxis häufig eingesetzt, erfordern aber hohe Aufwände, Kosten, realistische Testumgebungen und Testdaten. Aufgrund des Umfangs und der Komplexität von Big Data Systemen ist es in der Regel nicht möglich, alle verschiedenen Systemkonfigurationen und Testszenarien zu untersuchen. Modellbasierte Evaluationsansätze haben diese Nachteile dagegen nicht und können bereits sehr früh im Softwarelebenszyklus eingesetzt werden. Im Kontext von Big Data Systemen unterstützen bestehende Ansätze jedoch nur wenige Performancemetriken, erfordern Metawissen über Frameworks und müssen manuell erstellt und gepflegt werden. Ziel dieser Arbeit ist es, einen modellbasierten Evaluationsansatz für Big Data System zu entwerfen und dessen Anwendbarkeit durch die automatische Extraktion und Simulation von Modellen zu unterstützen.

Forschungsmethode Diese Dissertation folgt einer designorientierten Forschungsmethode. Sie nutzt einen entwicklungszentrierten Ansatz, um bestehende modellbasierte Konzepte und Ansätze im Bereich Big Data anzuwenden und zu erweitern. Artefakte wie beispielsweise Prototypensysteme werden iterativ entwickelt und kontinuierlich verbessert. Deren Anwendung wird durch die Formulierung von Szenarien und deren Evaluation in kontrollierten Experimenten auf ihre Eigenschaften (wie die Vorhersagegenauigkeit) mittels Simulationen demonstriert.

Ergebnisse An Anwendungsbeispielen im Bereich der Internet der Dinge wird der zunehmende Bedarf für die Planung und das Management der Softwareperformance von Big Data Systemen hervorgehoben. Zusätzlich werden Probleme bestehender modellbasierter Ansätze skizziert, die für Systeme in anderen Domänen gut funktionieren, aber relevante Eigenschaften wie die Modellierung von verteilten und parallelen Datenverarbeitungen nicht abbilden. Ein Ansatz zur Modellierung und Simulation der Softwareperformance wird eingeführt und für die Stapel- und Datenstromverarbeitung in verschiedenen Szenarien hinsichtlich zunehmender Datengrößen und zusätzlicher Hardwareressourcen evaluiert. Um die Lösung tool-agnostisch und technologieunabhängig bereitzustellen, wird ein Formalismus und eine entsprechende domänenspezifische Sprache eingeführt. Es werden Prototypen vorgestellt, die die Extraktion von Modellinstanzen dieser Sprache für den Ausführungsplan und Ressourcenanforderungen der Softwarearchitektur, Datenmodelle und Hardwareressourcen automatisieren. Um Performanceabläufe mit geringem Aufwand zu überwachen und Ressourcenbedarfe ableiten zu können, wird zusätzlich ein Samplingansatz entwickelt. Der automatisierte modellbasierte Performanceevaluierungsansatz lieferte genaue Vorhersageergebnisse (bzgl. der Reaktionszeit und Auslastung von Prozessoren) für zwei Anwendungen des maschinellen Lernens in mehreren Szenarien.

Beitrag zur Forschung Bestehende modellbasierte Ansätze konzentrieren sich auf die Vorhersage der Metrik Antwortzeit. Der Ressourcenbedarf wird nicht explizit modelliert und vorhergesagt, sondern teils nur implizit angenommen. Diese Arbeit stellt einen Formalismus und eine domänenspezifische Sprache vor, um performancerelevante Merkmale und Ressourcenanforderungen von Big Data Systemen unabhängig von deren Technologie zu beschreiben. Während für Bereiche anderer Systeme bereits einige automatisierte Modellextraktionen existieren, gibt es nur wenige Ansätze für Big Data Systeme. Diese Ansätze verwenden dabei maschinelle Lerntechniken und stellen somit Black-Box-Ansätze dar. Diese Dissertation zeigt einen automatisierten White-Box-Ansatz und stellt eine Herangehensweise vor, wie modellbasierte und messbasierte Evaluationsansätze integriert und kombiniert werden können. Softwarearchitekturen werden automatisiert extrahiert und parametrische Ressourcenbedarfe abgeleitet und mit Softwarekomponenten verknüpft. Als Ergebnis werden Performancemodelle auf Architekturebene erstellt, die simuliert und zur Vorhersage und Evaluation der Softwareperformance von Big Data Systemen verwendet werden können.

Beitrag zur Praxis Performancemodelle werden in der Praxis häufig nicht eingesetzt, da die Erstellung und Nutzung komplex ist und keine Toolunterstützung verfügbar ist. Diese Dissertation trägt zur Anwendbarkeit und Verwendbarkeit modellbasierter Performanceevaluationen bei. Der Prototyp zur Extraktion von Modellen automatisiert die manuelle Erstellung und Wartung im Falle von Softwareupdates. Darüber hinaus erlaubt der Ansatz notwendige Kapazitäten zu planen und die Skalierbarkeit von Anwendungen ohne aufwändige praktische Auswertungen in Testumgebungen zu analysieren. Des Weiteren werden Performancemodelle durch die Einführung einer domänenspezifischen Sprache, die nur performance-relevante Parameter enthält, von den Benutzern abstrahiert. Auf diese Weise sollen auch Softwareentwickler ohne Metawissen über Big Data Frameworks und Expertenwissen über Performancemodelle den automatisierten Ansatz anwenden können.

Limitationen Ressourcenbedarfe werden basierend auf einer initialen Ausführung einer Anwendung abgeleitet. Dabei ist es wichtig, dass die Testdaten mit den Produktionsdaten vergleichbar sind, um später genaue Vorhersageergebnisse zu gewährleisten. Daneben berücksichtigen die vorgestellten modellbasierten Prototypen Festplattenzugriffe nicht explizit, sondern nur implizit. Das Messen genauer und feingranularer Messungen für Lese- und Schreibanforderungen und deren Verknüpfung mit Softwareoperationen war nicht möglich, ohne das verwendete Framework zur Datenbereitstellung speziell zu instrumentieren. Da die Simulation des Hauptspeichers nicht oder nur sehr eingeschränkt von Simulationsframeworks unterstützt wird, wurden auch Ressourcenbedarfe für die Allokation und Freigabe von Hauptspeicher nicht modelliert.

Acknowledgement

Throughout this dissertation I have received a great deal of support and inspiration.

I would first like to thank Prof. Dr. Helmut Krcmar, who gave me the the opportunity to pursue this dissertation under his supervision and encouraged me to research on the subject of big data. Thank you for the dedicated support, freedom as well as guidance during this dissertation. I would also like to thank Prof. Dr. Alexander Pretschner for being the second examiner and Prof. Dr. Martin Bichler for taking the chair of the examination board.

My special thank goes to fortiss, in particular, Dr. Harald Rueß and Thomas Vallon, who allowed me to take on responsibilities and gave me this opportunity, the freedom and resources during this thesis and beyond.

I would like to thank Andreas Wolke, who advised my master thesis and substantially contributed to my decision to start in research. Many thanks go to my former and current colleagues at fortiss Christian Vögele, Felix Willnecker and, especially, Andreas Brunnert, who guided me and introduced me to the international performance engineering community, Alexandru Danciu, for his great support also at human level, and Peter Bludau, for a fun and exciting environment to work in.

Finally, my biggest thank goes to my parents for their unlimited support not only during this dissertation but also during my education. I am deeply grateful to my parents, my brothers and, above all, my girlfriend Tina.

Munich, Germany, September 2020 Johannes Kroß

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations and Acronyms

API	Application Programming Interface	12
DAG	Directed Acyclic Graph	9
DSRM	design science research methodology	26
DStream	Discretized Stream	ix
HDFS	Hadoop Distributed File System	18
GB	Gigabyte	17
IT	Information Technology	15
IoT	Internet of Things	1
MB	Megabyte	18
RDD	Resilient Distributed Dataset	8
SQL	Structured Query Language	8
YARN	Yet Another Resource Negotiator	15
ACM	Association for Computing Machinery	27
BISE	Business & Information Systems Engineering	28
ICPE	International Conference on Performance Engineering	31
IEEE	Institute of Electrical and Electronics Engineers	27
APM	application performance management	20
CPU	Central Processing Unit	3
DML	Descartes Modeling Language	24
DSL	Domain-specific Language	1
FCFS	First Come First Served	23
HDD	Hard Disk Drive	23
LQN	Layered Queuing Network	23
PCM	Palladio Component Model	3
PS	Processor Sharing	23
QN	Queuing Network	23
QPN	Queuing Petri Net	23
RDSEFF	Resource Demanding Service Effect Specification	24
RQ	research question	3
RR	Round Robin	23
SPE	software performance engineering	20
UML	Unified Modeling Language	24

Part A

Chapter 1

Introduction

The volume of available data exponentially grows as a result of increasing ubiquity of information and communications technology and the Internet of Things (IoT) (Atzori/Iera/Morabito, 2010; Schermann et al., 2014). The term associated with this huge amount of information is *big data*, which describes data sets that are so large and complex that they require specialized data storage, management, analysis, and visualization technologies (Chen/Chiang/Storey, 2012). Systems that are specialized for such capabilities are called big data systems. A common example represents the Apache Hadoop framework with its distributed file system to store massive amounts of data and its integrated MapReduce data processing to access and analyze this data (Apache Hadoop, 2015).

Big data systems are more and more used for analytical use cases and have to meet real-time requirements. The performance of such systems plays an increasingly important role so depending systems operate efficiently and organizations are able to receive the expected value (Barbierato/Gribaudo/Iacono, 2014). This requires engineers to plan performance requirements and size capacities adequately in advance in order to avoid bottlenecks and optimize system configurations. Practically evaluating the performance is a complex task. Another approach to address these problems are performance models (Becker/Koziolek/Reussner, 2009). They allow to depict performance characteristics of software systems and can be analytically solved and simulated to allow performance predictions.

One of the challenges is to model performance characteristics of big data applications and derive parametric resource demands of each application component. Therefore, this work aims on proposing a Domain-specific Language (DSL) for specifying big data systems and a modeling approach for Apache Spark applications on Apache Hadoop. As creating models requires sophisticated metaknowledge regarding big data frameworks but also modeling and simulation approaches, we additionally introduce a prototype to automatically derive models based on measurements. By adapting model parameters and simulating models, we are able to predict the performance of applications for different scenarios. We evaluate the prediction accuracy of adapted models compared to correspondingly executed applications. Our approach enables architects to evaluate performance metrics such as response time and utilization without expensive practical evaluations in test beds.

1.1 Motivation and Problem Statement

The MapReduce paradigm and the corresponding open source implementation by Apache Hadoop enabled software engineers to process and exploit big data (Dean/Ghemawat, 2008; Apache Hadoop, 2015). By now, there have been multiple frameworks released such as Apache Spark, Apache Storm, and Apache Fink that enhanced the software performance of big data systems and allow for processing different types of applications and workloads, such as graphs, data streams, and machine learning (Zaharia et al., 2016; Marz/Warren, 2015; Carbone et al., 2015). For all application and workload types, the software performance of such frameworks and systems is vital for a successful application (Brunnert et al., 2014).

Evaluating the software performance is a difficult and complex task (Wang/Khan, 2015). Before starting the development of a new software application different technologies may be compared depending on the type of workload in order to select an efficient solution. Before deploying a big data application into production the performance and scalability may also be evaluated with different system configurations for different scenarios such as increased data input and increased hardware resources.

Practical measurement-based approaches such as performance tests are usually expensive. The require realistic test systems and test data as well as expert performance knowledge by software and systems engineers, for instance, to deduce root causes of performance bottlenecks (Brunnert et al., 2015; Jain, 1991). In order to evaluate different configurations and scaling behaviors, multiple test runs must be executed. As the extent of big data systems systems and configurations is immense, usually, only a subset of different settings can be evaluated in a reasonable amount of time. Furthermore, tests commonly run with a reduced amount of data and hardware resources due to their availability and involved costs. Therefore, it is often not possible to draw conclusions about required computing capacities while guaranteeing certain performance requirements (e.g., real-time processing). Consequently, it is also difficult to estimate associated costs. Lastly, practical performance evaluations usually occur late in the development lifecycle resulting in additional efforts in case changes in algorithms and software framework are necessary.

In contrast, model-based approaches provide an alternative to measurement-based approaches. They can be used already at the beginning as well as during software development. By adapting models and using analytical solvers or simulations to predict different performance metrics the can be used for various use cases (Becker/Koziolek/Reussner, 2009; Brosig et al., 2015; Brunnert/Krcmar, 2017). For instance, they allow to proactively optimize system and deployment configurations, evaluate scalability, test software design alternatives, and apply different workloads.

1.2 Research Goal and Research Questions

The goal of this dissertation is to develop a model-based approach in order to predict and evaluate the software performance of big data applications. Existing approaches only consider the response time of applications but not demands for resources (i.e., CPU) and lack of tool support. Creating models manually takes a lot of effort and is error-prone and slow as software systems are complex and continuously evolve (Brunnert et al., 2015). Therefore, the goal of this dissertation is also develop a software solution to automatically extract performance models for big data systems in order to support software and system engineers and to enhance the applicability of model-based performance evaluations.

In order to achieve the research goals, this dissertation tries to answer the following research questions (RQs):

RQ 1: What features must a meta-model support to model the performance of big data systems?

Model-based performance are well understood for classical software systems such as business applications. A variety of different meta-models exist that support to model performance characteristics of software architectures and resource demands of software components. Performance models on architecture-level provide a way to model software components separated from hardware environments and workload descriptions. Such existing models, however, do not support features of big data frameworks such as distributed and parallel computing. In the area of big data, also several model-based approaches have been introduced. However, they do not separate performance models on architecture-level, are specific to one technology and to one processing paradigm, and do not allow for modeling resource demands and predicting resource utilization. This first question focusses on understanding the challenges and features of distributed and parallel systems such as IoT and big data applications. Existing model-based approaches are reviewed and evaluated. On this basis, an existing, widely used meta-model (i.e., the Palladio Component Model (PCM) (Becker/Koziolek/Reussner, 2009)) and its simulation engine is extended to support theses features independent of technologies.

RQ 2: How can the performance of batch and stream applications be modeled and simulated?

This question focuses on applying and evaluating the developed meta-model using example technologies. A concept is developed to depict and transfer all relevant performance characteristics of big data systems. This includes data workload, hardware resources, the execution architecture and execution components of applications. For the latter, resource demands have to be estimated, for instance, for a Central Processing Unit (CPU) in order to be able to predict the CPU utilization. To allow for adapting model parameters and evaluate different system configurations (e.g., increased hardware resources and data workload), it is essential to estimate and model resource demands of execution components and relationships between components with parametric dependencies. In order to evaluate our modeling approach, we apply it on two different processing types, batch and stream processing, using example applications from an open source benchmark suite and assess the prediction accuracy in controlled experiments. Therefore, we create one initial model for an application and adapt model parameters according to different scenarios such as upscaling data workload. Afterwards, we simulate these models and compare the prediction accuracy of response times and resource utilization with measurement results from executed applications with corresponding settings and configurations.

RQ 3: How can performance models and resource demands of big data systems be automatically extracted?

For a software system, the manual creation of a corresponding performance model requires a lot of expert knowledge and effort, especially, as software is continuously subject to changes and evolvements. This usually prevents engineers from using them. In order facilitate the use and applicability of our approach, this question focuses on automating the approach of extracting performance models. We develop a DSL for big data systems and a prototype that uses interfaces of big data frameworks to extract hardware resources, data workload, and execution architectures. The prototype also uses monitoring traces to estimate resource demands and relate these demands to components and intra-component relationships of execution architectures. In order to use monitoring traces, the performance applications must be profiled. As profiling adds a considerable amount of overhead to the software performance of applications, we develop a lightweight profiler that uses a sampling approach to extract stack traces and CPU measurements. In order to use and simulate DSL instances, we transform them to performance models as specified in RQ 2. Similarly, we evaluate the prediction accuracy of our approach for different scenarios using controlled experiment and apply more complex big data applications that we were not able to model by hand.

1.3 Thesis Structure

This dissertation consists of three Parts A, B, and C. Figure 1.1 illustrates the structure of the parts and contents as well as the line of argumentation.

Part A introduces this dissertation. The current chapter 1 describes the problem statement, explains research goals and corresponding RQs, and illustrates the thesis structure. Chapter 2 provides and overview of the conceptual background including related terms, disciplines, methods, and technologies. Chapter 3 outlines the research design, applied research methods, and the included publications.

Part B includes six embedded publications P1 to P6 in chapters 4 to 9. The publications are results from research done by the author as part of this dissertation. Section 3.3 provides a short summary of each publication and contribution to the corresponding RQ.

Part C concludes and discusses this dissertation. Chapter 10 first summarizes the results of the publications and describes assumptions and limitations. Afterwards, it outlines contributions to research and to practice and, finally, provides an outlook of future research.

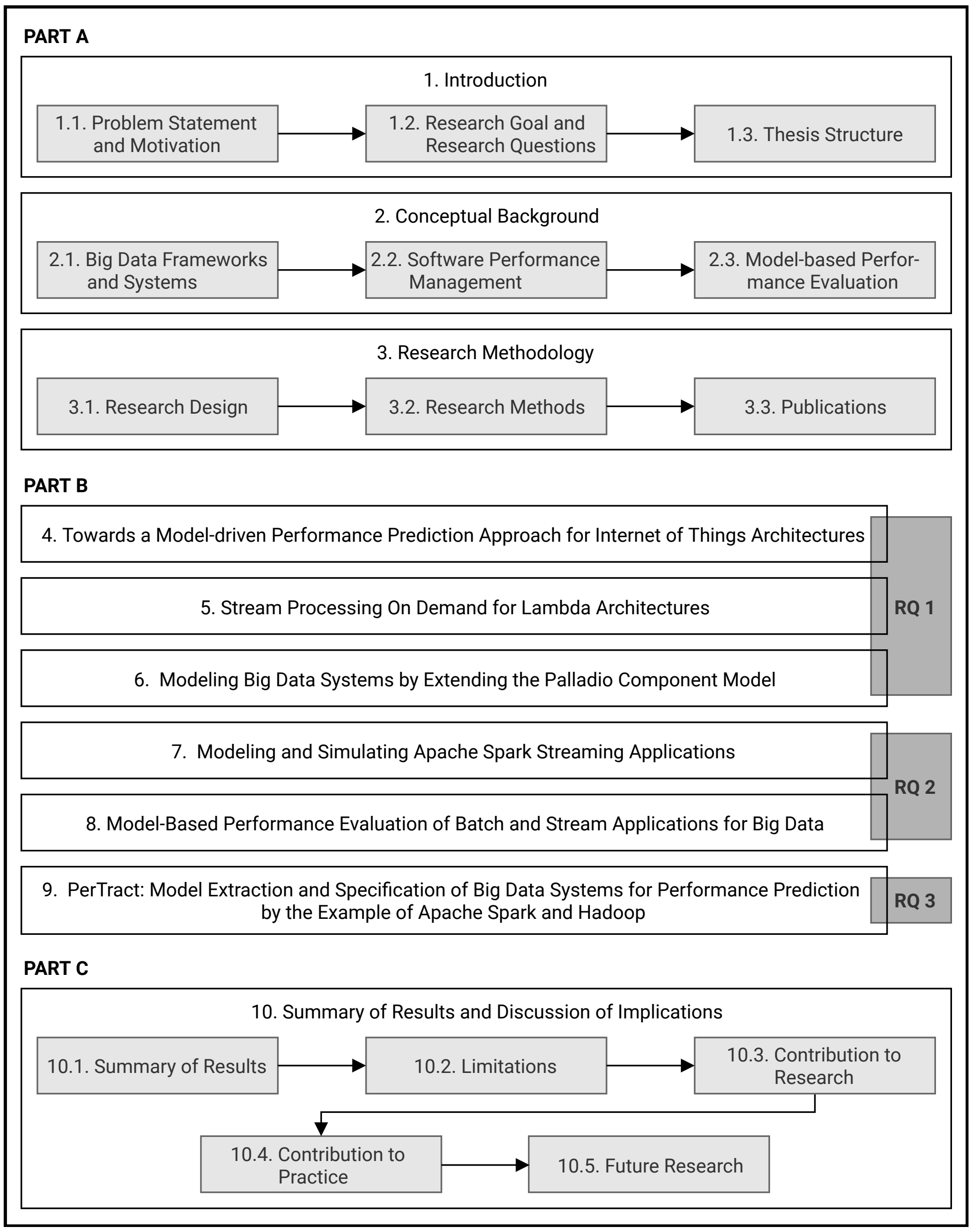

PART A
1. Introduction
1.1. Problem Statement and Motivation
1.2. Research Goal and Research Questions
1.3. Thesis Structure
2. Conceptual Background
2.1. Big Data Frameworks and Systems
2.2. Software Performance Management
2.3. Model-based Performance Evaluation
3. Research Methodology
3.1. Research Design
3.2. Research Methods
3.3. Publications
PART B
4. Towards a Model-driven Performance Prediction Approach for Internet of Things Architectures
5. Stream Processing On Demand for Lambda Architectures
6. Modeling Big Data Systems by Extending the Palladio Component Model
RQ 1
7. Modeling and Simulating Apache Spark Streaming Applications
8. Model-Based Performance Evaluation of Batch and Stream Applications for Big Data
RQ 2
9. PerTract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop
RQ 3
PART C
10. Summary of Results and Discussion of Implications
10.1. Summary of Results
10.2. Limitations
10.3. Contribution to Research
10.4. Contribution to Practice
10.5. Future Research

Figure 1.1: *Structure of this dissertation*

Chapter 2

Conceptual Background

In this section we provide foundations about technologies and approaches on which this thesis builds upon. We will give an overview of current big data systems in section 2.1. This includes different frameworks and process paradigms on application level, but also on resource and infrastructure management level and data and storage management level. Section 2.3 describes two disciplines as part of software performance management. Section 2.3 gives an overview of performance modeling and simulation approaches.

2.1 Big Data Frameworks and Systems

This section presents two kinds of processing types - batch processing (section 2.1.1) and stream processing (section 2.1.2). Afterwards, we illustrate how these applications are managed in cluster resources (section 2.1.3) and access data (section 2.1.4).

2.1.1 Batch Processing

Batch applications process historical data with big volume and tend not to be used for low latency scenarios (Chen/Zhang, 2014). The MapReduce implementation of Apache Hadoop represents one of the first technologies for this purpose. Inspired by this, Apache Spark and, recently, Apache Flink are two additional dedicated frameworks.

MapReduce

Data are often so large that they are required to be distributed across hundreds or even thousands of machines. In order to analyze such data within a certain response time, data processing is similarly aimed at being distributed and parallel. Since distribution data and parallelizing computations, which also involves fault tolerance and load balancing, had to be repetitively dealt with by developers, Google, in particular Dean/Ghemawat (2008), designed and implemented the programming model MapReduce.

Programming and Data Model The idea is similar to and based on constructs from functional programming languages such as Lisp and is described in the following based on Dean/Ghemawat (2008).

As the name suggests, the model consists of the two functions map and reduce.

$$map \qquad (k1, v1) \qquad \rightarrow list(k2, v2) \tag{2.1}$$

The map function receives one key/value pair as input and returns a set of key/value pairs. The MapReduce framework groups values by the same key and invokes the reduce function.

$$reduce \qquad (k2, list(v2)) \qquad \rightarrow list(v2) \tag{2.2}$$

The reduce function receives this set of keys and an associated a set of values for each key and returns a set of result values.

Execution Architecture In order to visualize the flow and understand performance characteristics, we consider the flow of a MapReduce operation as illustrated in figure 2.1.

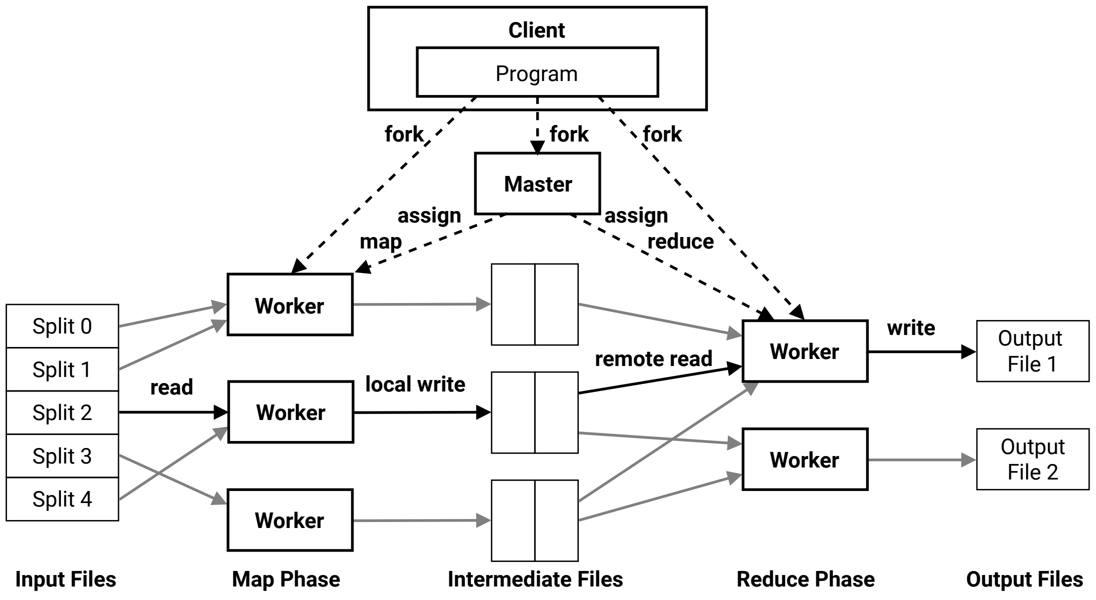


Figure 2.1: *Execution flow of MapReduce (adapted from Dean/Ghemawat, 2008)*

If the big data application (*user program*) is executed, one *master* and multiple *workers* will be created. The *master* distributes tasks to the *workers*, namely, map and reduce tasks, which in turn execute the map and reduce function. Dependent on available resources of each *worker*, tasks will be executed in parallel.

In the *map phase*, one map task will be initiated by the *master* for each *split*. A *split* is a block of an input file. Usually, input files are split into several *splits* and, respectively, blocks (the default size for Apache Hadoop is 128 megabytes). Within each map task, the *worker* reads the *split*, parses the key/value pairs and invokes the map function for each pair (Dean/Ghemawat, 2008). When possible, the *master* aims to only assign map tasks to those *workers* where a replica of the corresponding input *split* is on the local disk of the *worker* available. This shall improve the reading speed as well as reduce network transfers and usage. The output of tasks in the *map phase* is sorted by key and periodically spilled to *intermediate files* on local disks, e.g., after a circular in-memory buffer exceeds a certain threshold for Apache Hadoop (Kroß/Krcmar, 2017). The data locations of these files are reported to the *master*.

In the *reduce phase*, a *worker* receives these locations by the *master* and uses remote method invocations to read the data from other *workers'* disks. Therefore, each worker has its own key space. When a *worker* received all data for its corresponding key space, they are sorted by key and invokes the reduce function of each key and its associated intermediate values. The output is written and appended to output files (Dean/Ghemawat, 2008).

The number of *splits*, disk usage as well as network usage represent limiting factors regarding performance. Therefore, it is important to note that the *map* and *reduce phase* are not necessarily executed sequentially. For Apache Hadoop MapReduce, the *reduce phase* starts after a certain fraction (default 5%) of map tasks has finished (configured by the parameter *mapreduce.job.reduce.slowstart.completedmaps*) (Apache Hadoop, 2015). This involves the advantage of distributing *remote reads* over time and relieve network bandwidth. Furthermore, a so-called combiner function can be implemented, which, in practice, may be equal the reduce function. It will be executed during and directly after the *map phase* to already compute preliminary results values and reduce the amount of data that are remotely read in the *reduce phase* (Dean/Ghemawat, 2008).

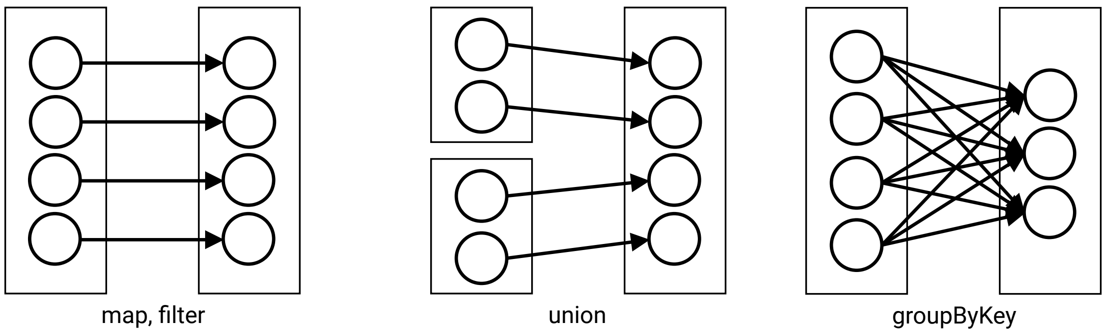
Apache Spark

The emergence of big data has lead to multiple new software frameworks that are specialized for e.g., batch processing, stream processing, and Structured Query Language (SQL) querying. Since big data is highly diverse and messy, pipelines and applications often need to combine such different approaches (Zaharia et al., 2016). Therefore, data and intermediate results must be reused, for instance, in iterative algorithms such as machine learning and graph processing (Zaharia et al., 2012a). However, reusing results for iterative processing, for instance, in MapReduce will require applying multiple applications and, therefore, to always write these results to disk and read them between these applications, which affects response times. For this purpose, specialized add-on frameworks such as Pregel and HaLoop had been developed, but also lack of providing abstractions and interfaces for general reuse (Zaharia et al., 2012a).

Programming and Data Model In order to address these issues, Zaharia et al. (2016) developed Apache Spark to provide a unified engine for distributed data processing. The programming model is similar to MapReduce, but introduces Resilient Distributed Datasets (RDDs) for in-memory data-sharing in order to allow for processing different workloads, which previously required separate engines. An RDD is a read-only, partitioned data collection on which two different types of operations can be applied - transformations and actions (Zaharia et al., 2012a). Transformations create and return new RDDs from either stored data or other RDDs. Examples include map, filter, and join operations. Actions, on the other hand, only return a value to the application or save data to disk. Examples constitute reduce, count, and save operations. Moreover, RDDs can be specified to be persistent and reusable by setting a storage strategy (e.g., in-memory by default). They also always contain necessary information to be reconstructed (including its partitions) from stored data. Therefore, a Directed Acyclic Graph (DAG) is used to track lineage across a wide range of associated transformations and Zaharia et al. (2012a) provided RDDs with an interface that exposes five pieces of information:

- a set of partitions
- meta data about its partitioning scheme
- meta data about its data placement
- a function for computing the dataset based on its predecessors
- a set of RDD dependencies

RDDs are put into reference and Spark distinguishes *narrow dependencies* and *wide dependencies* as illustrated in figure 2.2, where rectangles represent RDDs and circles partitions (Zaharia et al., 2012a). For *narrow dependencies*, each partition of a predecessor RDD is referenced by at most one partition of the successor RDD. For *wide dependencies*, each partition of a predecessor RDD may be references by multiple partitions of the successor RDD so data are shuffled.



(a) *Narrow dependencies* **(b)** *Wide dependencies*

Figure 2.2: *Examples of dependencies of operations in Apache Spark (adapted from Zaharia* et al.*, 2012a)*

Execution Architecture Spark stages operations to execute them in an optimized ways as illustrated in figure 2.3. In addition to figure 2.2, circles are partitions that are already in-memory and dashed rectangles represent stages. Operations that lead to narrow dependencies are pipelined into a *stage* so they will be executed in direct succession on a computer node (i.e., *map*, *union*). New *stages* will be only created for operations that require partition data to be shuffled across nodes and, thus, cause wide dependencies (i.e., *groupBy*, *join*), and for partitions already in memory (i.e., *RDD B*) (Zaharia et al., 2012a).

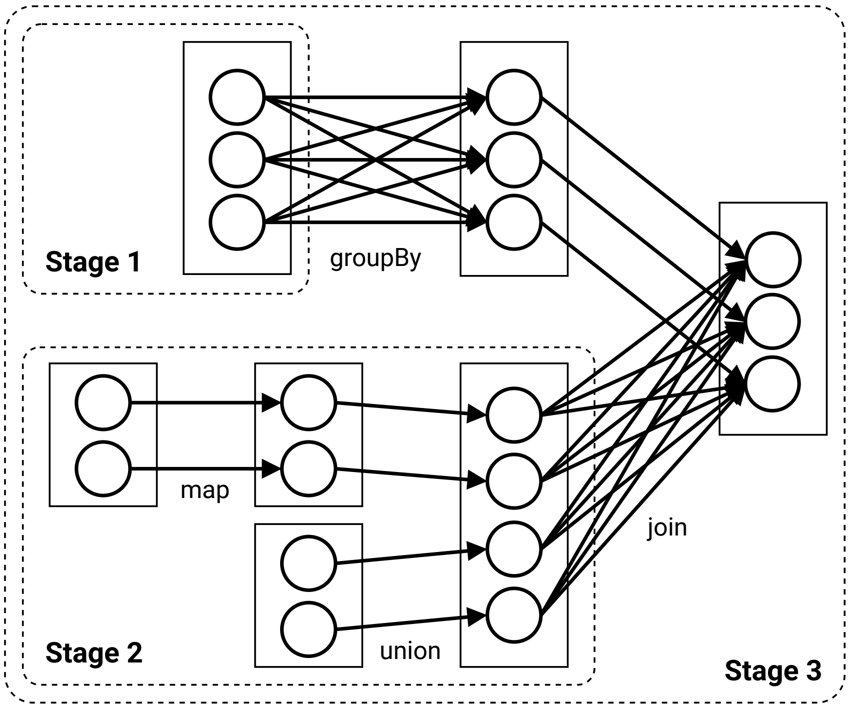


Figure 2.3: *Execution flow of Apache Spark (adapted from Zaharia* et al.*, 2012a)*

The operating principle and execution flow of Apache Spark is illustrated in figure 2.4 and described subsequently based on Kroß/Krcmar (2017). The main process is the *driver program*, which owns a *SparkContext* that orchestrates the application. First, it connects to a *cluster manager* (see section 2.1.3) and allocates *executors* to *worker nodes* of a cluster. *Executors* are processes with its own *cache* to run *tasks* in parallel and are exclusively assigned to one application in order to be isolated from other applications. Second, the *SparkContext* schedules an application's *tasks* on *executors.* An application consists of one or several jobs. Each job exclusively contains stages, which again contain *tasks.* As already mentioned, a DAG is formed based on associated operations that are grouped together into stages of *tasks.* The number of *tasks* of one stage is dependent to the number of RDD partitions. Stages are executed sequentially as well as their parent jobs.

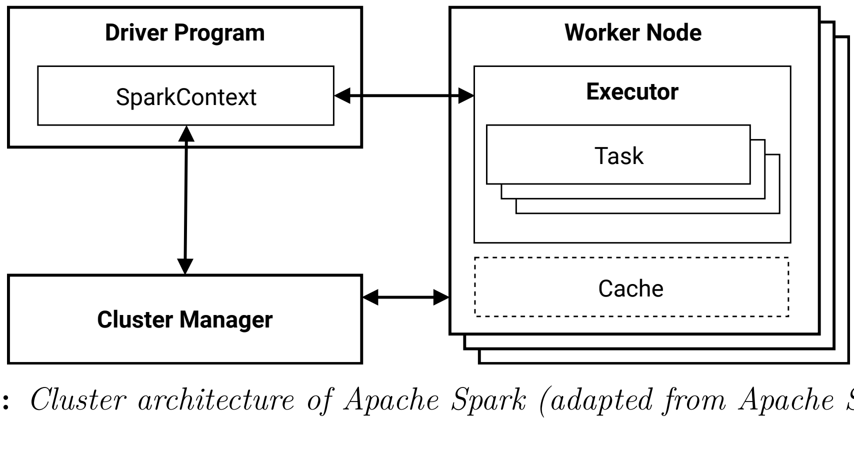


Figure 2.4: *Cluster architecture of Apache Spark (adapted from Apache Spark, 2015)*

2.1.2 Stream Processing

In contrast to batch applications, stream processing systems concentrate to continuously analyze big volumes of live data with low latency (Chen/Zhang, 2014; Kroß/Krcmar, 2017). Therefore, two programming models can be differentiated - one mini-batch model and one operator-based model (Hesse/Lorenz, 2015; Zaharia et al., 2012b). The basic idea of the former paradigm is to divide data streams into mini or micro batches and apply batch processing on those. The latter paradigm instantly processes each record at a time as it enters the system (Hesse/Lorenz, 2015). In the following, we will first describe the Spark Streaming library as part of Apache Spark, which is the only major framework that implements the mini-batch model. Afterwards, two examples are presented that implement the operator-based model - Apache Storm, which historically is one the first sophisticated stream processing systems, and Apache Flink, which is a more recently developed system and involves similar optimizations to Apache Spark.

Spark Streaming

As already mentioned, the motivation of Apache Spark was to design a core engine that supports multiple workload and applications types. Therefore, it includes several extension libraries such as for machine learning (MLlib), graph processing (GraphX), SQL querying, and, data stream processing (Spark Streaming) (Apache Spark, 2015).

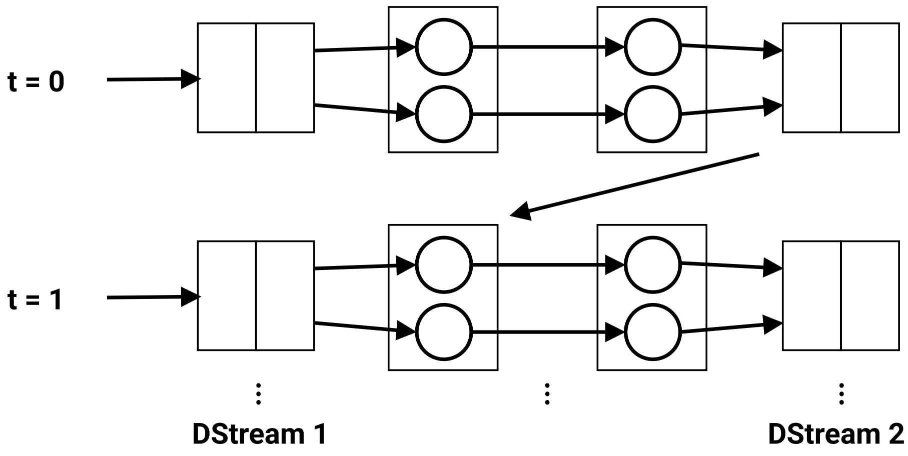


Figure 2.5: *DStream processing model of Apache Spark (adapted from Zaharia* et al.*, 2012b)*

In order to allow for stream processing, the idea is to treat “streaming computations as a series of deterministic batch computations on discrete time intervals” (Zaharia et al., 2012b). Therefore, Zaharia et al. (2012b) introduce a new programming model DStream, which is “a sequence of immutable, partitioned datasets (specifically, RDDs) that can be acted on through deterministic operators” as illustrated in figure 2.5. They are created either from input data streams, which will be split and received in sequential intervals (i.e., *t=1*, *t=2*), or from operations on other DStreams (i.e., *DStream 2* was created by a *batch operation* on *DStream 1*). Moreover, there is one DStream for each series of datasets (Zaharia et al., 2012b). Similar to Apache Spark’s core engine, a DStream is partitioned (2.6) and the number of partitions depends on the number of partitions of the input stream. Furthermore, operations with narrow dependencies will be also grouped together into stages. In similar way to operations on regular RDDs, there are two sets of specialized operations for DStreams - transformations and output operations. The former return new DStreams and are either stateless, where data dependencies are within the same time interval, or stateful, where dependencies to results from prior intervals exist (i.e., windowing, incremental aggregation, and state tracking) (Zaharia et al., 2012b). Output operations, on the other hand, only return data to external systems such as file systems or databases.

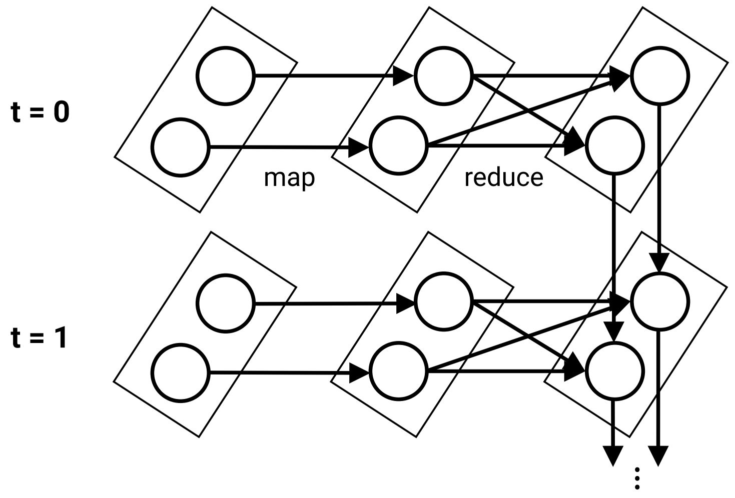


Figure 2.6: *Operations on DStreams and its partitions in Apache Spark (adapted from Zaharia* et al.*, 2012b)*

Since Spark Streaming is an extension library, a developed application will use the cluster architecture (figure 2.4) in the same way and is also orchestrated by one *SparkContext*. From a concept perspective, however, Apache Spark will create one job for each DStream, for each interval, whereas only one job is active and others are queued (Kroß/Krcmar, 2017; Apache Spark, 2015).

Apache Storm

In contrast to Spark Streaming and its mini-batch approach, Apache Storm represents one of the early stream processing systems and uses an operator-based model (Hesse/Lorenz, 2015). Nevertheless, it is similarly designed to be horizontally scalable, resilient, extensible, efficient, and easy to administer. These characteristics include adding and removing node to / from a cluster during operations, handling fault such as hardware failures, invoking external interfaces, and reusing data structures in-memory (Toshniwal et al., 2014). Apache Storm was created by Nathan Marz who also proposed the lambda architecture (Toshniwal et al., 2014; Marz/Warren, 2015).

Programming and Data Model Storm processes tuples of data streams that run through so-called topologies, which are defined in detail as follows:

> "A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components. Vertices are further divided into two disjoint sets – spouts and bolts. Spouts

are tuple sources for the topology. Typical spouts pull data from queues, [. . .] bolts process the incoming tuples and pass them to the next set of bolts downstream" (Toshniwal et al., 2014).

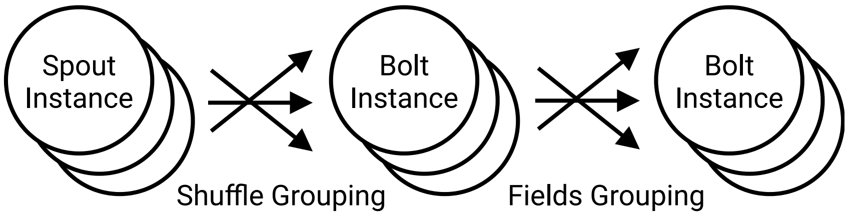


Figure 2.7: *Execution flow of a Apache Storm topology (adapted from Toshniwal* et al.*, 2014)*

In contrast to DAGs in Apache Spark, a topology is not acyclic. An instance for a topology is illustrated in figure 2.7 which is based on (Toshniwal et al., 2014) and incrementally counts the number of words. It contains a *TweetSpout* that receives or pulls data tuples from systems (e.g., Apache Kafka) or)Application Programming Interfaces (APIs) (e.g., Twitter) and continuously forwards them to connected bolts. The *ParseTweetBolt* parses words from a text. which is contained in the initial tuple, and emits a new tuple for each word containing the word itself as well as the count (i.e., 1). This procedure is very similar to the *map* operation and, respectively, phase of MapReduce. Analogically to the *reduce* phase, the *WordCountBolt* receives each tuple, sums up the counts for each word, and outputs the results to an external system. The latter may also be performed in time intervals and an internal counter will be reset (Toshniwal et al., 2014).

Execution Architecture Apache Storm also involves a master worker architecture. As illustrated in figure 2.8, the master is called *Nimbus*. It receives applications, orchestrates the execution on worker nodes, and maintains the cluster state via Apache *ZooKeeper* (Hunt et al., 2010), a dedicated service for coordinating processes of distributed applications . Each worker nodes runs one *Supervisor* for communication and one or several worker processes for each application.

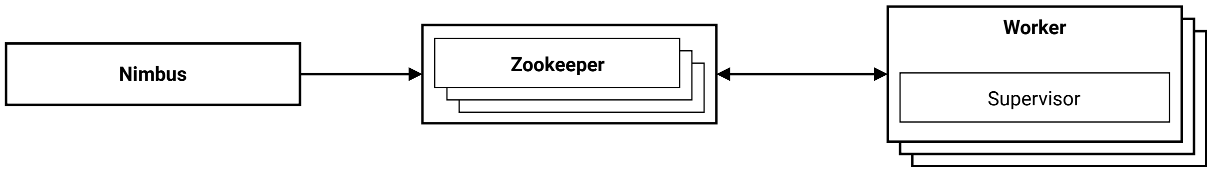


Figure 2.8: *Cluster architecture of Apache Storm (adapted from Toshniwal* et al.*, 2014)*

Furthermore, each worker process is exclusively dedicated to one application and may execute different tasks of the topology of an application. Therefore, a task is represents the operations of a bolt or a spout. Although tasks provide intra-bolt/intra-spout parallelism and executors intra-topology parallelism, Apache Storm does not involve a similar concept to group operators with narrow dependencies and reduce the number of threads and shuffles like Apache Spark (Kroß/Krcmar, 2017; Toshniwal et al., 2014). Data tuples are shuffled between spouts and bolts as illustrated in figure 2.7 and different partitioning strategies are provided (i.e., *Shuffle Grouping*, *Fields Grouping*).

Apache Flink

Apache Flink is an open source framework for processing data batches and stream that uses a single execution model in order to cover different big data applications such as continuous data pipelines, machine learning, and graph processing (Carbone et al., 2015). We present Apache Flink only in section 2.1.2 *Stream Processing* since stream processing is at its core (Carbone et al., 2015).

Programming and Data Model The core runtime uses distributed streaming dataflows as abstractions in the programming model and the execution engine. Apache Flink does not distinguish between analyzing events in a continuous way and analyzing historical data. For batch scenarios, data are considered as finite streams where the time of records is ignored. It begins to process data at different points and maintains different types of state (Carbone et al., 2015). Therefore, it includes one dedicated API for batch processing and one for stream processing on top of the core runtime. Similarly to Spark Streaming and Apache Storm, Apache Flink involves streams and operations (transformations) (Apache Flink, 2017a). A stream is a continuous flow of data records. An operation that receives on one ore more streams as input, transforms it, and returns one or more streams as output (Apache Flink, 2017a). Apache Flink also provides three different concepts to refer to time (Carbone et al., 2015):

- **processing time** - the time an event was processed
- **event time** - the time an event emerged
- **ingestion time** - the time an event entered Apache Flink

As mentioned, distributed streaming dataflows are also used by the execution engine. Therefore, a runtime program is a DAG of stateful operators such as filter, join and window functions (e.g., *SRC1*) as vertices and intermediate data streams as edges as illustrated in figure 2.9 (Carbone et al., 2015). Similar to other frameworks, the DAG will be executed in a parallel way. Streams are constituted of several partitions. Stateful operators represent tasks and are executed in subtasks in parallel where one subtask will be created for each partition. There are two types of intermediate streams - pipelined and blocking streams (Carbone et al., 2015). The former (e.g., *IS1*, *IS2*) is used to exchange data in parallel between operators and allow for a pipeline execution. In comparison to Spark Streaming, results are not stages, but pipelined results can be directly processed by successor operators (e.g., from *SRC1* to *OP1*).In contrast and similar to Spark Streaming, the blocking streams (e.g., *IS3*) buffer all data between two operators that are executed in different stages (e.g., *OP1*, *SNK1*). This allocates more memory and data may have to be spilled to disk. In order to avoid this performance decrease as well as materialization, Apache Flink prefers pipelined streams for streaming applications (Carbone et al., 2015). Furthermore, figure 2.9 illustrates the transmission of *Control Events*. These are produced by operators and are used to coordinate checkpoints (checkpoint barriers), to progress event time (watermarks), and to indicate the end of an iteration (iteration barriers) (Carbone et al., 2015). For iterations, Apache Flink includes dedicated operators that are composed of an execution graph.

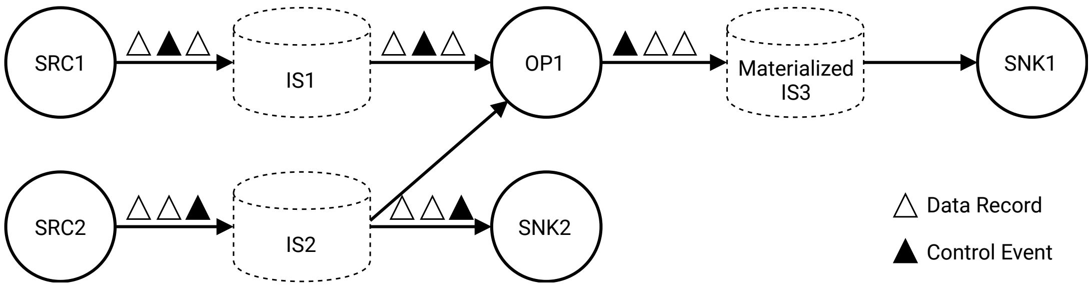


Figure 2.9: *Dataflow graph of Apache Flink (adapted from Carbone* et al.*, 2015)*

Execution Architecture Similar to already mentioned frameworks, Apache Flink follows a master worker architecture as illustrated in figure 2.10. In order to execute an application, a *Flink Client* initially creates a dataflow graph based on the application code and sends it to the *Job Manager*. The *Job Manager* serves as master and orchestrates the execution. This includes to schedule operators, to monitor its states as well as the state of streams, and to manage checkpoints and recoveries. The execution is conducted on one or several *Task Managers* that represent worker nodes. Therefore, each *Task Manager* contains *Task Slots* to execute the operators through subtasks. A *Task Slot* isolated subtasks and has a fixed amount of memory reserved that is available for them (Apache Flink, 2017b).

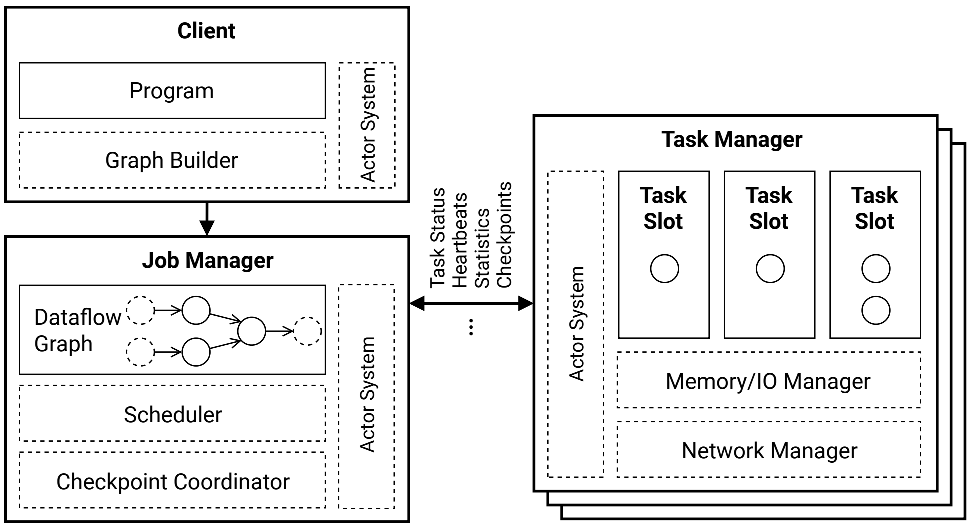


Figure 2.10: *Cluster architecture of Apache Flink (adapted from Carbone* et al.*, 2015)*

2.1.3 Resource Management

Although some of the presented big data software frameworks are able to run in a standalone cluster, for productive deployments and Information Technology (IT) operations usually a dedicated resource management technology is used. This implicated several advantages such as to operate a cluster of computer nodes that is able to support and run applications that implement different software frameworks, dynamically manages overall resources, isolates applications, and provides special recovery mechanism (Vavilapalli et al., 2013). Subsequently, two technologies are presented as examples.

Apache YARN

At the beginning of Apache Hadoop, the main focus of its design was MapReduce and the programming model was coupled with the management of resources (Vavilapalli et al., 2013). Apache Yet Another Resource Negotiator (YARN) is the result of decoupling both parts in order to allow for supporting and executing different programming models and applications. Therefore, it includes the concept of an application master (*AM*) as part of an application. This application master is a process run by YARN. It orchestrates the application's execution flow, programming model, and task fault tolerance and, for this purpose, requests required resources.

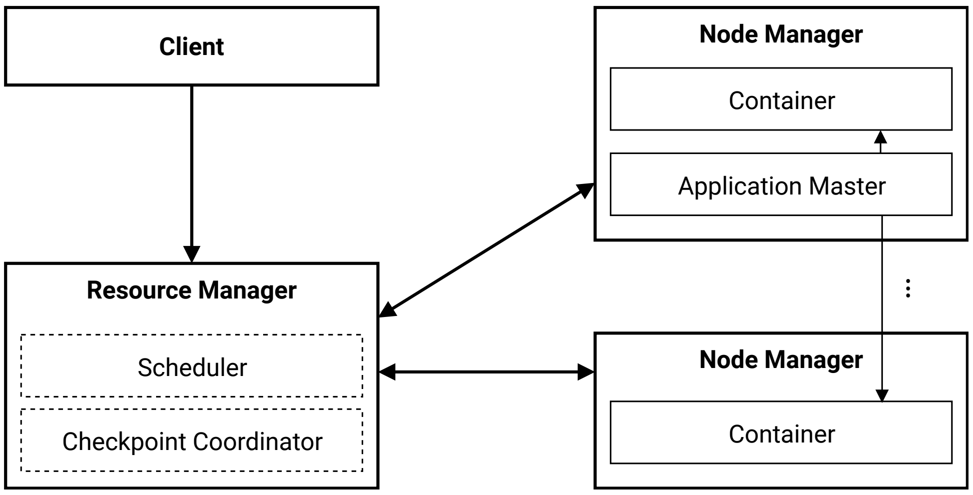


Figure 2.11: *Architecture of Apache YARN (adapted from Vavilapalli* et al.*, 2013)*

YARN follows a master worker architecture as illustrated in figure 2.11. The *Resource Manager* represents the master node. It receives applications by *clients* and forwards each one to a *Scheduler*. The *Scheduler* dynamically allocates the requested resources, so-called *containers*, by the application master with due regard to fairness, capacity, and (data) locality (Vavilapalli et al., 2013). A container is an abstract notion for resources such as memory and CPU cores and runs tasks on a assigned node (Kroß/Krcmar, 2017). In particular, the application master itself runs inside a container and its requests include the amount of containers as well as the resources for each container. Afterwards, the *Scheduler* allocates and spawns the containers on *Node Managers (NM)* that represent worker nodes. Therefore, the *Resource Manager* is connected to them to monitor the cluster state and access resources. Each *Node Manager* is responsible for managing the execution of its local containers, reporting their state, and monitoring and providing resource information (Kroß/Krcmar, 2017).

Hadoop MapReduce in YARN The Hadoop implementation of MapReduce includes a MapReduce application master (*MRAppMaster*) (Apache Hadoop, 2017b). It requests one container for each map and one container for reduce task and schedules as well as executes each task inside the container. Afterwards, completed containers will be reported. The client that submits the user program is required to be attached the running application.

Apache Spark in YARN Applications will launch an application master that requests resources for executors, and each executor will run in an own resource container (Kroß/Krcmar, 2017). Therefore, Apache Spark can be executed in two different modes on YARN - cluster mode and client mode. For the former, the *driver program* and *SparkContext* run inside the application master on the cluster (Apache Spark, 2017). For the latter, the *driver program* and *SparkContext* run at the client and the client is attached to the application for its entire lifetime (Kroß/Krcmar, 2017).

Apache Storm in YARN Applications can be executed using third-party extensions such as Slider [1], an Apache incubator project, or an extension by Yahoo Inc.[2]. Therefore, *Nimbus* will run in the application master container and each *Supervisor* in an own container. Usually, ZooKeeper will not be executed inside any YARN container, but is required to be operated externally.

Apache Flink in YARN In order to run one or more applications, first a session in YARN must be created where the Apache Flink cluster will run (Apache Flink, 2017c). One container for *Job Manager* and one container for each *Task Manager* will be initially created for the session and applications will be submitted to this session. This implies that once the session is created, computing resources are static and Apache Flink will schedule applications on those. Similarly to Apache Spark in YARN, there are two ways to start the session in YARN - either attached, where the client must be attached across the lifetime, or detached. Whereas both approaches require to start a cluster first, it is also possible to only run one Apache Flink application on YARN without starting a cluster (Apache Flink, 2017c). For this approach, however, the client must be attached to application for its lifetime.

Apache Mesos

Apache Mesos was developed at a time where YARN did not exist yet, but was coupled within the fist Hadoop version of MapReduce. Therefore, Hadoop was only able to run applications from the Hadoop ecosystem. In case software engineers developed an application using a different framework, it could not be executed in the Hadoop cluster. This limitation and problem was one of the reasons that lead to the development of Apache Mesos. It is a platform in order use and share a cluster of resources in a fine-grained manner (e.g., regarding data locality) for multiple diverse big data frameworks (Hindman et al., 2011). Therefore, a two-level scheduling abstraction called resource offers is introduced (Hindman et al., 2011). First, Mesos offers a certain bundle of resources to the software framework of an application based on a scheduling policy. Second, the frameworks decides which resources it will accept. In this way, Mesos delegates the task scheduling und execution to the framework itself.

Mesos uses a master worker architecture that is illustrated in figure 2.12. The *Mesos master* involves an allocation module, which implements the mentioned resource offer mechanism, and orchestrates the *slave* and, respectively, worker nodes, in which the tasks of various big data frameworks run. Each framework has to include a scheduler component and a executor component that is executed on the worker nodes. The scheduler accepts

[1]https://github.com/apache/incubator-slider
[2]https://github.com/yahoo/storm-yarn

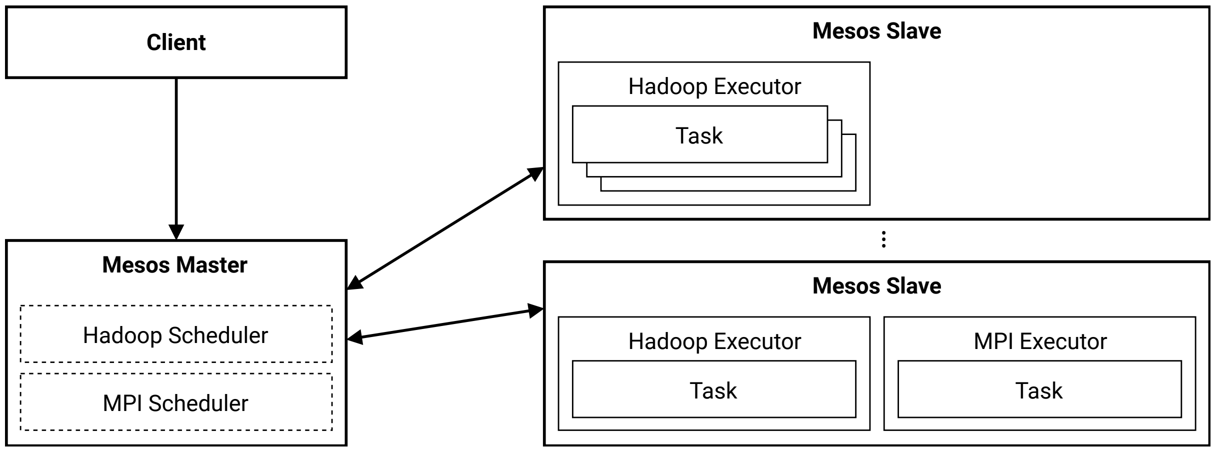


Figure 2.12: *Architecture of Apache Mesos (adapted from Hindman* et al., *2011)*

certain resource offers (e.g., 3 CPU cores and 15 Gigabyte (GB) memory) and reports the tasks that shall be executed as well as the resources for each task to the *Mesos master*. The master then sends the tasks to the *slave* nodes and allocated the resources to the executor component.

2.1.4 Data and Storage Management

Independent on batch and stream frameworks, applications require access to data sinks. There are several frameworks specialized on either batch or stream purposes since theses include significantly different requirements. In the following, one example for each type will be presented. Note that there are also a few approaches in industry (e.g., the Kappa architecture (Kreps, 2014)) where only one data sink technology is used for serving batch as well as stream applications.

Apache HDFS

As part of the Apache Hadoop ecosystem, the Hadoop Distributed File System (HDFS) is developed to provide applications data access with high availability, reliability and horizontal scalability (Kroß/Krcmar, 2017; Shvachko et al., 2010). Similarly, it implements a master worker architecture and requires a special client to access data. As other filesystems, HDFS is able to create, read, and delete files as well as directories.

NameNode It represents the master node and manages the entire metadata including the so-called namespace. It is a tree of files and directories and contains data information such as per permissions, modifications, and access times (Shvachko et al., 2010). The metadata also include the mappings to the actual data locations on the DataNodes and are kept in memory as an image. Furthermore, the NameNode handles data access by clients and manages renaming, mapping, storing, reading and replicating data on DataNodes (Apache Hadoop, 2017a). Usually, there is only one NameNode in a cluster. Since it can be the single point of failure, the current Apache Hadoop implementation also contains standby concepts with redundant NameNodes in order to ensure high availability (Apache Hadoop, 2017a; Shvachko et al., 2010).

DataNodes They are the worker nodes and responsible for storing data and, more specifically, data blocks as delegated by the NameNode. In HDFS, files are split into blocks (128 Megabyte (MB) by default) and replicated at multiple DataNodes to allow for reliability (Shvachko et al., 2010). Two files are created for each block to store the data content as well as the metadata. Therefore, DataNodes are invoked by the NameNode to replicate blocks to other nodes, remove blocks, shut the entire DataNode down, or to send a report of all blocks' information, which is used for load balancing (Shvachko et al., 2010). To communicate and execute these remote invocations, heartbeats are used that also involve the purpose to report the service availabilities and ensure integrity in the cluster. Furthermore, nodes perform a handshake if they register with the cluster and exchange a shared, unique namespace identifier across the cluster and a identifier of the DataNode (Shvachko et al., 2010).

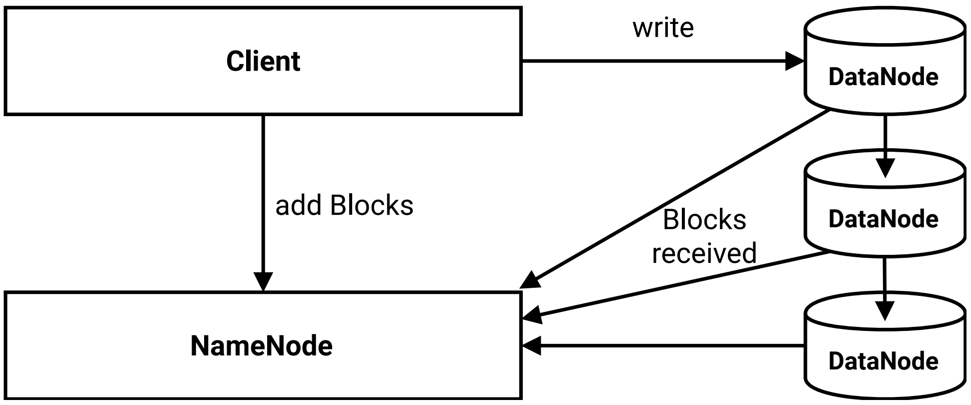


Figure 2.13: *Architecture of Apache HDFS (adapted from Shvachko* et al.*, 2010)*

Users access the paths in the namespace in order to refer to directories and files using the HDFS client. Figure 2.13 shows a basic interaction of the client with a HDFS cluster (Shvachko et al., 2010). In case of writing a file, the client indicates it to the NameNode and receives a list of DataNodes that will store the first block of the file. It then creates a sequential pipeline among the according DataNodes and directly sends the block data to the pipeline where it will be forwarded at each node (White, 2015, pp. 72-73). Each DataNode also acknowledges the finished written data block at the NameNode. Afterwards, the client receives a new list of DataNodes for the next block and this process repeats until all blocks have been sent successfully. In case of reading a file, the client also indicates it to the NameNode and receives a list of DataNodes that contain a replica of the blocks of the file and directly accesses it at each DataNode (Shvachko et al., 2010).

Apache Kafka

HDFS is intended and usually used as incoming data source for batch applications. In contrast, stream applications usually exploit messaging systems or brokers to receive data as soon as they are created in a stream (Kroß/Krcmar, 2017). Apache Kafka represents a widely used example for such as system that is built for scalability and high throughput and, in contrast, uses a peer-to-peer architecture (Apache Kafka, 2015). The basic architecture and concepts of Apache Kafka is illustrated in figure 2.14 and will be explained in the following based on Kreps/Narkhedem/Rao (2011).

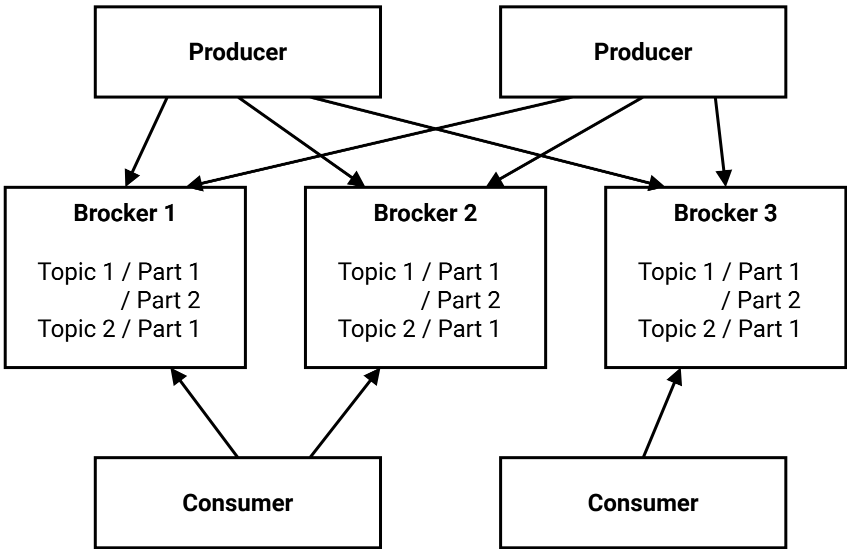


Figure 2.14: *Architecture of Apache Kafka (adapted from Kreps/Narkhedem/Rao, 2011)*

Topics They represent a data stream of messages of a certain type. Messages are created and, respectively, published by *producers*, while they are pulled and, respectively, subscribed by *consumers*. *Topics* are stored on servers and, therefore, split into partitions where each topic constitutes a logical log and each message is a logical offset in the log that only consist of a payload of bytes. As a result, there are not any identifiers and index structures used.

Producers They publish messages to a *topic* either one by one or as as set per request. Therefore, the method of data serialization can be customized. Furthermore, messages can be published to a random partition of a *topic* or certain partition based on a key or function.

Consumers One or several *consumers* belong to a consumer group and subscribe to a *topic* by creating one or several sub-streams in which the messages are distributed and can be concurrently received. Each consumer group subscribed to a certain *topic* receives all messages, but each message is only published to one *consumer* of its parent group. *Consumers* provide the offset when pulling messages and also acknowledge the latest read message offset. Furthermore, messages from one partition are guaranteed to be received in sequential order by Apache Kafka in contrast to messages from multiple partitions.

Broker An Apache Kafka cluster usually consists of several nodes on which topics and their partitions are distributed and replicated. They are called *brokers* and are always stateless so they do not know what messages have been received by each *consumer*. Furthermore, the cluster does not follow a master worker architecture, but coordinates itself decentralized. This involves maintaining a registry for *brokers* as well as for *consumers* to handle additions and removals. Furthermore, it includes an ownership registry to relate *consumers* to partitions and an offset registry for already consumed messages of partitions. The entire coordination is accomplished using Apache ZooKeeper.

2.2 Software Performance Management

As a basis for the performance evaluation approaches introduced in this dissertation, we use and integrate activities from software performance engineering (SPE) and application performance management (APM). This section provides an overview of both areas.

2.2.1 Software Performance Engineering

SPE can be defined as a systematic, quantitative approach used throughput the software development cycle in order to meet performance requirements (Smith, 2007; Woodside/Franks/Petriu, 2007). Different views exist on what approaches and activities SPE comprises. For instance, Smith (2007) emphasizes mainly model-based approaches to predict the performance of a software system early in the development cycle. In contrast, Woodside/Franks/Petriu (2007) additionally consider measurement-based approaches to apply testing late in the development cycle when a software prototype can already be run and measured. Throughout all different views, SPE involves a set of different activities that are described in the following subsection based on Smith (2007) and Woodside/Franks/Petriu (2007).

One key activity is *defining and clarifying performance requirements*. For instance, requirements can be collected regarding the response time (as experienced by system users and for single system components), throughput, and hardware resources. Performance requirements may be also refined and clarified, for instance, based on predictions.

Another SPE activity is *identifying concerns and scenarios*. Concerns may be essential system operations and factors affecting the software performance. Scenarios describe different use cases of a system that may invoke such system operations (e.g., worst-case scenarios).

Based on the concerns and scenarios, *predicting the performance* using model-based approaches (see section 2.3) is a key activity. Prediction results help to evaluate design alternatives, understand scalability of architectures, and identify critical parts of a software system.

Another key activity represents *performance testing*. It is a measurement-based approach and evaluates the execution a software system under certain load with test data. Performance testing may also be the basis to derive resource demands for model-based approaches. It should be also used to verify model specifications and validate prediction results. Vice very, model-based approaches can support in designing performance tests and at software evolution.

2.2.2 Application Performance Management

APM can be defined as collection of activities by organizations in order to ensure that the performance of applications meets the business goals during operations (Menascė, 2002). Menascė/Almeida (2002) defines APM in the context of web-based applications, which we apply to big data applications. To the best of our knowledge, there has not been a definition specifically related to big data software systems introduced, but rather how big data technologies can be used to implement APM (e.g., Rabl et al. (2012)).

APM can be implemented in a reactive way and in a proactive way Menascė (2002). In reactive way, the application performance is monitored and it will be reacted if problems occur and requirements are not met. In the other way, proactive processes are applied to reduce and prevent the occurrence of problems. For both ways, performance monitoring forms the basis to derive measurements that are analyzed for these purposes.

Monitoring can be accomplished by using event-driven and sampling-based techniques (Brunnert et al., 2015; Menascė/Almeida, 2002; Lilja, 2005). The former approach captures a measurement if an event occurs (e.g., invocation of a method), the latter captures a measurement in intervals (e.g., every second)(Brunnert et al., 2015).

Monitoring can be used to retrieve measurements on different levels of granularity for metrics such as response times (and availability), throughput, and resource utilization. Therefore, monitoring can take place at hardware-level (e.g., CPU, disk drives, and memory) as well as at software-level. At software-level, software is usually instrumented. Instrumentation can be achieved statically at design time and dynamically at runtime (Brunnert et al., 2015). Techniques include code modifications, compiler modifications, and middleware interception (Brunnert et al., 2015; Jain, 1991; Lilja, 2005; Menascė/Almeida, 2002).

2.3 Model-based Performance Evaluation

By the example of an architecture-level performance model, figure 2.15 illustrates the procedure of a typical model-based performance evaluation. The software is modeled by depicting the structure and components of the software architecture and the behavior and resource demands of software components. The workload is modeled by specifying the usage of the system. The hardware is modeled by depicting resources and their capacities. In order to predict different performance metrics, these models can be transformed to and solved by analytical models or transformed to be used by simulation engines.

In general, performance models can be differentiated into analytical and architecture-level models (Brunnert et al., 2015). This section presents an overview of both types.

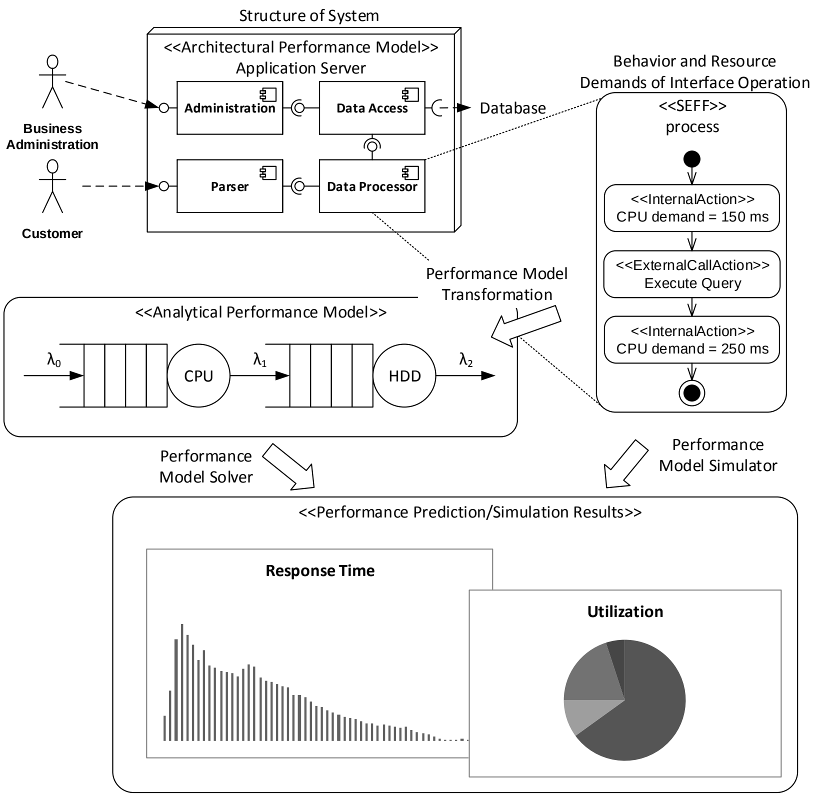


Figure 2.15: *Model-based performance evaluation (Brunnert* et al.*, 2015)*

2.3.1 Analytical Performance Models

Examples for analytical performance models represent Petri nets, Queuing Networks (QNs), Queuing Petri Nets (QPNs) and Layered Queuing Networks (LQNs) (Brunnert et al., 2015). In the following, we concentrate on QNs and LQNs since this dissertation extends and uses the PCM, which uses model-2-code transformations to derive simulation model based on QNs (Becker/Koziolek/Reussner, 2009).

Figure 2.15 includes an excerpt of a QN model. Hardware resources (i.e., CPU and Hard Disk Drive (HDD)) are characterized as a queue with a waiting line and a resource that serves transactions (Menascė/Almeida/Dowdy, 2004, pp. 18-19). Arrivals join the waiting line if the device does not idle, wait to use the device based on a queuing discipline, and depart to the next queue after having been served (Menascė/Almeida/Dowdy, 2004, pp. 184-198).

Among many queuing disciplines, there are four common ones (Menascė/Almeida/Dowdy, 2004, p. 27). *First Come First Served (FCFS)* serves arrivals in the order of arrival at a queue. *Priority queuing* serves the arrival that has the highest priority. *Round Robin (RR)* serves each arrival for a short period of time in a circular fashion. Finally, *Processor Sharing (PS)* represents the theoretical limit of RR as short periods of time approaches zero.

Input parameters of a QN are service demands, which specify the total average service time provided by a resource, and the workload intensity, which describes the arrivals (Menascė/Almeida/Dowdy, 2004, pp. 27-28). Arrivals are grouped into one or more classes. Therefore, QN is open if all classes are open, closed if all classes are closed, and mixed if open as well as closed classes exist (Menascė/Almeida/Dowdy, 2004, pp. 27-28). In open classes, the workload intensity is specified by an arrival rate (i.e., λ) describing the number of requests per unit time. Consequentially, there are unbounded number of arrivals in the system and the throughput is known as it equals the arrival rate (Menascė/Almeida/Dowdy, 2004, p. 20). In closed classes, the workload intensity is modeled by the number of concurrent requests. After a requests has been served, usually subsequent requests follow, for instance, after a certain think time has elapsed. In contrast to open classes, arrivals are bounded as the number of requests is known in the system and the throughput does not depend on the input arrival rate, but is an output parameter and determined at solving a QN (Menascė/Almeida/Dowdy, 2004, p. 20).

Furthermore, different types of single queue systems can be represented by a notation with three attributes, for instance, *G/G/1*. The first attribute indicates the distribution of interarrival times, for instance, a generic distribution G and an exponential distribution with a Markov property M (Menascė/Almeida/Dowdy, 2004, pp. 184-198). Similarly, the second attribute indicates the service time distribution. The third attribute indicates the number of resources.

LQN are extended QN for software systems with nested simultaneous resource possession (Franks et al., 2009). In a LQN, a resource can stop and wait for a nested request to another resource during its service. In contrast, only one resource can be used at a time in a QN. As a result, it is more suitable for modern distributed systems.

2.3.2 Architecture-Level Performance Models

Examples for architecture-level performance models are PCM and the Descartes Modeling Language (DML) (Becker/Koziolek/Reussner, 2009; Huber et al., 2017). Both are specialized to model and evaluate the performance of reusable software components.

PCM is constructed based on the roles in the component-based software engineering development process. Figure 2.16 illustrates the process model (Reussner et al., 2011; Becker/Koziolek/Reussner, 2009). *Component developers* model a repository of *component specifications* via provided and required interfaces. The behavior of the provided services of a component is model via a Resource Demanding Service Effect Specification (RDSEFF). It is similar to Unified Modeling Language (UML) activity diagram and allows to specify parametric descriptions of how the services uses hardware resources and how other services of required components are invoked. *System architects* specify an *assembly model* of an application based on the repository of components. *System deployer* model the hardware resources and deploy components from the assembly model on these hardware resources in an *allocation model*. Finally, *domain experts* specify use cases and the workload for the interfaces of the assembled application in a *usage model*.

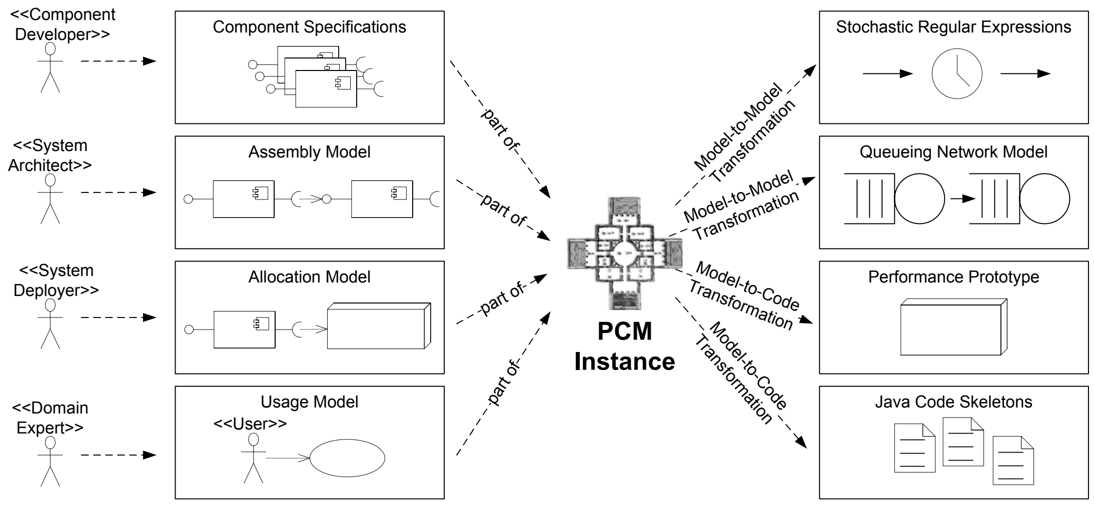


Figure 2.16: *Palladio process model (Reussner* et al.*, 2011)*

PCM provides tool support to create graphical models, predict the performance, and analyze and visualize results. In order to predict the performance, this dissertation uses the simulation framework SimuCom (Becker/Koziolek/Reussner, 2009). Therefore, model instances are transformed to simulation code that is used by SimuCom. The simulation supports $G/G/n$ queues and is based on simulation of resources and queuing networks (Becker/Koziolek/Reussner, 2009). In particular, a thread is started for each arrival according to the workload and the according use case is simulated as specified in the usage model. For each component and its RDSEFF code is generated to place resource demands description on corresponding resources as specified in the assembly model. For each resource, a request queue and scheduler is generated based on a queuing discipline. Threads that request resources to process demands will be delayed until the demand is processed. Throughout the simulation, probes include in the simulation code monitor response times and queue lengths to calculate the simulation results.

Chapter 3

Research Methodology

This chapter describes the research design that this dissertation pursues and the research methods applied for this purpose. We outline the embedded publications of this thesis as well as related publications that are (co-)authored and do not substantially contribute to the RQs.

3.1 Research Design

This thesis carries out research based on the design science research methodology (DSRM) by Peffers et al. (2007). Design science deals with inventing and developing artifacts to achieve certain goals (March/Smith, 1995; Simon, 1996; Hevner et al., 2004). DSRM is a commonly accepted framework and incorporates principles and practices from different researchers in key prior literature (Peffers et al., 2007). Its aims are achieving consistency with prior literature, providing a nominal process model and providing a mental model for presenting and evaluating design science research in information systems.

The DSRM process includes six activities in a nominal sequence although researchers are not expected to proceed in sequential order, which are described in the following (Peffers et al., 2007):

1. **Problem identification and motivation.** The research problem shall be defined and the value of a solution shall be justified to motivate the audience to pursue the solution and to follow the reasoning of the researcher.
2. **Define the objectives for a solution.** The goals shall be inferred rationally from the defined problem and can be quantitative or qualitative.
3. **Design and development.** The intended functionality of the artifact(s) shall be defined and the artifacts shall be created (e.g., constructs and models).

4. **Demonstration.** The artifact(s) shall be applied to solve the defined problem (e.g., by experimentation and simulation).

5. **Evaluation.** The artifact(s) shall be evaluated how it provides a solution to the defined problem, for instance, by quantitative measures (e.g., simulations), quantifiable measures of system performance (e.g., response time), and any empirical evidence.

6. **Communication.** The defined problem, the artifact(s), the effectiveness, and contribution shall be communicated to researchers.

3.2 Research Methods

Following the research design activities we applied different research methods in order to carry out these activities and serve its different purposes. This section describes these research methods.

Literature Reviews are a method that are usually part of research publications and shows related research that already exists but also areas with a research gap (Peffers et al., 2007; Webster/Watson, 2002). As Webster/Watson (2002) and Levy/Ellis (2006) propose, we reviewed literature, first, by selecting relevant literature and conducting keyword searching, second, by forward searching, and, third, by backward searching. We describe each step in detail in the following.

In order to identify relevant literature we started with related scholarly databases and leading workshops, conferences, and journals of related topics. We considered the following databases:

1. Association for Computing Machinery (ACM) Digital Library[3]
2. Institute of Electrical and Electronics Engineers (IEEE) Xplore[4]
3. Springer Link[5]
4. ScienceDirect[6]
5. Google Scholar[7]

We include Google Scholar, which indexes many additional outlets, to avoid the narrowness of searching only one or two database vendors (Levy/Ellis, 2006). Related literature were mainly found in the following workshops, conferences, and journals:

[3]https://dl.acm.org
[4]https://ieeexplore.ieee.org
[5]https://link.springer.com
[6]https://www.sciencedirect.com/
[7]https://scholar.google.com

1. European Performance Engineering Workshop (EPEW)
2. Symposium on Software Performance (SSP)
3. International Workshop on Big Data and Cloud Performance (DCPerf)
4. ACM/SPEC International Conference on Performance Engineering (ICPE)
5. IEEE International Conference on Software Architecture (ICSA)
6. IEEE International Symposium on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS)
7. International Conference on Performance Evaluation Methodologies and Tools (ValueTools)
8. International Conference on the Quality of Software Architecture (QoSA)
9. USENIX Symposium on Networked Systems Design and Implementation (NSDI)
10. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)
11. Business & Information Systems Engineering (BISE)
12. IEEE Transactions on Software Engineering (TSE)
13. Journal of Systems and Software (JSS)
14. Journal on Software and Systems Modeling (SoSyM)
15. Performance Evaluation Journal

We used and combined the keywords *performance evaluation*, *performance prediction*, *performance models*, *model-driven*, *simulation*, *big data*, *model extraction*, and *model generator* to gain initial insight into related work. Based on the identified works, we further searched backwards by reviewing references of these articles, reviewing additional articles published by the authors, and including keywords of the articles that have not been used in our search yet. We concluded the literature review by searching forwards and reviewing articles that have cited the identified articles.

Prototyping is a research method where instantiations such as prototype systems are produced that, inter alia, represent artifacts in design science (Hevner et al., 2004). Throughout this dissertation, we iteratively applied prototyping to develop and continuously enhance our artifacts. Similarly, the prototypes were constantly evaluated for their utility of addressing research problems and achieving research objectives.

Scenarios describe a *descriptive* design evaluation method used to to demonstrate the use of artifacts and around them (Hevner et al., 2004). Motivated by performance management perspectives and common difficulties in practice, we formulate exemplary scenarios that we claim to address with our prototypes. These scenarios are evaluated then by experimental evaluations methods.

Controlled Experiments represent an *experimental* design evaluation method and have the aim to study an artifact in a controlled environment for its qualities (Hevner et al., 2004). We continuously applied them to execute and test each prototype system in a reproducible evaluation. We use an open source and widely accepted benchmark suite that defines and includes software applications, software configurations, system configurations, and data workload generation.

Simulations also constitute an *experimental* design evaluation method and execute the artifact with artificial data (Hevner et al., 2004). We use simulations as part of controlled experiments in order to evaluate our model-based performance evaluation approach, in particular, the prediction accuracy of our prototypes compared to measurement results.

3.3 Publications

Part B of this thesis includes six publications of the author. Table 3.1 shows the details for each publication, which include a publication number, the authors, the title, and the outlet.

Publications *P1*-*P3* address the first RQ: *What features must a meta-model support to model the performance of big data systems?*. In publication *P1*, we motivate the importance of performance and scalability of complex system-of-systems architectures on the basis of big data and IoT scenarios. We describe the problems involved in evaluating the performance and propose our vision for model-driven performance prediction in order allow for evaluations early during software and system development. Furthermore, we describe the existence of existing model-driven approaches that focus on user-driven business applications but do not support features as found in data-driven applications. We formulated our aim to combine and extend existing modeling approaches. In publication *P2*, we examined the PCM, an existing approach for business applications in order to model and predict the batch layer of the lambda architecture. The lambda architecture is a generic principle to implement a big data architecture. We used Apache MapReduce as exemplary technology. The evaluation was conducted on a single node cluster as we noted missing features of PCM that would have been required to model Apache MapReduce applications in a distributed setup. In publication *P3*, we published the missing features and proposed two extensions to the meta-model of the PCM to answer the first RQ. The extensions address the modeling of distributed and parallel operations on application side and modeling of clustered resources on infrastructure and hardware side.

Publications *P4* and *P5* focus on answering the second RQ: *How can the performance of batch and stream applications be modeled and simulated?*. In publication *P4*, we apply the extended meta-model of PCM and derive a model for a simple streaming application. We further extend PCM's simulation framework to allow for actually making use of the extensions. We evaluate the modeling approach by simulating the application on a number of increasing hardware nodes and compare the predicted response time with measurement results from correspondingly executed applications. In publication *P5*, we apply our meta-model not only for a stream application but also for a batch application. In addition, we

No.	Authors	Title	Outlet
P1	**Kroß**, Voss, Krcmar	Towards a Model-driven Performance Prediction Approach for Internet of Things Architectures	Open Journal of Internet Of Things (OJIOT)
P2	**Kroß**, Brunnert, Prehofer, Runkler, Krcmar	Stream Processing On Demand for Lambda Architectures	Computer Performance Engineering. Lecture Notes in Computer Science
P3	**Kroß**, Brunnert, Krcmar	Modeling Big Data Systems by Extending the Palladio Component Model	Softwaretechnik-Trends [Software Engineering Trends]
P4	**Kroß**, Krcmar	Modeling and Simulating Apache Spark Streaming Applications	Softwaretechnik-Trends [Software Engineering Trends]
P5	**Kroß**, Krcmar	Model-Based Performance Evaluation of Batch and Stream Applications for Big Data	International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)
P6	**Kroß**, Krcmar	PerTract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop	Big Data and Cognitive Computing

Table 3.1: *Publications embedded in this thesis*

estimate and model resource demands for CPU resources and predict the CPU utilization as well as evaluate its prediction accuracy while executing applications. Compared to publication *P4*, we evaluate the approach not only for increasing hardware resources but also for increasing data workload.

Lastly, publication *P6* addresses the third RQ: *How can performance models and resource demands of big data systems be automatically extracted?*. As we experienced limitations for simulating data streams with more than 500,000 events in PCM, we wanted to abstract our approach and be independent from modeling and simulation frameworks. In publication *P6*, we introduced a DSL to describe performance characteristics of big data systems and presented an approach to automatically extract such DSL instances from big data frameworks. This includes frameworks for data processing, resource management, and data management. Based on the DSL specification, we provide an automatic transformation of DSL instances into PCM performance models in order to simulate them and predict performance metrics. In order to demonstrate and evaluate our approach, we applied the automatic extraction on two machine learning applications. We adapted parameters of DSL instances for different upscaling scenarios and compared the prediction accuracy to measurements results of correspondingly executed applications.

In addition to the embedded publications, we describe publications that we (co)authored with regard to the topic of this dissertation but do not substantially contribute to answering the research questions. As before, table 3.2 lists these further publications.

No.	Authors	Title	Outlet
P7	Danciu, **Kroß**, Brunnert, Willnecker, Vögele, Kapadia, Krcmar	Landscaping Performance Research at the ICPE and its Predecessors: A Systematic Literature Review	International Conference on Performance Engineering (ICPE)
P8	**Kroß**, Brunnert, Prehofer, Runkler, Krcmar	Model-based Performance Evaluation of Large-Scale Smart Metering Architectures	International Workhop on Large-Scale Testing (LT)
P9	Brunnert, van Hoorn, Willnecker, Danciu, Hasselbring, Heger, Herbst, Jamshidi, Jung, von Kistowski, Koziolek, **Kroß**, Spinner, Vögele, Walter, Wert	Performance-oriented DevOps: A Research Agenda	Technical Report, SPEC Research Group – DevOps Performance Working Group
P10	**Kroß**, Willnecker, Zwickl, Krcmar	PET: Continuous Performance Evaluation Tool	International Workshop on Quality-Aware DevOps (QUDOS)
P11	**Kroß**, Bezemer, Jian	Proceedings of the Sixth International Workshop on Load Testing and Benchmarking of Software Systems	International Workshop on Load Testing and Benchmarking of Software Systems (LTB)
P12	**Kroß**, Bezemer	Proceedings of the Seventh International Workshop on Load Testing and Benchmarking of Software Systems	International Workshop on Load Testing and Benchmarking of Software Systems (LTB)

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

Publication *P7* presents a historical overview of topics, methods, and trends within the software performance community. We systematically review published papers of the proceedings of the International Conference on Performance Engineering (ICPE) and its predecessors ACM Workshop on Software and Performance (WOSP) and the SPEC International Performance Evaluation Workshop (SIPEW).

In publication *P8*, we model and simulate the performance of an exemplary smart grid systems to demonstrate a model-based evaluation approach that supports design decisions at the beginning of developing a new system. In particular, we compare the scalability of the response time of two different architectures for analyzing data from smart meters.

Publication *P9* presents existing techniques and methods in the areas of SPE and APM. We further describe open research challenges in order to integrate activities of both areas, which are often considered separately.

In publications *P10*, we developed a prototype system that allows to store measurement data independent of the data collection software and includes a graphical user interface to visualize that. It addresses problems of researchers in the area of model-based performance evaluations as it automates to statistically compare simulation results with measurement results, which is usually carried out manually and error-prone.

Finally, publications *P11* and *P12* represent the proceedings of the LTB workshops in 2017[8] and 2018[9] that were co-organized by the author.

[8]http://ltb2017.eecs.yorku.ca
[9]http://ltb2018.eecs.yorku.ca

Part B*

*In this part, the original publications are slightly modified, including the unification of format and reference styles, the correction of spelling errors, and minor orthographic and grammatical revisions. Furthermore, the references included in each research study have been integrated and are presented at the end of this dissertation. The published version of the included publications can be found in the appendix.

Chapter 4

Towards a Model-driven Performance Prediction Approach for Internet of Things Architectures

Authors	Kroß, Johannes[1] (kross@fortiss.org) Voss, Sebastian[1] (voss@fortiss.org) Krcmar, Helmut[2] (krcmar@in.tum.de) [1]fortiss GmbH, Guerickestraße 25, 80805 München, Germany [2]Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Open Journal of Internet Of Things
Status	Accepted
Keywords	Performance, Model, Simulation, Prediction, Evaluation, Internet of Things, IoT, Architectures
Individual Contribution	Problem and scope definition, construction of the conceptual approach, paper writing, paper editing

Table 4.1: *Bibliographic details for P1*

Towards a Model-driven Performance Prediction Approach for Internet of Things Architectures

Abstract Indisputable, security and interoperability play major concerns in Internet of Things (IoT) architectures and applications. In this paper, however, we emphasize the role and importance of performance and scalability as additional, crucial aspects in planning and building sustainable IoT solutions. IoT architectures are complicated system-of-systems that include different developer roles, development processes, organizational units, and a multilateral governance. Its performance is often neglected during development but becomes a major concern at the end of development and results in supplemental efforts, costs, and refactoring. It should not be relied on linearly scaling for such systems only by using up-to-date technologies that may promote such behavior. Furthermore, different security or interoperability choices also have a considerable impact on performance and may result in unforeseen trade-offs. Therefore, we propose and pursue the vision of a model-driven approach to predict and evaluate the performance of IoT architectures early in the system lifecylce in order to guarantee efficient and scalable systems reaching from sensors to business applications.

4.1 Introduction

Since several years Internet of Things (IoT) constitutes one of the main future topics for industries (Atzori/Iera/Morabito, 2010). Information and communication technologies for small devices continuously become not only more affordable, but also more powerful regarding processing. This enables these devices to be connected to the Internet. Additionally, big data technologies emerged and enabled organizations to store huge amounts of data and to analyze incoming data streams with sophisticated algorithms in real-time (Schermann et al., 2014). This has facilitated the evolution of IoT and enables organizations to build solutions for a highly diverse range of use case scenarios in different domains. Therefore, IoT may be considered as an umbrella term for different disciplines that already have longer histories (e.g., industry automation) and, additionally, promotes the integration of these different disciplines, for instance, the automatic combination of sensor data with enterprise resource planning data.

Although being promoted very much, only a few IoT use cases are implemented in industry yet. Contradictorily, there are already hundreds of IoT platforms and technologies that are waiting to be exploited. In addition, there are several initiatives to define standards (Atzori/Iera/Morabito, 2010), however, their establishment progresses slowly and an oversupply of vendor-specific implementations hamper the development of integrated solutions. For instance, an IoT developer survey with 528 participants conducted by the Eclipse IoT Working Group, IEEE IoT, and AGILE-IoT suggests that security, interoperability, and connectivity represent the three major concerns among all participants for developing IoT solutions (Skerrett, 2016). However, for developers and organizations that have already deployed IoT solutions, performance becomes the third concern over connectivity. This reflects our comprehension that performance is not considered sufficiently when building architectures and finalized developments become very costly to counteract on late in the software life cycle.

We emphasize the role and importance of performance in terms of response time, throughput, and resource utilization. It is a vital aspect in planning and building sustainable IoT solutions as they involve multi-domain environments including constrained devices, gateways, and platforms of which the latter combines big data technologies and business applications. All these levels can have a direct impact on the performance of an overall system. Furthermore, evaluating the impact of design choices (e.g., regarding security, interoperability, and platforms) at development time is difficult, especially, for large-scale operations. These are only some of the factors that complicate IoT performance management.

In order to address and solve these issues, we propose the vision of a model-based approach for representing components and performance-influencing factors of IoT architectures and allow for *performance-by-design*. These models shall serve as input for analytical solvers or simulation engines and allow for predicting different performance metrics (Figure 4.1) (Brunnert et al., 2014). In this way, architectures can proactively be evaluated regarding bottlenecks and scalability. Required resource demands can be planned and the throughput and response time behavior of subsystems can be estimated. The derived performance metrics and predictions shall also contribute to support communication and collaboration between developers (e.g., embedded developers and developers for business applications) and operations.

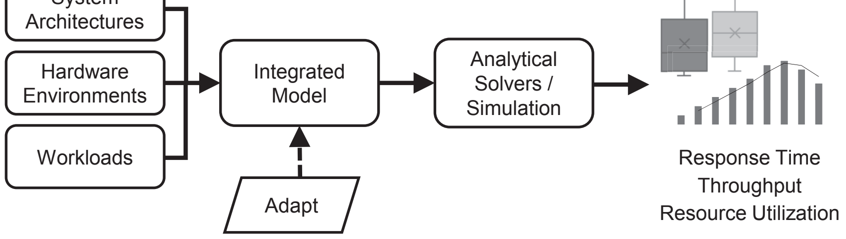


Figure 4.1: *Model-based prediction approach*

4.2 Model-driven Performance Prediction

Our vision and its realization is driven by the following three research questions, which we use to explain our proposal and intentions:

1. What resource requirements and performance difficulties occur and are relevant on different levels of IoT architectures?
2. Which existing approaches and technologies can be used for implementing the integrated modeling concept?
3. How can existing meta-models and simulation approaches of different levels be integrated and combined?

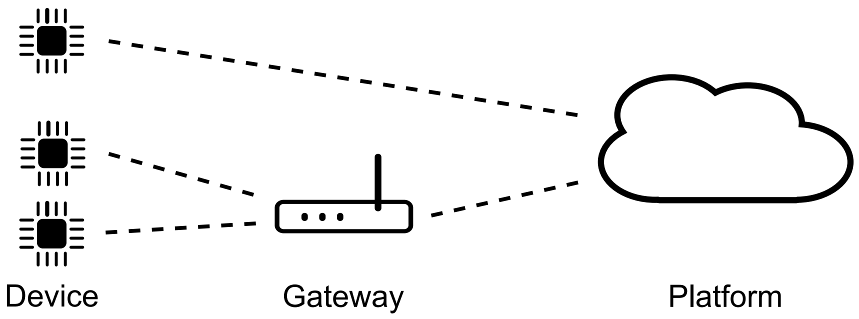


Figure 4.2: *Abstract IoT architecture*

In order to address the first research question the different levels and developer roles of architectures must be considered. Figure 4.2 shows a very basic IoT architecture that is reduced to the essential three layers. First, constrained devices and controllers represent the things in IoT. Second, gateways, routers, and smart devices enable fog computing at the edge and may integrate as well as pre-analyze data from devices (Skala et al., 2015). Third, platforms process, store, and aggregate data from different sources and enable business applications to analyze and report data to end users. The connectivity and communication between the levels is not limited to one direction. In addition, non-functional requirements such as performance, security, and interoperability are topics that influence all levels. For performance engineers, for instance, the following questions occur:

1. How shall computing resources (e.g., CPU, disk, memory, network) be sized on each level?
2. Shall gateways pre-analyze data and of how many devices per gateway?
3. What is the impact of protocol, security, and interoperability choices on the overall performance?
4. Does the architecture scale linearly with increasing number of devices and gateways?

Since the IoT stack is highly diverse, different developers and engineers are involved in the implementation. Embedded and systems developers are responsible on the device level and also partly on the gateway level. As gateways continuously expand, become smarter, and are able to run sophisticated operating systems, application developers also constitute a part on the gateway level. On the platform level, a mix of data scientists, application developers, and web developers implement the integration and visualization of data. Due to this mix of interests and engineering disciplines, we see the need to investigate the performance requirements on each level and for each role in order to understand influencing factors in a holistic view that need to be included in our model approach.

Similarly, previous and present related modeling approaches consider these disciplines in separated ways. As mentioned, IoT provides and increases the opportunity of combining existing approaches. Use case scenarios arise, for instance, that require capacity planning for devices and gateways based on formal models which are already well understood in the domain of business information systems. Since there is a tremendous number of modeling approaches, the second research questions addresses reviewing existing methods and technologies for different levels with regards to our vision. In the following we list one example technology for each level.

For the device level, for instance, AutoFOCUS3[1] represents an integrated model-based tool for the development process of embedded systems (Aravantinos et al., 2015). It includes the activities for modeling requirements, software architectures, hardware platforms, and deployments as well as for generating code. The software architecture is built up by different software components that may be connected to each other to allow for interactions and may also be decomposed into multiple hierarchical subcomponents. The hardware architecture includes resources such as processors and memory that can be linked. It also involves platform architectures for execution and runtime environments such as operating systems or Java virtual machines. The integration and combination of these models enables developers to apply different analysis and synthesis methods such as testing, model checking, deployment, and automated scheduling (Aravantinos et al., 2015).

The Eclipse Framework for Distributed Industrial Automation and Control (4diac) [23] is part of the Eclipse IoT ecosystem and represents an instance for modeling the gateway level. It provides an open source infrastructure for distributed industrial process measurement and control systems based on the IEC 61499 standard Zoitl/Strasser/Valentini (2010). In order to model software architectures, 4diac includes an application editor that allows for representing function block networks consisting of one or multiple function blocks and their interaction via events. Similarly, a separate editor is included to model the specification of hardware by modeling devices and resources. By the means of several more editors and an own runtime environment, 4diac supports the development of industrial IoT applications and facilitates portability, interoperability, configurability, and scalability as promoted by IEC 61499 (Zoitl/Strasser/Valentini, 2010).

[1]http://af3.fortiss.org
[2]https://eclipse.org/4diac/
[3]http://fortiss.org/research/projects/4diac/

For the platform level, the performance management work tools (PMWT)[4] provide several integrated approaches to automate, support and integrate performance engineering activities across the software lifecycle (Brunnert et al., 2014). This includes the automatic generation of models for enterprise applications based on performance measurements (Brunnert/Krcmar, 2017), modeling complex user behavior of applications (Vögele et al., 2016), and simulating the performance of big data applications (Kroß/Krcmar, 2016).

In order to address the third research question, we will examine similarities of model-based approaches for the different levels and domains of IoT architectures. For instance, models on architecture-level may often separate their meta-model as illustrated in Figure 4.1. One or several models are used to describe the software and system architecture, its components and its activity flow. Another model is used to describe the hardware and resource environment such as computing nodes with processors, disks, and memory connected via a network. An additional usage model is used to describe the use case scenarios of the software architecture and the workload.

Although implicitly considering performance aspects, present solutions focusing on the device and gateway level usually concentrate on guaranteeing functionality and safety (Aravantinos et al., 2015). In contrast, there are a lot of performance models to predict and analyze behavior on the platform level. Existing models on architecture-level that provide the benefits we seek with our vision, however, only focus on classical business applications and involve different requirements. For instance, the workload of business applications is mostly user-driven such as the number of parallel user accesses, whereas IoT applications are mostly data-driven such as the volume, velocity, and variety of incoming data. In addition, massive distributed and parallel computing and resource clusters are properties that are usually not found in business applications. Therefore, we aim at combining existing model approaches and additionally implementing missing functionalities so we are able to model the performance requirements we identified in the first research questions.

In order to be able to predict the performance behavior of the architectures, we will also implement simulators and solvers for deriving different metrics such as response time, throughput, and resource utilization. To evaluate our approach (Figure 4.1), we plan to model applications from IoT benchmark suites and adapt these models for various scenarios such as increasing number of things and increasing resource capacities. After simulating the models, we will compare simulation results with measurement results of applications from the benchmark that we adapted in the same way.

4.3 Related Work

As already mentioned, IoT involves and combines a variety of application domains and development stacks. Consequently, there are a lot of related, but also highly diverse approaches of which we try to refer to examples to the best of our knowledge in this Section. There are also several solutions available to model constrained devices, simulate networks within IoT architectures on a very detailed level, and evaluate them for throughput and

[4]https://pmw.fortiss.org/tools/pmwt/

latency issues. For instance, Wang et al. (2016) apply the network simulator OPNET for IoT cloud solutions; Brambilla et al. (2014) propose a simulation methodology to test large-scale IoT systems with interconnected devices in urban environments and include several network protocols and different mobility, network, and energy consumption models.

Furthermore, many related approaches specifically analyze and compare the performance of different protocols or technologies on different layers. For scenarios in which devices and gateways do not have a wired connection, for instance, Costantino et al. (2012) investigate LTE as a suitable interconnection in terms of its efficiency, bandwidth, and coverage. In contrast, Daud/Suhaili (2017) provide a performance evaluation of protocols for the application layer in IoT architectures. Therefore, they compare the hypertext transfer protocol (HTTP) and the constrained application protocol (CoAP) for message formatting, communication, and request handling on different test beds.

There are several developments of performance benchmarks for IoT, however, mostly on the platform level. Arlitt et al. present an analytics benchmark called IoTAbench (Arlitt et al., 2015). It allows for generating, loading, repairing and analyzing synthetic data and was evaluated by the example of a smart metering use case and using a HP Vertica database. (Shukla/Chaturvedi/Simmhan, 2017) propose another benchmark for distributed stream processing platforms (i.e., Apache Storm) called RIoTBench. They provide different data workloads and generators as well as a set of 27 common IoT tasks for different domains. Furthermore, Medvedev et al. (2017) provide an evaluation of different IoT platforms with regards to performance characteristics.

4.4 Conclusion and Future Work

This paper proposes and pursues the vision of an model-based approach for predicting and evaluating the performance of IoT architectures and systems. It shall support developers and engineers at examining design choices early in the system lifecycle, finding potential bottlenecks, planning and sizing required resources on different levels, and predicting response times from sensors to visual results. We will start our future research and work with combining and integrating modeling approaches for embedded systems with approaches for big data systems as well as for business information systems. Therefore, we are currently developing a first prototype for an integrated modeling environment.

Chapter 5

Stream Processing On Demand for Lambda Architectures

Authors	Kroß, Johannes[1] (kross@fortiss.org) Brunnert, Andreas[1] (brunnert@fortiss.org) Prehofer, Christian[1] (prehofer@fortiss.org) Runkler, Thomas A.[2] (thomas.runkler@siemens.com) Krcmar, Helmut[3] (krcmar@in.tum.de) [1]fortiss GmbH, Guerickestraße 25, 80805 München, Germany [2]Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, 81739 Munich, Germany [3]Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Computer Performance Engineering. EPEW 2015. Lecture Notes in Computer Science
Status	Accepted
Keywords	Lambda Architecture, Big Data, Performance, Model, Evaluation
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

Table 5.1: *Bibliographic details for P2*

Stream Processing On Demand for Lambda Architectures

Abstract Growing amounts of data and the demand to process them within time constraints have led to the development of big data systems. A generic principle to design such systems that allows for low latency results is called the lambda architecture. It defines that data is analyzed twice by combining batch and stream processing techniques in order to provide a real time view. This redundant processing of data makes this architecture very expensive. In cases where process results are not continuously required to be low latency or time constraints lie within several minutes, a clear decision whether both processing layers are inevitable is not possible yet. Therefore, we propose stream processing on demand within the lambda architecture in order to efficiently use resources and reduce hardware investments. We use performance models as an analytical decision-making solution to predict response times of batch processes and to decide when to additionally deploy stream processes. By the example of a smart energy use case we implement and evaluate the accuracy of our proposed solution.

5.1 Introduction

With the increasing ubiquity of information and communication technology (ICT) and the emergence of the Internet of things (IoT) the available data amount is growing exponentially. Simultaneously, technologies have been developed to store, manage and analyze these diverse and high volumes of data, also known as *big data* (Schermann et al., 2014). These circumstances allow for applying analytics in order to gain knowledge and support decision-making. For more and more usage scenarios, these analytical capabilities must also meet specific time requirements such as real-time (Chen/Zhang, 2014). One common approach to design big data systems that can cover many use cases is the lambda architecture (Marz/Warren, 2015). It mainly consists of a batch layer and a speed layer. The former iteratively processes a set of historical data in batches while the latter processes the arriving data stream in parallel to incrementally analyze latest data. By joining the output of both layers query results always reflect current data.

Nowadays, various complementary technologies with different characteristics exist to build a big data system and there is hardly one technology solution that fits most use cases of an organization. Although the lambda architecture simply is a generic design framework which offers a solution for many use cases, nonetheless, a variety of technologies can be applied for the batch or speed layer. Examples for the batch layer are Hadoop MapReduce (Apache Hadoop, 2015), Apache Pig (Apache Pig, 2014), and Apache Spark (Apache Spark, 2015) and for the speed layer Apache Storm (Apache Storm, 2015), Apache Spark Streaming (Apache Spark, 2015), Apache Samza (Apache Samza, 2015), or Amazon Kinesis (Amazon Web Services, 2015). This multitude leads to the development of complex system of systems, which often results in performance issues and high resource requirements (Brunnert et al., 2014). Furthermore, the lambda architecture intends to process all data twice in both layers. Batch processes also analyze data from the ground up in each iteration to ensure fault tolerance in case of hardware failures or human mistakes (Marz/Warren, 2015). These fundamental ideas require costly resources. For use cases where time constraints are not continuously needed or lie between several minutes, it can be often an important question whether a speed layer is really required or not. However, this question can usually not be answered during system development nor in test systems under realistic workload. As stream processing heavily utilizes main memory, the speed layer can also become an expensive investment (Liu/Iftikhar/Xie, 2014).

Therefore, we propose a speed layer or stream processing, respectively, on demand. The idea is to exclusively use batch processes as often as possible and switch on stream processing only when batch processes are likely to exceed response time constraints. In this way, computing power is utilized more efficiently and resources can be saved as well as be available for other processes. In case of virtualized environments, investments can be directly decreased by reducing cloud service resources. In order to switch on stream processing at the right time, it is inevitable to predict the response time of succeeding batch iterations. For this purpose, we use performance models. They allow to describe performance influencing factors of software systems and to predict performance metrics such as response time, throughput and utilization by means of analytical solvers or simulation engines (Brosig et al., 2015). Therefore, we integrate estimated resource demands into the model based on measurements from batch processes to simulate an accurate system behavior. This enables us to efficiently schedule stream processes.

In this paper, we first give a detailed description of our proposed approach in Section 5.2 and how we use performance models to support decision-making. In Section 5.3, we validate our approach in an experiment. We describe the selected use case, the setup and sample algorithm for the batch layer, and the prototype performance model to predict batch processes. Afterwards, we discuss the experimental results we derived for different workload scenarios. In Section 5.4, we reflect related work in the area of the lambda architecture and, finally, conclude our paper with providing an outlook for future work in Section 5.5.

5.2 Stream Processing On Demand

In order to make decisions about when to switch on stream processing, we use performance models as an analytical solution. As illustrated in Figure 5.1, the iterative process is divided into two main steps in which the following Sections 5.2.1 and 5.2.2 are structured.

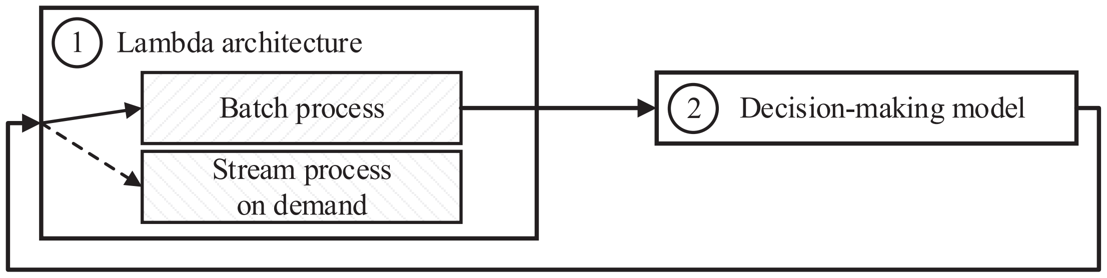


Figure 5.1: *Stream processing on demand process (© 2015 Springer)*

First, one batch iteration and, potentially, a concurrent stream process are started within the lambda architecture. Second, after the batch process has ended, a decision-making model is used to decide whether stream processing is required in the next batch process iteration or not. Basis of decision-making is a performance model which is used to predict the response time of a batch process. Afterwards, the procedure is repeated.

5.2.1 Data Processing in the Lambda Architecture

As already mentioned, our focus is on data processing, namely batch and stream processing, within lambda architecture and not storing data sets or results. Figure 5.2 illustrates the data flow and structure of batch and speed layer that differ from each other. Starting point is a shared data source which either streams the same data into each processing layer or gets accessed by each layer to retrieve data. Within the batch layer, all data are stored in a data set. A special characteristic of the data set is that it is append-only and data are not updated or removed (Marz/Warren, 2015). Batch processes use the data set to operate on. In doing so, each batch process usually analyzes a huge set of historical data which leads to response times of minutes or hours for one batch job. The results are written to separate views, which is also considered as serving layer by Marz/Warren (2015) for batch results. Batch processes constantly run iteratively and start from the beginning once a batch job has finished. If a batch process starts, only data that have been created before are included. Consequently, data that arrive during the current batch process are only included in the next new batch process. Since all data are analyzed in each cycle, each new result view can replace its predecessor. As the batch layer does not rely on incremental processing, it has the advantage of being a robust system where everything can be recomputed and reconstructed in case of hardware or software failures or human mistakes (Marz/Warren, 2015).

In contrast to the batch layer, the speed layer does not keep a record of historical data and solely uses main memory. As of today, stream processes run permanently and analyze each incoming message. They incrementally calculate and immediately update their result views. Thus, both layers include separated views and, in practice, usually different technologies are used as underlying databases because of their distinct requirements regarding read and write operations. In order to receive a holistic result, the view of both layers have to be merged in a query.

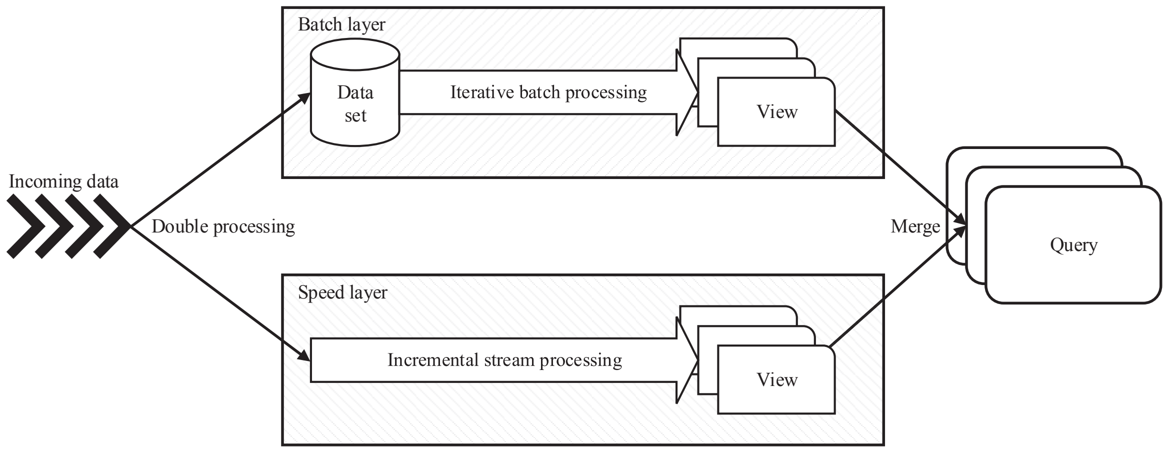


Figure 5.2: *Composition and data flow of batch and speed layer of the lambda architecture (adapted from Marz/Warren, 2015) (© 2015 Springer)*

Although both layers process the same data, the results of queries that merge views only reflect data that are processed once at the time of the query. The purpose of the speed layer is to analyze the data prior to the batch layer and enable low latency by incremental updated result views. As a result, a past view of the speed layer can be discarded as soon as a subsequent batch job has finished.

A typical implementation of the lambda architecture as illustrated in Figure 5.2 would be to use Apache Kafka (Apache Kafka, 2015) - a publish-subscribe messaging system - as shared source for incoming data. For the batch layer, HDFS can be used as data set and Hadoop MapReduce for batch processing. For storing batch results, which Marz/Warren (2015) also describe as serving layer, ElephantDB[1] represents a specialized database for this purpose. For the speed layer Apache Storm (Apache Storm, 2015) is an example of an appropriate technology and Apache Cassandra (Apache Cassandra, 2015) of a database.

5.2.2 Decision-making Model

To decide when to switch on stream processing, we predict the response time of succeeding batch processes and build a decision-making model. To comprehend why it is necessary to predict the succeeding batch processes, the chronological sequence of batch and stream processes as intended by the lambda architecture is illustrated in Figure 5.3. As already mentioned, results of batch processes are not available until they finish, while results of stream processes are incremental and can be queried at any time. Supposing one *batch process i* has ended and a decision must be made at time y on whether additional stream processes are needed afterwards or not, the earliest point in time where results of stream processes can be reasonably used is at time z. *Stream process j* considers only data newer than time y. Therefore, a batch process is required that has analyzed data before time y. However, the corresponding *batch process j* will only start after time y and end at a given

[1]https://github.com/nathanmarz/elephantdb

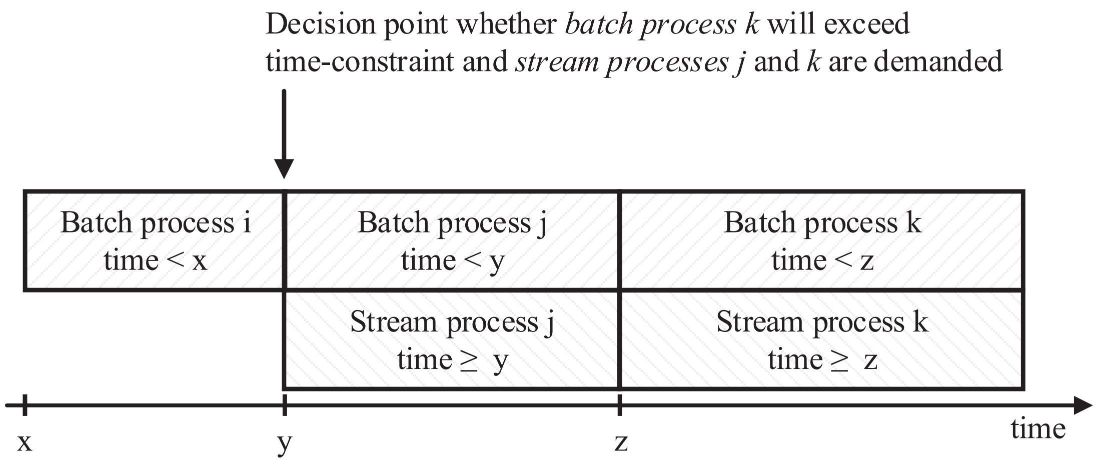

Decision point whether *batch process k* will exceed time-constraint and *stream processes j* and *k* are demanded
Batch process i time < x
Batch process j time < y
Batch process k time < z
Stream process j time ≥ y
Stream process k time ≥ z
x
y
z
time

Figure 5.3: *Chronological sequence of batch and stream processes (© 2015 Springer)*

time z. Thus, a decision must already be made at time y, if *batch process k* violates time-constraints so stream processes are switched on at time y. Consequently, query results after time z will have consistently incorporated all data.

The above mentioned response time prediction is part of our decision-making model. Its procedure is depicted in Figure 5.4. Starting point is a finished batch process iteration. The response time of the second next batch iteration is predicted by using a performance model, which takes two inputs - the time constraint for the duration of a batch process and the load intensity. The latter means information about the incoming data of the batch layer. For instance, this can be in the form of a variable distribution as modeled by the LIMBO tool (von Kistowski/Herbst/Kounev, 2014). The prediction can be accomplished by means of simulation or analytical solving. If the predicted response time does not lie within the specified time limitation, the model tries to start batch processing in parallel with stream processing, otherwise the model considers batch processing only as sufficient.

5.3 Experimental Validation

For the evaluation of our proposed approach, we conduct a controlled experiment which is described in the following Subsections. First, we discuss the selected use case. Second, we list the used setup and technologies of our exemplary batch layer as well as the sample algorithm for data processing. Afterwards, the performance model prototype to support decision-making is presented. Finally, we evaluate the accuracy of the inferred decision-making on the basis of three selected scenarios and discuss results from our observed measures.

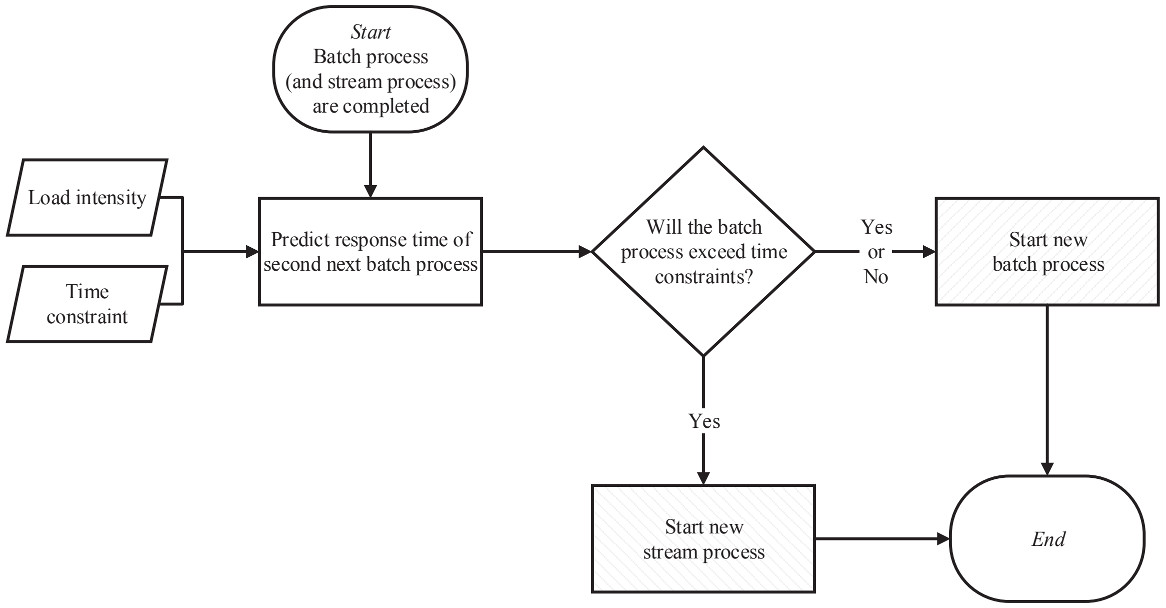


Figure 5.4: *Decision-making model (© 2015 Springer)*

5.3.1 Use Case and Design Options

To represent incoming data and their distribution, we pick the example of a common smart energy use case as illustrated in Figure 5.5.

Here, several hundred wind turbines are positioned in several wind farms in different geographic locations with long distances onshore or offshore. In order to operate efficiently, they measure several thousand parameters per turbine such as pressure, temperature or vibrations of rotor blades. As they are subject to various influences, wind turbines are not always in operation and do not measure data, for instance, if they are defect or are maintained. While onshore wind turbines and wind farms, respectively, tend to have a time-based availability between 95-99%, the values for offshore wind farms with distance less than 12km range from 67.4% to 90.4% (Faulstich/Hahn/Tavner, 2011). However, wind turbines include also downtimes, if wind is too strong or too weak which is described by the metric energy-based availability. Faulstich/Lyding/Tavner (2011) compared time-based and energy-based availability of wind turbines. In an extreme case where the downtime due to defects and the downtime due to wind speed does not overlap, the energy-based availability lies within 90.4-95,2%.

Dependent on a wind turbine's availability, we assume it either produces a set of measurement data with constant volume or does not produce any output data. As a result, wind turbines generate not only immense amount of heterogeneous data, but also variable load which makes it difficult to predict the production rate of data. As soon as data are generated, they flow into a central data center where they are processed. Dependent on the use case, data are handled in different ways. They can be gathered and stored in a central repository where batch processing can be used to extract, transform, and load (ETL) data and to apply complex analytics. This procedure usually lies in the range of minutes or hours and is not suitable for real-time requirements. For this purpose, stream

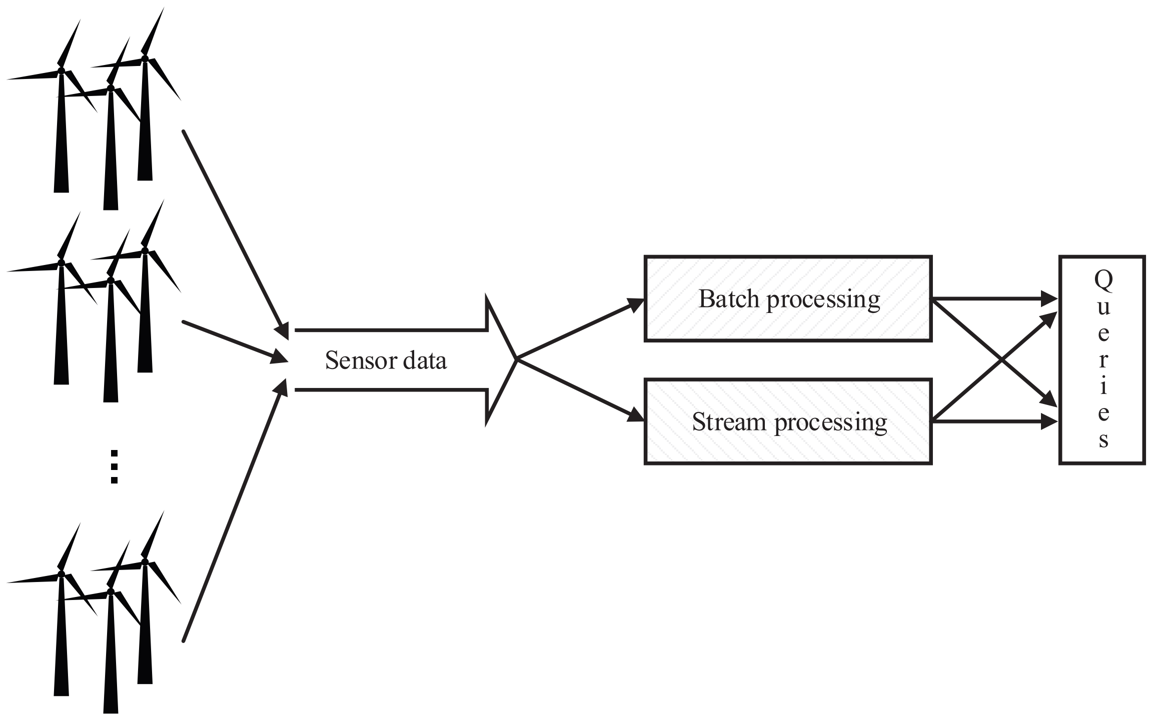


Figure 5.5: *Data processing of wind power facilities (© 2015 Springer)*

processing can be used to directly process data as they stream in. Here, analytical algorithms may be designed in a simpler and less complex way than at batch processing as well as implemented in slightly different way as they produce incremental results.

In scenarios where low latency results are required and normally stream processing is chosen, but also analysis of historical data by batch processing need to be incorporated for conclusive results, the lambda architecture is an appropriate solution that allows for serving such use cases. Therefore, on both processing layers, stream and batch, the same kind of algorithm is implemented and results are joined.

Sensor data can be used for a variety of analytical scenarios such as for condition monitoring, diagnostics, predictive analytics or maintenance, and load forecasting. For our experiment, we concentrate on the latter example. Since the introduction of energy exchange such as the continuous intraday spot market of the European power exchange (EPEX), power can be bargained in 15-minute intervals up to 45 minutes before delivery which enables providers as well as consumers to efficiently act on short notice. In this case, the time-constraint is within 15 minutes. Typical forecast methods for short-term load forecasting include different exponential smoothing methods such as an autoregressive integrated moving average (ARIMA) model (Taylor, 2008) or recurrent neural networks (Schäfer/Zimmermann, 2006). Furthermore, these algorithms are often applied on a sliding window of historical data.

Therefore, we will use this smart energy scenario as an example for our proposed approach and generate sensor data that are processed by one central system in similarly way as we have modeled it in a previous work (Kroß et al., 2015a). The generator produces comma-separated values (CSV) files that represent measurements from wind turbines of one wind farm. Listing 5.1 shows the file structure and syntax.

Listing 5.1: *Example of generated monitoring data from wind turbines*

```
id,    timestamp,                  power,    param1, ...   paramN
12,    2015-04-01 08:23:04.125,    12.67,    value1, ...   value1
15,    2015-04-01 08:23:03.973,    13.49,    value2, ...   value2
13,    2015-04-01 08:23:04.096,    12.59,    value3, ...   value3
...
```

Each line represents a measurement of one wind turbine consisting of a *id*, *timestamp*, a *power* value and several hundred more parameters which we generated randomly and do not include in our succeeding analytic algorithms.

5.3.2 Implementation of the Batch Layer

To examine the accuracy of response time prediction for batch processes, we setup the batch layer using HDFS to store data sets and Hadoop MapReduce for batch processing. For simplicity, we installed a single node cluster in pseudo-distributed mode so Apache Hadoop runs only on one machine, but their daemons have their own Java processes. In order to do load forecasting and apply the data generator as mentioned in Section 5.3.1, we implemented a simple moving average algorithm in a Hadoop MapReduce job. It is based on an example algorithm[2].

The MapReduce programming model intends to implement one map and one reduce function. The former takes a key/value pair as input and produces a set of key/value pairs, whereas the latter takes a key and set of associated values and combines the values to another smaller set (Dean/Ghemawat, 2008). In our case the map function is implemented as

Listing 5.2: *Map function pseudo code*

```
map(Object key1, String value1):
    // key1:   file name
    // value1: measurements of wind turbines of one farm
    for each line l in value:
        kv = parse(l)
        emit({kv.id, kv.timestamp}, {kv.timestamp, kv.power})
```

The function is called for each file within a given folder. It receives one CSV file and its value, which are multiple rows of measurement data of wind turbines. The algorithm reads every line and parses it in order to filter the *id* of a wind turbine, the *timestamp* of the measurement and the *power* value that describes the generated power to that time. Afterwards it releases a composite key containing the *id* and *timestamp*, and the values *timestamp* and *power*. By using a composite key Hadoop sorts the ids of wind turbines

[2] https://github.com/jpatanooga/Caduceus/

and, in a secondary sort, the timestamp for each id. Subsequently, the reduce method results in a simpler design as displayed in Listing 5.3.

Listing 5.3: *Reduce function pseudo code*

```
reduce(Object key, Iterator<object> values):
    // key:    an object containing id and timestamp
    // values: power values ordered by timestamp
    result = simpleMovingAverage(values)
    emit(id, result)
```

The reduce function is called for each different wind turbine and calculates the actual simple moving average. It receives the key object and a list of values as input which contains timestamps and power values sorted by the former. The function itself calculates the *result* and emits it with the corresponding wind turbine *id*.

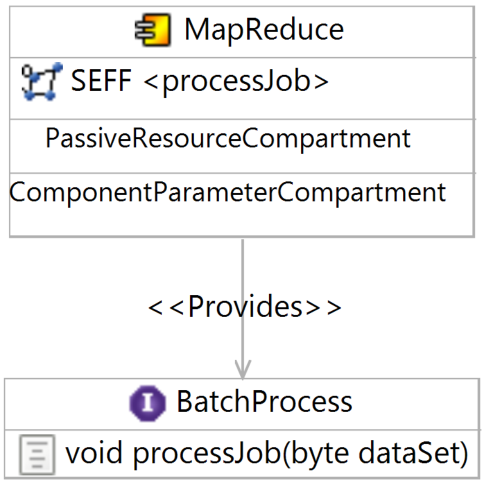
5.3.3 Performance Model Prototype

We use the Palladio component model (PCM) (Becker/Koziolek/Reussner, 2009) for our performance model. PCM is an annotated software architecture model that allows for describing performance relevant factors of software architecture, execution environment and usage profile (Brosig et al., 2015). Such performance models enable software architects and performance engineers to predict performance metrics such as response time, utilization or throughput by means of simulation or analytical solving.

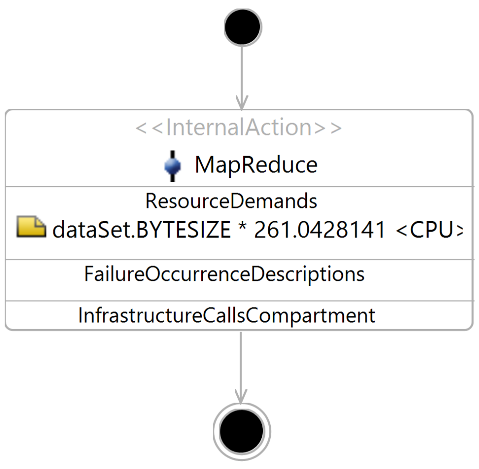
PCM is divided into several sub-models. In the repository model, we specify a batch process as a software component with its service effect specification (SEFF) to describe the resource demands of the provided service. In the resource environment model, we describe the hardware resources and processing rates on which a batch process will be executed. The concrete assignment of modeled batch processes to resources is determined in the allocation model. Finally, we specify the load intensity from wind turbine measurements in the usage model.

Figure 5.6 shows the substantials of modeling the batch process in our performance model. As shown in Figure 5.6a, we specify one interface *BatchProcess* with the method *processJob* to analyze an input data set. The implementation of the interface and its method is modeled by the component *MapReduce* with the corresponding SEFF. As illustrated in Figure 5.6b the SEFF itself solely consists of a CPU resource demand in dependence on an incoming data set size. The data set size is specified in the usage model, in our case, in gigabyte.

In order to define the CPU resource demand and simulate a realistic system behavior we integrated measurements into our performance model. Therefore, we measured response times of the MapReduce job described in Section 5.3.2 while running it. Afterwards, we used an approximation with response times, which is also implemented by the LibReDe library (Spinner et al., 2014), to estimate the required CPU time each process takes per transaction. One transaction means exactly one batch process that analyzes a set of


MapReduce
SEFF <processJob>
PassiveResourceCompartment
ComponentParameterCompartment
<<Provides>>
BatchProcess
void processJob(byte dataSet)

(a) *Repository model*


<<InternalAction>>
MapReduce
ResourceDemands
dataSet.BYTESIZE * 261.0428141 <CPU>
FailureOccurrenceDescriptions
InfrastructureCallsCompartment

(b) *Service effect specification (SEFF) <processJob>*

Figure 5.6: *Modeling a batch process with the Palladio component model (© 2015 Springer)*

messages. In our case, the resulting resource demand we estimated is 261 as represented in Figure 5.6b.

In order to predict results, PCM instances must be first transferred to be either simulated or solved analytically. Available model transformations are a model-to-text transformation like SimuCom (Becker/Koziolek/Reussner, 2009), queuing Petri nets (QPN) transformations as well as a transformation to layered queuing networks (LQN). Brosig et al. (2015) evaluated these model transformations with regards to their efficiency and accuracy. In our application scenario, time is critical and the model need to be solved as efficiently as possible so resulting predictions are available at an early opportunity and the next batch process can be initiated. Therefore, we recommend the use of a model transformation to LQNs. It showed to be the most efficient solution as it is an analytical solver (Brosig et al., 2015).

The performance model prototype has the limitation that is does not reflect the scheduling of processes itself within a cluster, for instance, as accomplished by Apache Hadoop YARN. Therefore, we assume sufficient available resources so batch and stream processes always run without interference.

5.3.4 Controlled Experiment

To conduct our experiments we run the mentioned data generator to produce CSV files for 10 wind farms with 100 wind turbines each, whereas one wind turbine approximately produces one measurement every second. Afterwards, we run the implemented Hadoop MapReduce job which reads only data measured within a sliding window of 24 hours. While the batch process is running, meanwhile we determine the incoming data volume. After the batch process is finished, we predict the response time of the second next batch process using our performance model. For the immediate succeeding batch process, we exactly know the data volume it will process as we know the historical data distribution and tracked new arrived data. For the batch process to be predicted, the data volume must be estimated. Therefore, a variety of specialized tools and algorithms exist to classify and forecast workload such as the approach by Herbst et al. (2014). As we target an efficient solution and a short-term forecast is required, namely, only the next point, we only use a naïve forecast in this study. It does not involve any computational overhead and simply takes the value of the latest observation as next forecast point in contrast to other methods such as cubic smoothing splines or ARIMA 101 that are more appropriate for scenarios with strong trends or noises (Herbst et al., 2014). In our case, the next forecast point equals the arrived data volume which has not been absorbed by the last batch process yet. Afterwards, we trigger the performance models with the predicted load intensity as input, and compare the predicted response time with the eventual measured response time.

As already mentioned, the aim is to minimize the usage of the speed layer. The level of potential resource reductions and costs savings that can be achieved depends on the characteristics of the underlying workload and variations in data distributions. The effectiveness of our solution itself, however, depends on how well the data volume is predicted and, especially, how accurate batch processes are predicted. Therefore, we concentrate on the latter in this controlled experiment and perform three selected scenarios with different load intensities by assuming different availabilities of wind turbines based on Faulstich/Hahn/Tavner (2011); Faulstich/Lyding/Tavner (2011) to evaluate the accuracy of our solution.

Table 5.2: *Measured and predicted results of batch processes (© 2015 Springer)*

Scenario	WTA	Fluctuation	PRT	MRT	RE
1	85 %	± 0 %	12.78 minutes	12.17 minutes	5.01 %
	90 %	± 0 %	13.53 minutes	13.60 minutes	0.51 %
	95 %	± 0 %	14.28 minutes	15.47 minutes	7.69 %
2	85 %	+ 5 %	12.78 minutes	13.82 minutes	7.53 %
	90 %	+ 5 %	13.53 minutes	15.03 minutes	9.98 %
3	90 %	− 5 %	13.53 minutes	12.58 minutes	7.55 %
	95 %	− 5 %	14.28 minutes	13.17 minutes	8.43 %

In the first scenario, we assume the wind turbine availability (WTA) is constant during two following batch iterations. Consequently, the measurement data wind turbines produce do also not fluctuate so the predicted load intensity using a naïve forecast is very close to the actual measured load intensity. In the second scenario, we assume an increase of the WTA of 5 % for the subsequent batch process and, vice versa, we assume a decrease in a final third scenario. For each scenario, we conduct several experiments with different WTA to also validate the prediction accuracy under different load intensities. Afterwards we compare predicted response times (PRT) with eventual measured response times (MRT) of the batch process and calculate the relative error (RE) of the PRT. The results are listed in Table 5.2.

For a WTA of 85% and no fluctuation during the following batch process, we predict the response time for the batch process to be 12.78 minutes. We measured a MRT of 12.17

minutes which leads to a RE of 5.01%. For a WTA of 90%, the RE of the predicted response time is only 0.51 % and 7.69% for a WTA of 95%.

In the second scenario, for a 85% WTA and a 5% increase of available wind turbines during the following batch iteration, the PRT is 12.78 minutes and the MRT 13.82 minutes with a 7.53% RE. Here, the PRT equals the same PRT as in the experiment for first scenario with a 85% WTA since the naïve forecast, as already mentioned, uses the last observation point, namely 85%, as next prediction point. The same occurrence also applies for the following experiments. The highest RE with 9.98% appeared for a WTA of 90% with +5% fluctuation at which the PRT is 13.53 minutes and the MRT 15.03 minutes.

For a decrease of the 5% WTA in the last scenario, we measured REs in the range similar to the former scenario. With a starting point of 90% WTA, the PRT is 13.53 minutes and the MRT 12.58 minutes. For 95% WTA, the PRT equals 14.28 minutes and MRT 13.17 minutes.

In our experiments, we showed that we are able to predict the response times of a batch process or MapReduce job, respectively, with RE between 0.51% and 9.98%. With regards to our exemplary use case, power can be traded every quarter of an hour in the intraday spot market. Assuming a fluctuating workload and a maximum acceptable response time of 14 minutes remaining one minute buffer, we would be able to accurately schedule stream processing in the second scenario, namely, not to switch on in the first experiment and to switch on stream processing in the second experiment as the MRT exceeds the time-constraint with 15.03 minutes. For a decreasing fluctuation, we would proper schedule stream processing for a starting WTA of 90%. However, for the last experiment in Table 5.2, we would have left the speed layer switched on as the PRT lies over 14 minutes in contrast to the MRT which is mainly caused by the naïve forecast.

5.4 Related Work

Similar to our use case, Sequeira et al. (2014) propose a system based on the lambda architecture to analyze energy consumption. Martínez-Prieto et al. (2015) adapted the architecture for semantic data and Casado/Younas (2015) give an extensive review about technologies for the lambda architecture. Regarding optimization or efficient resource usage of the architecture, however, related research mainly focuses on the processing layers itself. For instance, Aniello/Baldoni/Querzoni (2013) and Rychlý/Škoda/Smrž (2015) specify on scheduling stream processes, while Alrokayan/Vahid Dastjerdi/Buyya (2014) concentrate on scheduling batch processes.

Regarding predicting batch processes, there is comprehensive research available, for instance, specialized for MapReduce jobs
(Barbierato/Gribaudo/Iacono, 2014; Verma/Cherkasova/Campbell, 2011; Vianna et al., 2013) as well as for big data applications in cloud infrastructures (Castiglione et al., 2014).

To overcome redundancy regarding software development and infrastructure complexity, approaches such as storm-yarn[3] or by Nabi/Wagle/Bouillet (2014) exist to integrate stream processing in the Apache Hadoop environment. Summingbird[4] is an open source library that allows to write algorithms that can be used for batch as well as stream processing.

5.5 Conclusion and Future Work

This paper introduced a novel approach to use resources more efficiently when implementing the lambda architecture. It is applicable for usage scenarios where time constraints of queries are not permanently required to be low or lie within several minutes. To reduce processing power, we propose to switch on stream processing on demand in cases where batch processes are likely to exceed time requirements. By using historical information of incoming data and naïve forecasting to classify workload, we predicted the response time of succeeding batch iterations. Therefore, we used performance models in which we integrated estimated resource demands based on measurements. The results allow us to make decisions when additional stream processes are required or, vice versa, can be saved to reduce resource usage. If hardware provision is used in a as-a-service manner, it allows for reducing costs directly.

For future work we plan to automate the process illustrated in Figure 5.1. This involves to automatically measure incoming data during each batch iteration, apply workload forecasting techniques and trigger solving the performance model. Another challenge is to also integrate the speed layer into our test environment. This will enable us to examine our approach and its efficiency for successive batch iterations for a lengthy period of time. Furthermore, we will integrate other workload forecasting techniques besides the naïve forecast to evaluate possible prediction enhancements and scheduling optimizations.

[3]https://github.com/yahoo/storm-yarn
[4]https://github.com/twitter/summingbird

Chapter 6

Modeling Big Data Systems by Extending the Palladio Component Model

Authors	Kroß, Johannes[1] (kross@fortiss.org) Brunnert, Andreas[1] (brunnert@fortiss.org) Krcmar, Helmut[2] (krcmar@in.tum.de) [1]fortiss GmbH, Guerickestraße 25, 80805 München, Germany [2]Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Softwaretechnik-Trends - Sonderteil: Proceedings of the Symposium on Software Performance (SSP)
Status	Accepted
Keywords	Palladio Component Model, Performance Model, Big Data
Individual Contribution	Problem and scope definition, construction of the conceptual approach, paper writing, paper editing

Table 6.1: *Bibliographic details for P3*

Modeling Big Data Systems by Extending the Palladio Component Model

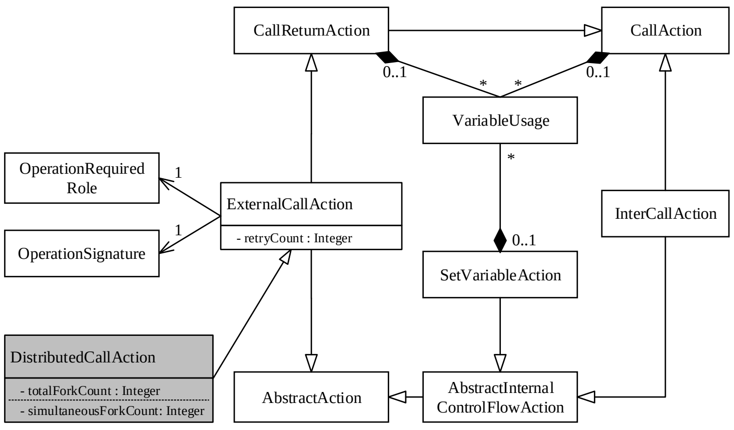
Abstract The growing availability of big data has induced new storing and processing techniques implemented in big data systems such as Apache Hadoop or Apache Spark. With increased implementations of these systems in organizations, simultaneously, the requirements regarding performance qualities such as response time, throughput, and resource utilization increase to create added value. Guaranteeing these performance requirements as well as efficiently planning needed capacities in advance is an enormous challenge. Performance models such as the Palladio component model (PCM) allow for addressing such problems. Therefore, we propose a meta-model extension for PCM to be able to model typical characteristics of big data systems. The extension consists of two parts. First, the meta-model is extended to support parallel computing by forking an operation multiple times on a computer cluster as intended by the single instruction, multiple data (SIMD) architecture. Second, modeling of computer clusters is integrated into the meta-model so operations can be properly scheduled on contained computing nodes.

6.1 Introduction

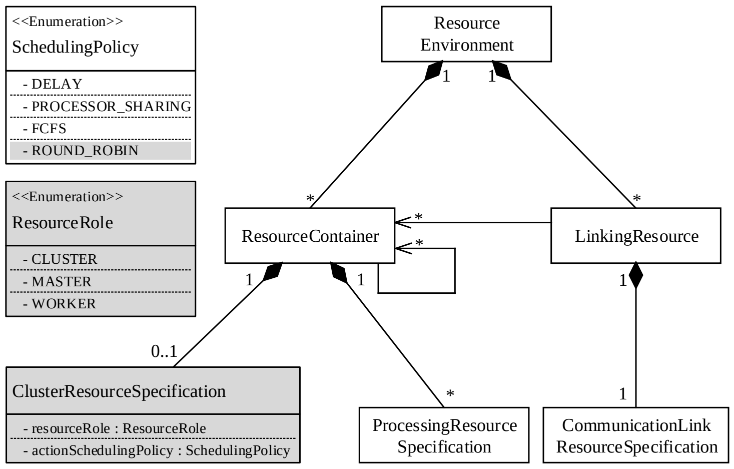
Exponentially growing volumes of data of various formats—referred to as big data—and the necessity of organizations to gain benefits have led to the development of big data systems (Casado/Younas, 2015; Schermann et al., 2014). These systems are specialized for storing and processing this data. A common example includes Apache Hadoop[1] consisting of a distributed file system called *HDFS*, a scheduler and cluster resource manager called *YARN* and the *MapReduce* model for parallel data processing (Dean/Ghemawat, 2008).

Although Apache Hadoop is originally built for commodity hardware, other systems such as Apache Spark (Streaming)[2] and Apache Storm[3] have emerged that enable low latency results on big data by also using in-memory computing (Zaharia et al., 2012a). Therefore,

[1]http://hadoop.apache.org/
[2]http://spark.apache.org/
[3]http://storm.apache.org/



(a) *Service effect specification (SEFF) actions*



(b) *Resource environment*

Figure 6.1: *Meta-model extension for the Palladio component model (PCM, Version 3.4.1)*

big data systems are able to meet continuously increasing performance requirements and to serve several additional use cases. Consequently, up-front performance evaluations for these systems and capacity planning for building an appropriate cluster become not only inevitable, but also difficult and costly (Brunnert/Krcmar, 2017; Brunnert et al., 2014).

One way to approach these challenges are performance models such as the Palladio component model (PCM) that focuses on component-based software architectures (Becker/Koziolek/Reussner, 2009). It allows to model factors influencing system performance and predict performance metrics such as resource utilization, response time, and throughput by analytical solving or simulation (Brosig et al., 2015). As the PCM meta-model does not allow to model some specific requirements of big data systems yet, we propose and contribute a meta-model extension in this paper. This includes to specify an external call of an action to be executed multiple times in parallel while limiting the number of concurrent actions. It also includes to model a resource cluster consisting of several resource containers with different resource roles such as found in distributed computing architectures.

6.2 Modeling Big Data Systems

Comparing big data systems to ordinary component-based software systems (e.g., for web applications), they make use of specialized processing paradigms. Casado/Younas (2015) list two main techniques that are common for big data systems, namely, batch and stream processing. They have in common that they utilize parallel and distributed computing on a distributed system in the form of a computer cluster. For this purpose, software developers implement components with operation signatures, for instance by using software libraries such as Apache Hadoop, and combine these components to build a task job that will be deployed on a computer cluster. In order to distribute this task job across linked resources, the components can be assembled in the form of a directed acyclic graph (DAG) (Zaharia et al., 2012a). For instance, the *MapReduce* paradigm consists of two vertices *map* and *reduce* that are linked by a directed edge. By this means, such systems are able to make use of all distributed computing resources and achieve horizontal scalability for increased workloads in terms of data volume or velocity.

Despite their shared characteristics, batch and stream processing adopt distinct approaches and are designed for different use cases. Batch processing is intended to be used on data sets with high volume (Casado/Younas, 2015). In doing so, a specified operation is applied on splits of a non changing distributed data set multiple times in parallel. For instance, implemented Hadoop MapReduce operations are applied on distributed files on the HDFS. Implemented operations using Apache Spark are applied on so called resilient distributed datasets (RDD). The amount of parallelism for one specified operation is usually limited by the split rate of a dataset. The amount of simultaneously running parallel operations is usually limited by the amount of available resources or by specified user configurations.

Stream processing, on the other hand, is designated for handling high velocity data streams with low latency and is also referred to as real-time processing (Casado/Younas, 2015). It distinguishes itself from batch processing by not operating on a data set, but rather operating on each data (e.g., Apache Storm) or a mini-batch (e.g., Apache Spark Streaming) that are kept in-memory. Therefore, data are continuously received from an unbounded data stream (e.g., in a message queue manner) and immediately processed by an operation. Similar to batch processing, the number of simultaneously running operations is limited by the amount of available resources or by specified configurations.

In previous work (Kroß et al., 2015a) we already modeled one *MapReduce* job on a single computer and predicted its response time. As we had to simplify several features and take limitations into account, we identified the need to extend PCM. Based on these findings and the above mentioned characteristics of batch and stream processing, we derive the following requirements of big data systems that we propose to extend PCM in order to allow for modeling typical big data systems:

1. **Distribution and parallelization of operations**
   Component developers specify reusable software components consisting of operations using software frameworks like Apache Spark. In doing so, they may specify,

but also may not know the definite number of simultaneous and/or total executions of an operation.

2. **Clustering of resource containers**
   System deployers specify resource containers with resource roles (e.g., master or worker nodes), link them to a mutual network and logically group them to a computer cluster.

On this basis, we propose the following extensions for the PCM meta-model, which are shown in gray in Figure 6.1 (note that we only depict the relevant parts of the meta-model regarding our approach). The PCM meta-model consists of several partial models according to different developer roles (Becker/Koziolek/Reussner, 2009). Figure 6.1a shows the actions of the service effect specification (SEFF) model. A SEFF describes the behavior of an implemented operation. The element we propose to extend is the *ExternalCallAction* that is used to call a required service (Becker/Koziolek/Reussner, 2009). Therefore, we introduce a *DistributedCallAction*. It contains the two additional input parameters *simultaenousForkCount* and *totalForkCount* that can be used to specify the simultaneous and/or total number of executions of an external call as mentioned in the first requirement. Since these parameters depend on the workload and resource environment, component developers can describe the two input parameters as well as the resource demand of an operation as dependencies in parameterized form. In this way, domain experts are able to specify the usage of the component afterwards as proposed by Becker/Koziolek/Reussner (2009).

Figure 6.1b shows the meta-model extension for the resource environment. Here, a *ResourceContainer* may or may not have several *ProcessingResourceSpecifications* to specify e.g., processors and hard disks. A *ResourceContainer* can also have a set of nested *ResourceContainers*. We propose to complement the *ResourceContainer* by a *ClusterResourceSpecification* which contains references to one *ResourceRole* as well as one *SchedulingPolicy*. These are both part of a *ResourceRepository*, that is intended to contain types of resources such as for middleware and operating system resources (Becker/Koziolek/Reussner, 2009). A *ResourceRole* is used to describe whether a *ResourceContainer* represents a cluster, a master or a worker. A *SchedulingPolicy* is used to describe how actions are distributed on a cluster.

An example for a modeled computer cluster is shown in Figure 6.2. An outer *ResourceContainer* is used to connect computing nodes to a cluster and includes a round robin strategy to schedule actions on its nested *ResourceContainer*. This enables system deployers to simply allocate components to a cluster. Furthermore, each nested *ResourceContainer* includes a *ResourceRole*. If only workers are specified, the cluster will represent a shared-nothing architecture. If one master is specified, it will be responsible for distributing actions. Therefore, its *ResourceContainer* operates usually special middleware with additional resource demands that can be modeled with *InfrastructureCalls* in PCM.

6.3 Related Work

Most of the existing performance modeling approaches for big data systems concentrate only on one technology, namely Apache Hadoop. Barbierato/Gribaudo/Iacono (2014) introduce a performance modeling language to evaluate the performance of queries using Apache Hive which is a data warehouse software on top of Apache Hadoop with a SQL-like language. Vianna et al. (2013) propose an analytical model, which combines a precedence graph model and a queuing network model, to model MapReduce workloads concentrating on the pipeline parallelism between *map* and *reduce* operations. Verma/Cherkasova/Campbell (2014) propose a framework consisting of micro benchmarks and a regression-based model to predict and evaluate response times of MapReduce processes for different cluster resource choices.

A more general approach regarding big data technologies is, for instance, introduced by Castiglione et al. (2014) which use Markovian agents and mean field analysis to model big data batch applications and to provide information about performance of cloud-based data processing architectures. However, there is no approach available to the best of our knowledge that tries to enable modeling of general batch and stream processes, and to predict the response time and cluster resource utilization for their concurrent execution.

6.4 Conclusion and Future Work

In this paper we introduced a generic performance modeling formalism to model essential characteristics of data processing as found in big data systems. For this purpose, we presented two meta-model extensions for PCM that enable performance engineers to model a computer cluster and to apply distributed and parallel operations on this cluster. This allows to model general stream processing as well as batch processing techniques independent of their technology and to realize up-front performance evaluations for response times, throughputs, and resource utilizations of CPU and memory of big data systems.

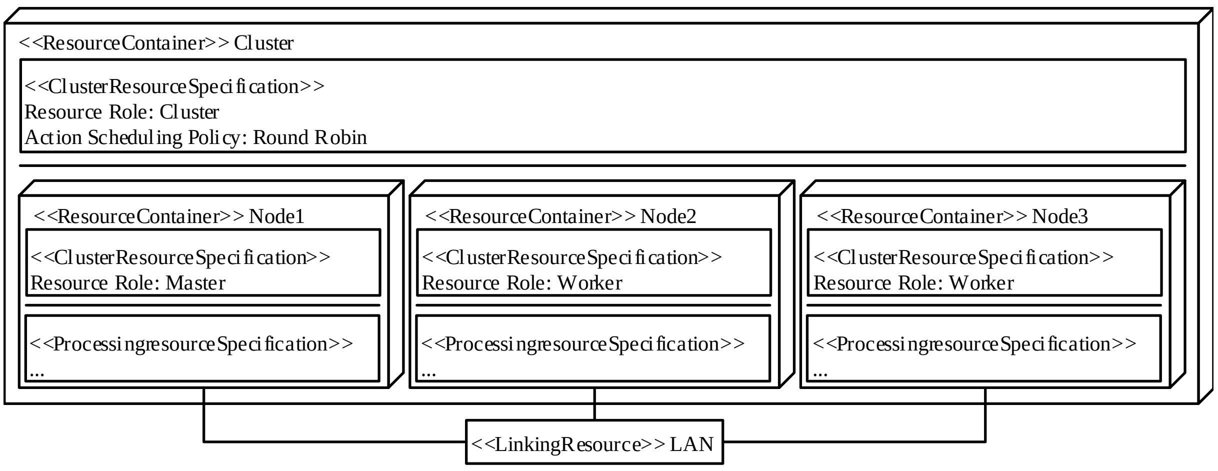

<<ResourceContainer>> Cluster
<<ClusterResourceSpecification>>
Resource Role: Cluster
Action Scheduling Policy: Round Robin
<<ResourceContainer>> Node1
<<ClusterResourceSpecification>>
Resource Role: Master
<<ProcessingresourceSpecification>>
...
<<ResourceContainer>> Node2
<<ClusterResourceSpecification>>
Resource Role: Worker
<<ProcessingresourceSpecification>>
...
<<ResourceContainer>> Node3
<<ClusterResourceSpecification>>
Resource Role: Worker
<<ProcessingresourceSpecification>>
...
<<LinkingResource>> LAN

Figure 6.2: *Example for a resource environment diagram*

We already implemented the meta-model extensions, graphical modeling editors, model-2-code transformations and basic functionalities of the associated simulation framework SimuCom (Becker/Koziolek/Reussner, 2009) to support our extensions. Although we do not consider network traffic between resource containers yet, first experimental results already look promising. In future, we plan to complete the SimuCom extension as well as integrate network traffic. Afterwards, we intend to comprehensively evaluate our meta-model extension in controlled experiments. This includes up- and downscaling scenarios regarding workload as well as resource capacities. Our long-term goal is to automatically derive performance models for batch and stream processes based on measurement data from middleware like Apache YARN.

Chapter 7

Modeling and Simulating Apache Spark Streaming Applications

Authors	Kroß, Johannes[1] (kross@fortiss.org) Krcmar, Helmut[2] (krcmar@in.tum.de) [1]fortiss GmbH, Guerickestraße 25, 80805 München, Germany [2]Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Softwaretechnik-Trends - Sonderteil: Proceedings of the Symposium on Software Performance (SSP)
Status	Accepted
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

Table 7.1: *Bibliographic details for P4*

Modeling and Simulating Apache Spark Streaming Applications

Abstract Stream processing systems are used to analyze big data streams with low latency. The performance in terms of response time and throughput is crucial to ensure all arriving data are processed in time. This depends on various factors such as the complexity of used algorithms and configurations of such distributed systems and applications. To ensure a desired system behavior, performance evaluations should be conducted to determine the throughput and required resources in advance. In this paper, we present an approach to predict the response time of Apache Spark Streaming applications by modeling and simulating them. In a preliminary controlled experiment, our simulation results suggest accurate prediction values for an upscaling scenario.

7.1 Introduction

Big data systems enable organizations to store and analyze data with high volume, velocity, and variety (Schermann et al., 2014). Corresponding processing techniques can be categorized into batch and stream processing (Hashem et al., 2015). Stream processing systems receive broad concentration nowadays as algorithms, transformations, and windowing mechanisms for streaming data constantly become more sophisticated and libraries for machine learning are available. They are mainly applied to process data and provide results in real time (Hashem et al., 2015). Therefore, the performance of such systems is particularly significant, for instance, to ensure high throughput for different workload scenarios and prevent queueing up of input stream data. However, planning the requirements of such applications and systems is complicated since environments and conditions to evaluate the performance for different scenarios, system configurations, and realistic workloads are usually not met as in productive environments (Brunnert et al., 2014; Kroß et al., 2015b).

We propose a modeling and simulation approach to predict the response time of stream processing applications i.e., Apache Spark Streaming. Therefore, we use the Palladio component model (PCM) (Becker/Koziolek/Reussner, 2009) and an extension for big data systems that we have presented in previous work (Kroß/Brunnert/Krcmar, 2015). The extension is open source[1] and includes a distributed call action to model parallel and distributed external calls in a service effect specification (SEFF) and a cluster resource specification to model a cluster of master and worker nodes and distribute resource demands to worker nodes.

7.2 Related Work

Regarding modeling, simulation, or analytical solving the performance of big data systems, most of prior research focuses on Apache Hadoop and its MapReduce paradigm and, therefore, on batch processing. There is one approach by Wang/Khan (2015), that focuses on predicting the response time of Apache Spark applications, however, only batch applications. Regarding stream processing, there is one patent by Ginis/Strom (2010) that describes a method on predicting the performance of publish-subscribe middleware messaging systems using queueing theory that, however, does not take resource demands for CPU, memory, or hard disk drives into account.

7.3 Modeling and Simulation Approach

There are several known stream processing systems available such as Apache Samza, Apache Storm, Apache Spark Streaming and Apache Flink (Hesse/Lorenz, 2015). We focus on Apache Spark Streaming[2] in this paper as it is one of the sophisticated technologies with a large community and supported by known benchmarks. It comprises a micro-batch model, in contrast to other technologies that use an operator-based model (Hesse/Lorenz, 2015).

A Spark Streaming application is constructed as follows[3]: it receives incoming data from streaming sources using a discretized stream called *DStream*. This data is fetched in the form of a micro-batch *job* that is iteratively executed in stream intervals. A *DStream* is represented by several resilient distributed datasets (*RDDs*). Afterwards, transformations such as *map* or *reduce* operations can be applied on a *DStream*. Spark builds a distributed acyclic graph (DAG) based on these related operations and splits them into *stages* of *tasks*. The number of parallel *tasks* is limited by the number of partitions of an *RDD*. Furthermore, transformations with narrow dependencies are consolidated in one *stage*, for instance, *map* and *filter* operations that do not require to shuffle data. *Stages* are executed sequentially and finally make up one *job*.

In order to model a Spark Streaming application, we specify one *job* executor component with a SEFF that involves a loop to start several asynchronous forked behaviours as displayed in Figure 7.1.

[1]https://git.fortiss.org/pmwt/bd.pcm.extension
[2]http://spark.apache.org/streaming/
[3]https://spark.apache.org/docs/1.6.0/streaming-programming-guide.html

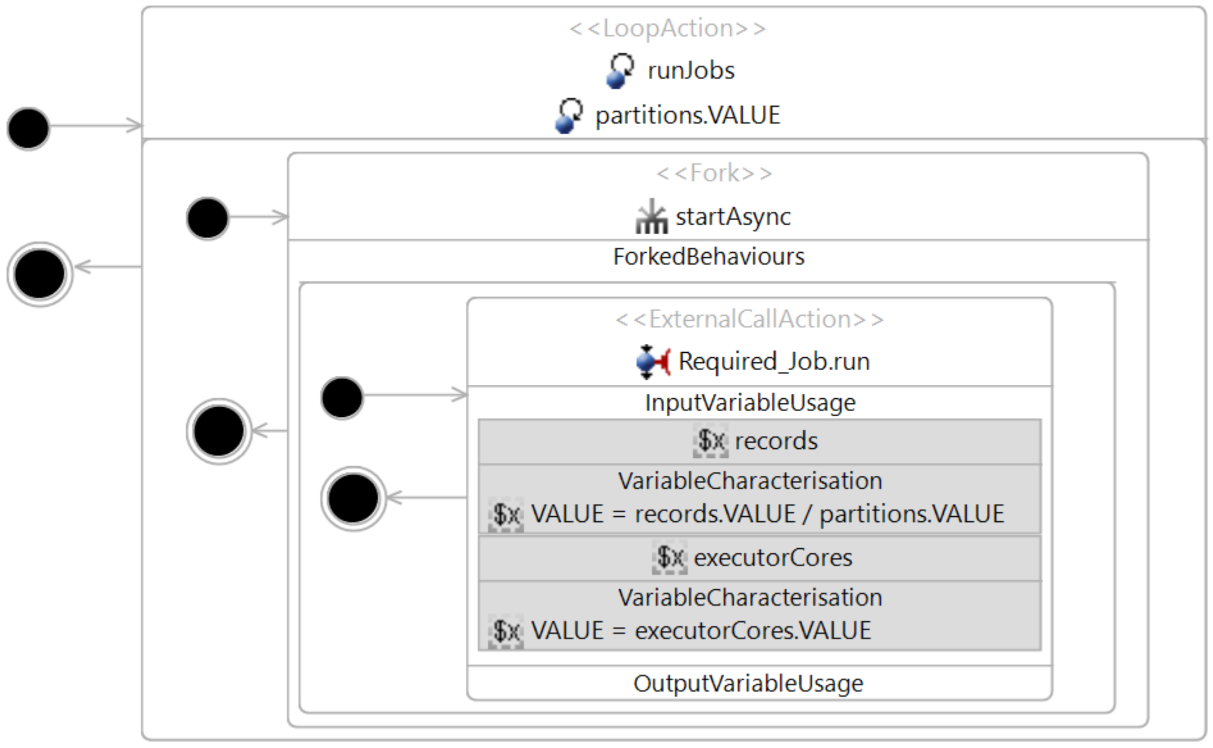


Figure 7.1: *SEFF of job executor component*

The loop length and, therefore, the number of executed behaviors depends on the value of the parameter *partitions* that is used to describe the number of topic partitions. In the forked behavior, we call the SEFF of the *stage* executor component with the parameters *records* and *executorCores*. The former parameter describes the number of records for each partition, the latter the number of cores that is configured when starting a Spark application with the equivalent parameter *spark.executor.cores*. The SEFF is illustrated in Figure 7.2.

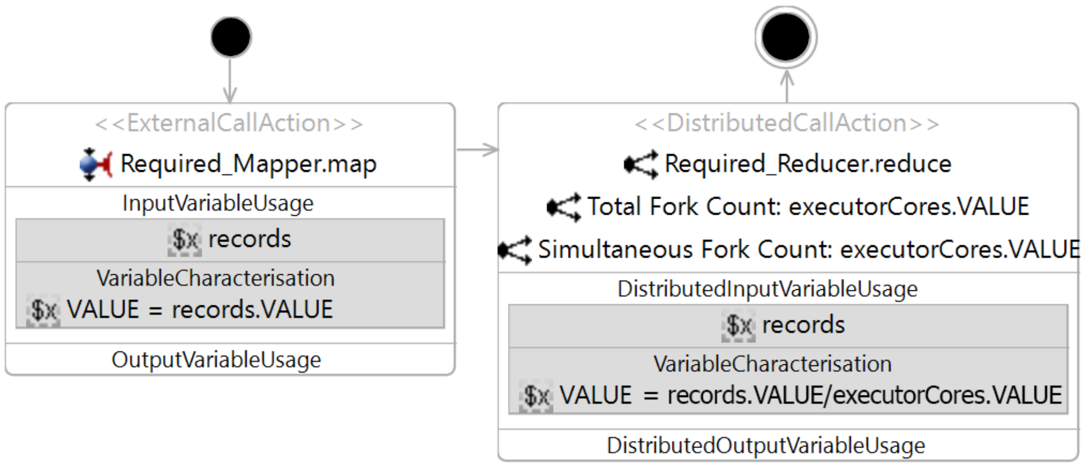

<<ExternalCallAction>>
Required_Mapper.map
InputVariableUsage
$x records
VariableCharacterisation
$x VALUE = records.VALUE
OutputVariableUsage
<<DistributedCallAction>>
Required_Reducer.reduce
Total Fork Count: executorCores.VALUE
Simultaneous Fork Count: executorCores.VALUE
DistributedInputVariableUsage
$x records
VariableCharacterisation
$x VALUE = records.VALUE/executorCores.VALUE
DistributedOutputVariableUsage

Figure 7.2: *SEFF of stage executor component*

For each *stage*, we model an external call or distributed call (Kroß/Brunnert/Krcmar, 2015), respectively, with the number of *records* as input parameter. Our sample application involves two stages *map* and *reduce*. The first SEFF *map* is invoked once since there is one *DStream* for each partition. The second SEFF is invoked with a distributed call of which the parallelism depends on the *executorCores* value. The *map* and *reduce* SEFFs involve three consecutive internal actions each with one parametric resource demands to specify the scheduler delay, serialization time, and computing time.

In order to model the hardware environment, we specify a resource container with a cluster resource specification (Kroß/Brunnert/Krcmar, 2015) for each node. For the master node, we model one parent resource container that includes a round robin action scheduling policy and a master resource role. Dependent on the number of worker nodes, we specify several nested resource containers with a worker resource role. In the usage model, we invoke the *job* executor component with its three input parameters, model a closed workload with one user, and specify the think time according to the stream interval.

7.4 Controlled Experiment

In our controlled experiment, we use the HiBench benchmark suite[4] of which we use the *distinct count* application. It involves two *stages map* and *reduce*. Therefore, data are streamed to a so-called *topic* in an Apache Kafka[5] cluster, a distributed publish-subscribe messaging system. The application is connected to that *topic* and applies a direct stream to query data from Apache Kafka using *DStreams*. Thereby, the level of parallel streams is defined by the number of partitions of one *topic* which, consequently, equals the number of *map stages*.

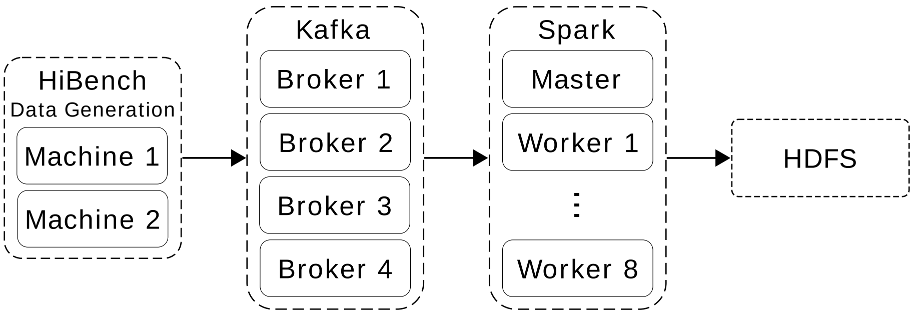


Figure 7.3: *Testbed setup*

Our experimental setup is shown in Figure 7.3. For the workload, we setup 2 virtual machines (VMs) that we use to generate data and a cluster consisting of 4 VMs for Apache Kafka brokers. For the application, we setup 1 VM for the master node, and 8 VMs for the worker nodes where the benchmark application will be executed. We use Apache YARN (2.7) as cluster manager and Apache Spark (1.6) as processing framework. We specified one Spark executor per worker node with 6 cores and 24 gigabytes memory.

We conducted four scenarios with a stream interval of 10 seconds as listed in Table 7.2.

Based on the 2 nodes scenario, we measured the delay and CPU resource demands for all *tasks* using the Spark monitoring API, adjusted the demands in dependence of the number of records, and included them into our repository model as listed in Table 7.3. We used this repository model for all upscaling scenarios. We adapted the number of workers in

[4]https://github.com/intel-hadoop/HiBench
[5]http://kafka.apache.org/

Table 7.2: *Conducted experiments*

Scenario	**Workload** (events/second)	**Kafka broker**	**Topic partitions**	**Spark worker**
2 nodes	$\sim$ 450,000	1	2	2
4 nodes	$\sim$ 450,000	2	4	4
6 nodes	$\sim$ 450,000	3	6	6
8 nodes	$\sim$ 450,000	4	8	8

the resource environment model and the partition parameter in the usage model for each scenario.

Table 7.3: *Parametric resource demands*

Map SEFF	
scheduler delay	10
deserialization	$0.000078549 * \text{records.VALUE}$
computing	$Norm(0.003320415 * \text{records.VALUE}, 0.0001553647 * \text{records.VALUE})$
Reduce SEFF	
scheduler delay	10
deserialization	$0.000013227 * records.VALUE$
computing	$0.000023370 * \text{records.VALUE}$

A boxplot of the measured response time (MRT) and the simulated response time (SRT) is illustrated in Figure 7.4. For the 2 nodes scenario, the mean MRT is 7.88 seconds and the mean SRT is 7.94 seconds, which gives a relative reponse time prediction error (RTPE) of 0.67%. In the 4 nodes scenario, the values deviate more with a RTPE of 21.14%. Our analysis of the measurements suggests that the processing time for each task did not behave as linear as in the other scenarios. In the 6 nodes and 8 nodes scenarios, the RTPE result in 3.41% and 2.26%.

Our models, simulation and measurements results, and analysis script are publicly available online (Kroß/Krcmar, 2016).

7.5 Conclusion and Future Work

In this paper, we proposed a modeling approach for stream processing systems using the example of Apache Spark Streaming. In a small controlled experiments, we simulated an upscaling scenario in which we increased the cluster size. Our predicted response times approach the measured ones closely.

At the moment, our extension for the simulation framework is for PCM 3.4.1 and we only consider delay and CPU demands. Therefore, we plan to incorporate our extension in the up to date PCM version and to additionally evaluate resources such as memory

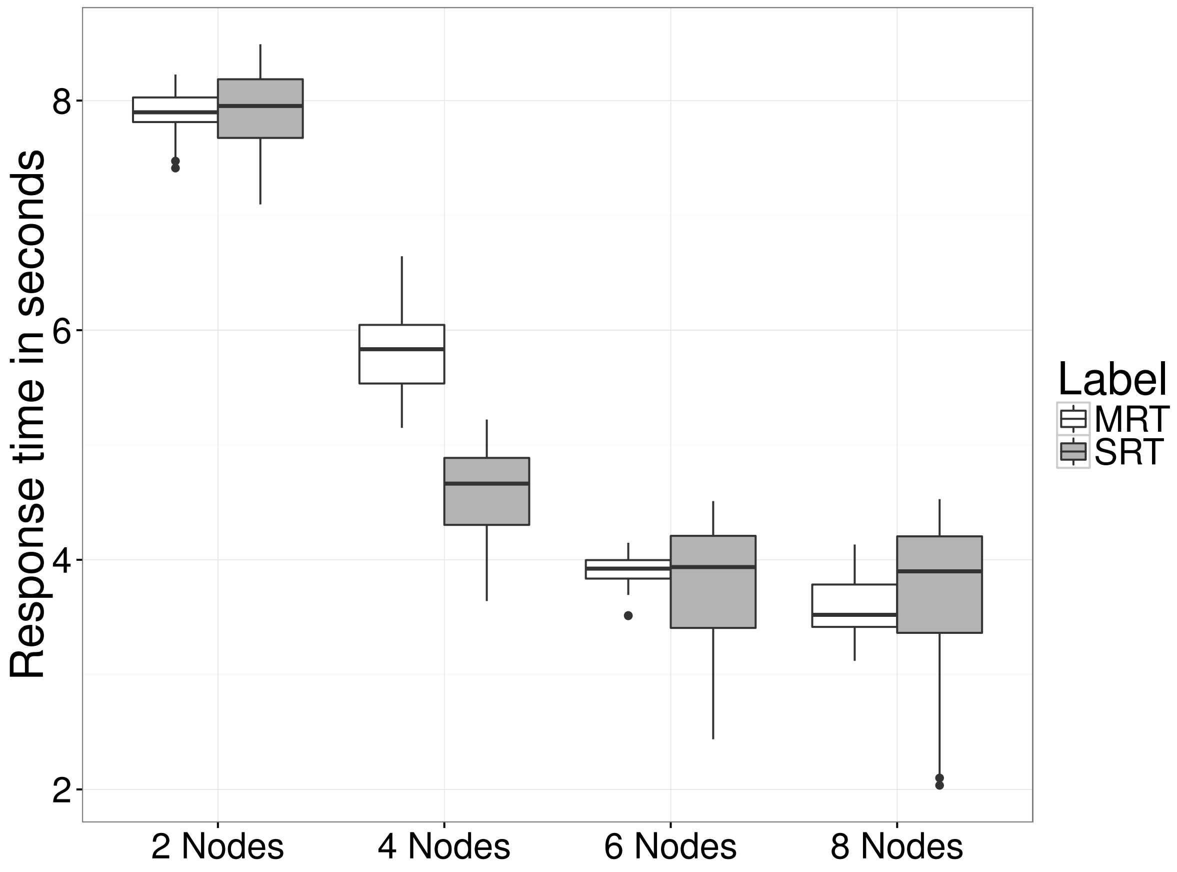


Figure 7.4: *Measured and simulated response times*

and network. Furthermore, we plan to extend our approach for operator-based processing frameworks such as Apache Flink and Apache Storm. Our long-term goal is to automatically derive performance models based on monitoring data, e.g., provided by APIs of processing frameworks.

Chapter 8

Model-Based Performance Evaluation of Batch and Stream Applications for Big Data

Authors	Kroß, Johannes[1] (kross@fortiss.org) Krcmar, Helmut[2] (krcmar@in.tum.de) [1]fortiss GmbH, Guerickestraße 25, 80805 München, Germany [2]Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Proceedings of the 25th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)
Status	Accepted
Keywords	Big Data, Performance, Modeling, Simulation
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

Table 8.1: *Bibliographic details for P5*

Model-Based Performance Evaluation of Batch and Stream Applications for Big Data

Abstract Batch and stream processing represent the two main approaches implemented by big data systems such as Apache Spark and Apache Flink. Although only stream applications are intended to satisfy real-time requirements, both approaches are required to meet certain response time constraints. In addition, cluster architectures continuously expand and computing resources constitute high investments and expenses for organizations. Therefore, planning required capacities and predicting response times is crucial. In this work, we present a performance modeling and simulation approach by using and extending the Palladio component model. We predict performance metrics of batch and stream applications and its underlying processing systems by the example of Apache Spark on Apache Hadoop. Whereas most related work concentrates on one specific processing technique and focuses on the metric response time, we propose a general approach and consider the utilization of resources as well. In different experiments we evaluated our approach using applications and data workloads of the HiBench benchmark suite. The results indicate accurate predictions for upscaling cluster sizes as well as workloads with errors less than 18%.

8.1 Introduction

The emergence of big data systems enabled organizations to store and process data with high volume, variety and velocity (Schermann et al., 2014). The Apache Hadoop family and the MapReduce paradigm paved the way for big data applications to be implemented in various areas across all industries (Dean/Ghemawat, 2008; Casado/Younas, 2015). Whereas these technologies were first designed to run on commodity hardware, frameworks such as Apache Spark arose and increased the performance of long-running applications. Their primary focus is to process a historical set of data in batches. Since there was also a need to analyze emerging data as they arrive, stream processing systems such as Apache Storm and Spark Streaming were developed in recent years.

The performance in terms of metrics such as response time, throughput, and resource utilization is a crucial aspect for both types of applications and depends on a variety of factors (Brunnert et al., 2014). It is vital, but also complicated to estimate the behavior and evaluate the impact of different scenarios such as changing data workload and resources (Wang/Khan, 2015). When deploying a big data application from a test to a production environment, for instance, data scientists are confronted with the challenge on how to size resource capacities in order to guarantee certain response times. Performance models represent an established way in order to address these challenges (Brunnert et al., 2015). They depict software systems, analytical solve or simulate their behavior, and predict different metrics (Brosig et al., 2015). Regarding big data applications, however, most related approaches focus on specific technologies (i.e., MapReduce) and processing types (i.e., batch). Furthermore, most efforts only consider the response time of applications in their approaches leaving out demands for resources. We propose and contribute a modeling and simulation approach for batch and stream processing systems by the example of Apache Spark. It includes resource demands and allows for predicting response times as well as resource utilization. Therefore, we use and extend the Palladio component model (PCM), a model designed for component-based software systems that represents performance-influencing factors on architecture-level (Becker/Koziolek/Reussner, 2009). Our extension and approach allows for simulating parallel operations as well as distributing them on a cluster of hardware resources. It supports big data architects to plan required capacities and examine the performance behavior under different conditions such as changing data workload.

In this paper, we first describe related literature in Section 8.2. In Section 8.3, we give an overview of batch and stream processing by the example of Apache Spark. Afterwards, we describe our modeling and simulation approach in Section 8.4. In Section 8.5, we assess the prediction accuracy of our approach and outline assumptions and limitations. Finally, we conclude our work and describe future activities in Section 8.6.

8.2 Related Work

Most of the former related work concentrates on the MapReduce paradigm or complementary database technologies such as Apache Hive or HBase. Many approaches also focus on the metric response time and do not consider resource demands and utilizations. Vianna et al. (2013) present a hierarchical model which combines a precedence graph mode as well as a queuing network model to predict the response time of MapReduce applications. They specifically focus on the intra-job synchronization delays between map and reduce tasks. Verma/Cherkasova/Campbell (2014) present a framework to predict the response time of MapReduce applications before migrating to a different cluster with different hardware. Therefore, they use micro-benchmarks on the initial cluster and a regression-based approach to model hardware differences between the initial and new cluster. Zhang/Cherkasova/Loo (2013a) present a framework including a platform performance model to depict different phases of a MapReduce application and predict the execution time in dependence on a new data set. For their approach they apply the model of the ARIA framework by Verma/Cherkasova/Campbell (2014).

Barbierato/Gribaudo/Iacono (2014) developed a language for the description of performance models which includes MapReduce applications. The approach allows to predict the response time. As main component the model uses the SQL-like query language of Apache Hive. Ardagna et al. (2016) proposed approaches to estimate response times of Hive requires. Therefore, they presented multiple performance analysis models with increasing complexity and accuracy such as queueing networks and stochastic well formed nets. Lehrig (2014) proposes an early design-time scalability/elasticity analysis of Software-as-a-Service (Saas) applications using architectural templates for Palladio. They plan to enrich it by big data technologies on the data layer such as replicable NoSQL databases and a MapReduce programming model. One general approach to model the behavior of batch applications is proposed by Castiglione et al. (2014). They use Markovian agents and mean field analysis to predict the behavior of concurrent interactive cloud, batch, and time constrained applications. However, they focus on cloud infrastructures and evolution dynamics of applications rather than on predicting performance metrics. Niemann (2016) present another approach to predict the performance and energy consumption of Apache Cassandra, a distributed data management system. They use queueing Petri nets for various workload as well as cluster sizes.

As part of the DICE EU project, Casale et al. (2015) propose a model-driven engineering for quality assurance of data-intensive software systems concentrating on Apache Hadoop, NoSQL databases, and stream processing (i.e., Apache Storm). Their approach aims at simulation, verification, and optimization for big data applications. The models contain three different model layers including a platform-independent model, a technology-specific model and a deployment-specific model (Guerriero et al., 2016). Gómez et al. (2016) also propose a strategy to transform the models into stochastic Petri nets. It shall enable engineers to asses performance requirements and they are currently validating their approach. For Apache Spark, Wang/Khan (2015) propose a simulation-driven prediction model that focuses on estimating response times. They also include read and write operations for hard disk drives (HDD) and the allocation of memory. Venkataraman et al. (2016) presented the framework Ernest for predicting the performance for analytical jobs using e.g., Apache Spark based on a optimal experiment design. Therefore, they predict the response time of applications ins dependence of the number of cluster nodes.

Regarding stream processing, there is one patent by Ginis/Strom (2010) that describes a method on predicting the performance of publish-subscribe middleware messaging systems using queueing theory that, however, does not take resource demands for CPU, memory, or hard disk drives into account.

8.3 Big Data Applications and Systems

There is a huge variety of big data solutions available that use different computing techniques (Chen/Zhang, 2014). In the following, we give an overview over batch and stream processing systems by the example of Apache Spark.

8.3.1 Batch Processing

Batch applications are designed to process a huge amount of historical data in a distributed and parallel way (Chen/Zhang, 2014). The Apache Spark framework is example for such applications and introduces so called resilient distributed datasets (RDDs) to keep and reuse data in memory. RDDs are parallel data structures to store intermediate results in memory and offer coarse-grained operations that can be applied on them (Zaharia et al., 2012a). An application is executed by forming a distributed acyclic graph (DAG) based on associated operations and grouping them into stages of tasks. A stage chains up operations with narrow dependencies in case a shuffle is not required (Zaharia et al., 2012a). The number of tasks of one stage depends on the number of RDD partitions. Stages are executed successively and constitute one job. One or more sequential jobs compose one Spark application. The application is orchestrated by one context, which runs in the main process called the driver program. It is responsible for allocating executors to worker nodes as well as scheduling tasks of an application on executors. An executor is a process that runs tasks in parallel. An application has always its own executors assigned in order to be isolated from other applications (Apache Spark, 2015).

8.3.2 Stream Processing

For applications that require to continuously analyze huge volumes of live data with low latency, stream processing systems are specialized for this purpose (Chen/Zhang, 2014). There are mainly two approaches - one mini-batch model and one continuous operator-based model (Hesse/Lorenz, 2015; Zaharia et al., 2012b). The former divides data streams into mini-batches and allows for batch processing, whereas the latter fetches and processes each record (e.g., Apache Flink) (Hesse/Lorenz, 2015). Apache Spark provides an extension module called Spark Streaming to apply the mini-batch model on data streams and reuse its core functionality. Therefore, Spark introduces discretized streams (DStreams). They allow for representing stream computations as a series of batch computations on mini-batch intervals and are represented as an ordered series of RDDs - one RDD for each interval (Zaharia et al., 2012b). Starting point of the data processing workflow is an input data stream that may be partitioned to increase parallel computing. Spark Streaming receives incoming data from such a stream source using a DStream and creates one RDD for each interval with the same amount of partitions as the input stream. Afterwards, transformations such as map or reduce operations can be applied on a DStream and RDD, respectively. As before, Spark builds a DAG based on related operations and splits these into stages of tasks. In contrast to batch processing, one job is created for each mini-batch. Jobs are continuously executed sequentially and always contain the same set of stages and tasks.

8.4 Modeling and Simulation Approach

This section first describes the extension for PCM. Afterwards, the derivation of the models and resource demands is outlined for batch applications followed by stream applications. Subsequently, the specification of cluster resources is described as well as the representation of data workloads.

8.4.1 Extending the Palladio Component Model

PCM enables engineers to describe performance relevant factors of software architectures (Brosig et al., 2015). It is implemented using the Eclipse Modeling Framework and consists of several models (Becker/Koziolek/Reussner, 2009). Software interfaces and components are specified in the repository model. Components provide the implementation for signatures of interfaces. Therefore, they contain a service effect specification (SEFF) in which the activities such as parametric resource demands and external calls of signatures are modeled. In the resource environment model, network and hardware resources are specified. The allocation model allows for deploying components on resources. The usage and workload is specified in the usage model.

In previous work, we already modeled and simulated big data applications (Kroß et al., 2015b; Kroß/Krcmar, 2016). Since PCM was not developed to support distributed and parallel computing as well as cluster architectures, we propose two meta-model extensions (Kroß/Brunnert/Krcmar, 2015). A *DistributedCallAction* is added to the SEFF meta-model. It extends the *ExternalCallAction* that is used to invoke a remote signature of a required service. The *DistributedCallAction* includes two variables *totalForkCount* and *simultaneousForkCount*. These specify the total number of executions of a remote signature call and the level of parallelism. Both variables can be specified using parametric dependencies . Furthermore, a *ClusterResourceSpecification* is introduced to complement a *ResourceContainer*. A *ResourceContainer* may represent a physical or virtual machine that hosts resources (e.g., CPU). The *ClusterResourceSpecification* contains two variables to reference a *ResourceRole* and a *SchedulingPolicy*. A *ResourceRole* is used to describe whether a *ResourceContainer* represents a cluster, a master or a worker. A *SchedulingPolicy* is used to describe how actions are distributed on a cluster.

For simulating models, PCM applies model to text (M2T) transformations to generate code that is used by the simulation framework SimuCom (Becker/Koziolek/Reussner, 2009). We reuse existing Palladio concepts to implement the M2T transformation of the *DistributedCallAction* as the following algorithm demonstrates.

```
1: $forks$ {array of length $simultaneousForkCount$}
2: $actionsPerForkCount \leftarrow totalForkCount/simultan\ eousForkCount$
3: for $i \leftarrow 0, simultaneousForkCount$ do
4:     $actions$ {array of length $actionsPerForkCount$}
5:     for $j \leftarrow 0, actionsPerForCountk$ do
6:         $actions[j] = createExternalCallAction$
7:     end for
```

```
 8:     $forks[i] = actions$
 9: **end for**
10: **return** $forks$
```

PCM supports modeling parallel calls of signatures from required services by using an *ExternalCallAction* inside a so-called *ForkedBehavior*. First, we create an array *forks* of type *ForkedBehavior* with length *simultaneousForkCount*. The parallel actions (or calls) per fork (*actionsPerForkCount*) are calculated by dividing *totalForkCount* by *simultaneousForkCount*. We fill each index of the array $forks$ with an array called *actions*. This array consists of *ExternalCallActions* according to the number of *actionsPerForkCount*. For example, if *totalForkCount* equals eight and *simultaneousForkCount* equals two, there will be two *ForkedBehaviors* and each will contain four consecutive *ExternalCallActions*.

For the *ClusterResourceSpecification* and its components, we implemented corresponding Java classes in the scheduler and SimuCom plugin of PCM. We also adapted the existing implementation of a simulated resource container to apply the scheduling of calls (i.e., round robin) on nested resource containers.

8.4.2 Modeling Batch Applications

We compose our components in the repository model similar to the DAG of an application, in this case Spark's DAG. We specify one application component as a starting point. It includes input parameters for the number of files, the size of one file, the default block size, and the number of executors. We model job components according to the number of Spark jobs. They are invoked sequentially by the application component with the same input parameters. Similarly, we model stage components corresponding with the number of Spark stages for each job. They are called sequentially by each job component and also receive the same parameters.

For each stage, we model one associated task component that will be invoked multiple times in parallel for which we use a *DistributedCallAction* (Kroß/Brunnert/Krcmar, 2015). Therefore, we model the first stage and the number of task executions different from the remaining stages. For the first stage, the number of task executions depends on the number of RDD partitions since input files are read from the storage layer. Spark will create a RDD for each input file and each RDD involves as many partitions as data blocks and splits, respectively. We use the above mentioned input parameters to specify the number of tasks and blocks n_{block} as the sum of data blocks for all files. In order to calculate the blocks for one file, we divide the size of a file x_{file} by the default block size x_{block} and take the ceiling in case there is a remainder ($x \in \mathbb{N}_{>0}$).

$$n_{block} = \sum_{i=1}^{n_{files}} \lceil x_{i,file} \div x_{block} \rceil \tag{8.1}$$

Furthermore, the input size for tasks of the first stage either match the default block size of the storage layer or the remainder split. Therefore, we specify a branch to include both cases and determine the probability $p_{defaultSplit}$ for a default block by dividing the amount of default blocks by the total amount of blocks.

$$p_{defaultSplit} = \frac{\sum_{i=1}^{n_{files}} \lfloor x_{i,file} \div x_{block} \rfloor}{n_{block}} \tag{8.2}$$

In contrast to the first stage, the data input for subsequent stages equals the output set of predecessor stages. We model these as a percentage of the dataset of the predecessor. Consequently, we do not need to specify two branches unlike for the first stage, but can directly call the task component. Additionally, the total number of tasks for subsequent stages depends on a fixed value configured by application developers. Since the configurable number of cores constitutes the limiting factor for concurrent tasks on a Spark executor, we specify an additional infrastructure component to model a pool of available cores. The component contains two SEFFs to acquire as well as release one core. In order to finally execute a task, a core must be acquired first and released after the task execution. The SEFF of the task component includes two consecutive resource demands, one for delay and one for CPU. In this work, we do not consider HDD demands and concentrate on CPU.

In order to estimate the function of the CPU demand, we profiled applications using the Java Management Extension (JMX) on the Java virtual machine of each Spark executor. We aggregated the CPU measurements of operations originating from *org.apache.hadoop.-net.unix.DomainSocketWatcher.run* for each stage during the application lifetime. We divided the combined measurements by the number of tasks of each stage to get the intercept of the function. In the same way, we transformed measurements for operations called by *org.apache.spark.scheduler.Task.run*. We additionally divided the latter value by the mean block size of the underlying dataset in order to the derive the slope of the function in dependence of the data size.

Regarding the delay demand, we used the Spark monitoring interface of the history server to calculate the mean response time of each task of a stage. We then subtracted the CPU demand per task to derive the delay. The monitoring interface also provides several metrics by itself. Although we also experimented to incorporate these metrics, the approach we described delivered more accurate prediction results for CPU and response time.

8.4.3 Modeling Stream Applications

The repository model for stream applications is also kept similar to the DAG of a Spark Streaming application as well as to our approach for batch applications. We model one application component as a starting point, which is intended to be triggered for each mini-batch interval. In contrast to the batch approach, we do not specify parameters in dependence on data sizes (i.e., megabytes), but in dependence on records. Therefore, the application component includes parameters for the number of records, the number of partitions of the data stream, and the number of executors. Since Spark Streaming creates one and the same job for each mini-batch, we create one job component. It is invoked by the application component using an asynchronous *ForkedBehavior*. In this way, the application component does not wait until the job component is finished and can be continuously triggered in time. According to the number of Spark stages, we model stage components that are called sequentially by the job component.

For each stage, we model one task component that will be invoked multiple times in parallel using a *DistributedCallAction* (Kroß/Brunnert/Krcmar, 2015). Similar to the batch approach, we model the first stage and the number of task executions different from the remaining stages. The initial number of stream partitions defines the number of RDD partitions and, therefore, the number of task executions for the first stage. For subsequent stages, a fixed value is used as parameter since it can be configured by engineers in the application configuration. The record input for subsequent stages equals the output set of predecessor stages. As before, we model these as a percentage of the dataset of the predecessor. For all stages, the final task will be invoked after a core is acquired, which will be released afterwards. Therefore, we also specify an infrastructure component to acquire and release available cores. The SEFF of the task component includes one delay demand and one for CPU demand. Both demands are calculated as for the batch approach but in dependence of the number of records.

8.4.4 Modeling Cluster Resources

In the resource environment model, we specify one parent *ResourceContainer* and multiple nested *ResourceContainer* depending on the number of workers. All containers are connected to a network via a *LinkingResource*. For each *ResourceContainer*, we model a *ClusterResourceSpecification*. For the parent *ResourceContainer*, we set a *MASTER* role and a *ROUND_ROBIN* policy. For the nested *ResourceContainer*, we configured a *WORKER* role. Additionally, processing resources (e.g., CPU) are added for each nested container.

8.4.5 Modeling Data Workload

The data workload is modeled in the usage model. For batch applications, the application component is invoked with four parameters. They specify the number of files that shall be processed, the size of each file, the default block size of the storage layer, and the number of Spark executors. A closed workload is used without any think time and with a population of one since there shall only one application to be executed. For stream applications, the SEFF of the application component is called with three parameters describing the sum of records within the stream interval, the number of stream partitions, and the number of Spark executors. Since the amount of records usually deviates slightly (e.g., due to network circumstances), a normal distribution was used to address this factor. We also specify a closed workload with a population of one. However, the think time is used to represent the time of mini-batch intervals. The application component is continuously invoked after the think time has elapsed.

8.5 Evaluation

In order to evaluate our approach we used the HiBench benchmark suite[1] to run sample applications in a test environment (Huang et al., 2010). Afterwards, we modeled and simulated these applications for selected scenarios and compared the measured and simulated response time and CPU utilization. We conducted four different scenarios - one upscaling scenario regarding cluster size and one upscaling scenario regarding data workload for batch as well as for stream processing each. In the subsequent Subsections, we describe our test environment setup, the evaluation of the batch scenarios followed by the stream scenarios.

8.5.1 Test Environment Setup

The hardware environment includes five IBM System X3755M3 servers, each consisting of four CPU sockets, 48 cores at 2.1 GHz in total, and 256 gigabyte (GB) random access memory (RAM). Each server is connected to a storage area network via Fibre Channel allowing for 8 gigabit per second (GBit/s). IBM System Storage EXP3512 is used for storing data. We virtualized each server using the VMware ESXi (5.1.0) hypervisor. We configured eight cores and 36 GB RAM for each virtualized machine (VM). On four servers, we allocated four VMs each that are used as worker nodes. On the remaining server, we allocated two VMs. One is used as master node and one for managing the cluster and initiating the benchmark applications. The following software is used on the VMs:

- CentOS Linux, 7.2.1511
- Oracle JDK, 1.8.0_60
- Apache Ambari, 2.4.2.0
- Hortonworks Data Platform, 2.5.3.0-37
- HiBench Suite, 6.0

Regarding HDFS we kept the default configurations including a replication factor of three and a data block size of 128 megabytes (MB). For YARN, we configured 26 GB and six virtual cores (vCores) per container, for Spark executors 22 GB as well as six cores.

8.5.2 Evaluating Batch Applications

We used the word count application of HiBench. It parses a set of input data and counts the appearance of each word (Huang et al., 2010). We conducted four upscaling experiments regarding cluster nodes and, similarly, four regarding data workload. Therefore, we

[1]https://github.com/intel-hadoop/HiBench

created one base repository model for the application. This model including its resource demands is used for all batch experiments. According to each experiment, the resource environment model and the usage model is adjusted. In order to evaluate the prediction accuracy of our approach, we consider the metrics response time and CPU utilization. For the simulation, we captured the simulated mean response time (SMRT) as well as the simulated mean CPU utilization (SMCPU) across the cluster. For the benchmark measurements, the applications were executed three times for each experiment and the measured mean response time (MMRT) as well as the measured mean CPU utilization (MMCPU) on user-level were calculated. The former is derived from the Spark monitoring API, the latter from the Ambari Metrics System. The results for all batch experiments are listed in Table 8.2 and the corresponding response times are illustrated in Figure 8.1.

Table 8.2: *Measurement and simulation results for batch applications (© 2017 IEEE)*

Cluster	**Workload**	**Response time** [milliseconds]			**CPU utilization**		
nodes	[gigabyte]	MMRT	SMRT	RTPE	MMCPU	SMCPU	CPUPE
4	28.72	104,599	103,218	1.32%	61.24%	61.95%	1.16%
8	28.72	68,205	62,233	8.76%	53.06%	55.77%	5.10%
12	28.72	54,984	52,632	4.28%	47.21%	47.5%	0.62%
16	28.72	50,140	47,181	5.90%	40.31%	42.18%	4.65%
16	57.36	74,657	69,583	6.80%	48.87%	48.22%	1.33%
16	86.08	101,675	93,292	8.24%	51.11%	49.66%	2.84%
16	114.72	119,977	122,740	2.30%	55.62%	52.07%	6.38%

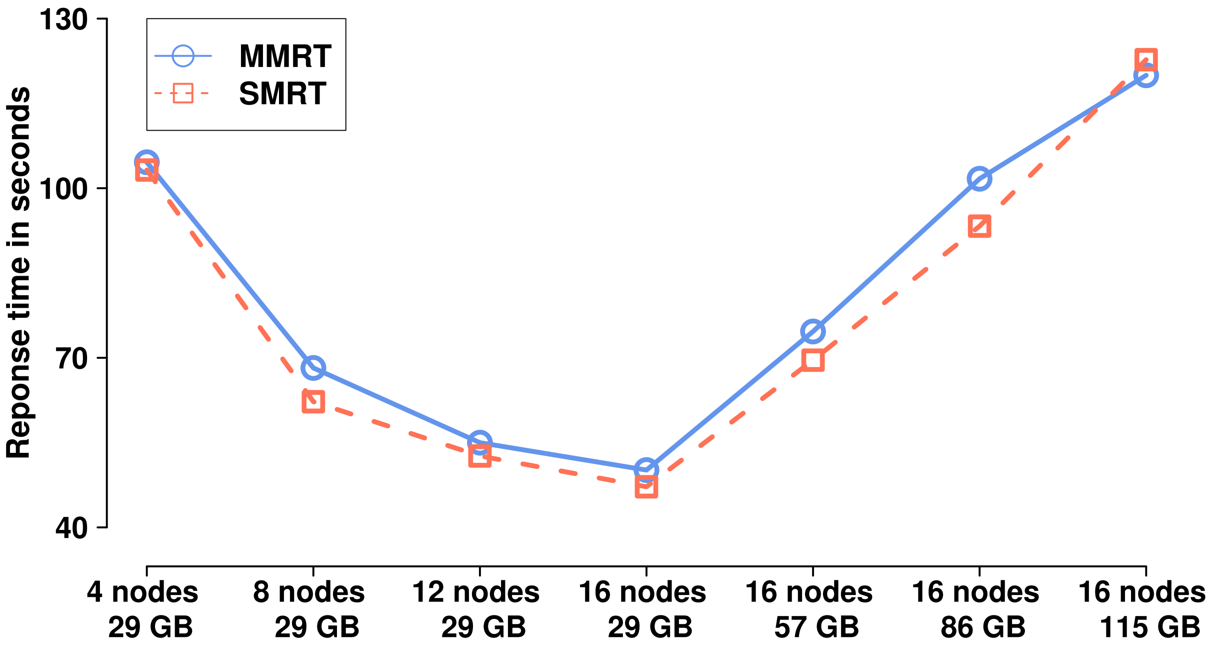


Figure 8.1: *Response times for batch applications (© 2017 IEEE)*

The starting experiment with four nodes resulted in a MMRT of 104,599 milliseconds (ms) and a SMRT of 103,218 ms leading to a relative response time prediction error (RTPE) of 1.32%. The MMCPU amounts to 61.24%, whereas the SMCPU lies at 61.95%, which gives a relative CPU utilization prediction error (CPUPE) of 1.16%. We resized the amount of nodes up to 16 nodes. Throughout, RTPE and CPUPE remained relatively low and were at most 8.76% and 5.10% both for the eight node scenario. On a cluster of 16 nodes, we additional increased the data workload. Similarly, we approximately resized the dataset by factors two, three, and four. Here, the RTPE was highest for a workload of 86.08 GB (8.24%), while the CPUPE deviated at most for 114.72 GB (6.38%).

Our approach showed to deliver relative prediction errors no more than 10% for both batch scenarios. While we slightly overestimated the CPU utilization values in the first scenario, we slightly underestimated them for the second scenario. Regarding response time, the prediction values were little lower than the measurement values in both scenarios except for one case.

8.5.3 Evaluating Stream Applications

We likewise used a word count application from the HiBench benchmark suite. The application involves stateful operators as well as checkpoints and acknowledgements. The application repeatedly fetches data from Kafka in a configured time interval of five seconds. We configured the environment so Kafka as well as Spark run exclusively on VMs. During the experiments, we adapted the number of Kafka brokers according to the number of Spark workers. Similarly, we adapted the number of Kafka partitions to always have six partitions on each Kafka broker to allow for optimal parallel processing of Spark (i.e., one partition per core (Marcu et al., 2016)). For our experimental results, we run the benchmark application and captured performance measurements for ten minutes leaving a ramp-up and ramp-down phase of five minutes.

We conducted four upscaling experiments regarding the cluster and four regarding data workload. Similar to the batch experiments, we derived a repository model including resource demands from the starting experiment and used this model for all simulations. The results are illustrated in Table 8.3 and the response times in Figure 8.2. For an interval of five second, the MMRT for the starting experiment is 3,006 ms and 31.81% MMCPU. The SMRT resulted in 3,029 ms and the SMCPU in 33.29%, which gives a RTPE of 1.21% and a CPUPE of 4.65%. With additional nodes, the relative prediction errors increased for both metrics and were at most for eight nodes (a scaling factor of four compared to the starting experiment). Here, the RTPE results in 17.02% and the CPUPE 13.43%. With increasing data workload, the RTPE and CPUPE decreased. In these experiments, we were not able to scale the workload by factor four since the input data could not be processed within the five second interval. While the SMCPU is constantly slightly lower than the MMCPU and the CPUPE behaves consistently, the SMRT is slightly too low for the last experiment as the MMRT increases abruptly. We conducted multiple experiments to further investigate the response time behavior of the application. We observed that the response times tend to rise rapidly as they converge to the time interval. Our prediction results still showed to provide accurate results with relative errors around 17% (Menascė/Almeida, 2002).

Our extension[2] as well as our models, simulation results, and measurements results are publicly available online[3].

Table 8.3: *Measurement and simulation results for stream applications (©2017 IEEE)*

Cluster nodes	**Workload** [events/second]	**Response time** [milliseconds] MMRT	SMRT	RTPE	**CPU utilization** MMCPU	SMCPU	CPUPE
2	100,000	3,066	3,029	1.21%	31.81%	33.29%	4.65%
4	100,000	2,363	2,515	6.43%	20.89%	19.91%	4.69%
6	100,000	2,124	2,358	11.02%	17.02%	15.44%	9.28%
8	100,000	1,956	2,289	17.02%	15.26%	13.21%	13.43%
8	150,000	2,754	2,820	2.40%	17.55%	16.16%	7.92%
8	200,000	3,296	3,350	1.64%	20.43%	19.10%	6.51%
8	250,000	4,614	3,880	15.91%	22.79%	22.04%	3.29%

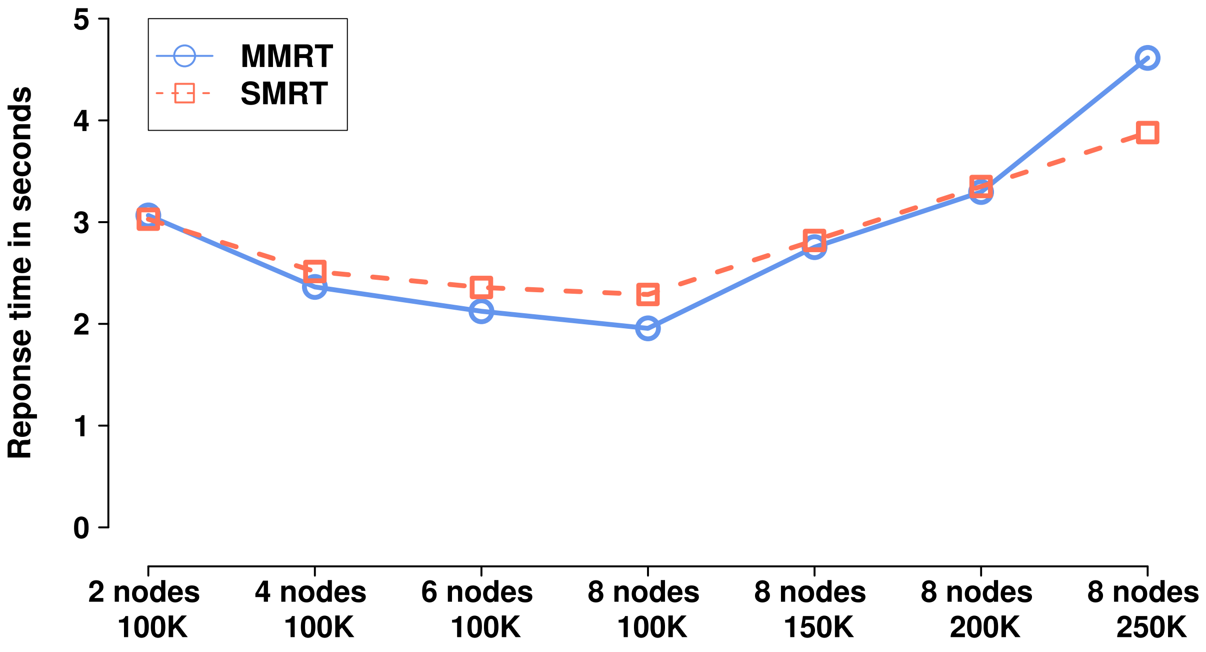


Figure 8.2: *Response times for stream applications (© 2017 IEEE)*

8.5.4 Assumptions and Limitations

In our experiments, we allocated one Spark executor on each node. It is also possible to size less cores and memory for spark executors and to allow for deploying multiple ones on one node. Although we are also able to model and simulate these scenarios, we did not evaluate such a case. We also evaluated our experiments in an exclusive cluster in which no other applications were running in parallel and using any CPU, HDD, or network. Regarding our modeling approach, we specified the input of a subsequent Spark stage probabilistically in dependence on the output data of a previous stage. Therefore, our

[2]http://git.fortiss.org/pmwt/bd.pcm.extension
[3]http://pmw.fortiss.org/research/ieee-mascots/

prediction error will increase, if the properties of the initial underlying data set change significantly. Furthermore, Heinrich/Eichelberger/Schmid (2016) discuss current problems such as modeling data structures and continuous data flows, but also potential solutions in modeling big data using Palladio.

8.6 Conclusion and Future Work

In this work, we presented an approach to model and simulate the performance behavior of batch as well as stream processing systems by the example of Apache Spark. Therefore, we extended PCM to represent resource clusters and distributed and parallel operations. This included a M2T transformation to generate corresponding simulation code and an adaption of the simulation platform SimuCom. We evaluated the approach by using sample applications of the HiBench benchmark suite. We conducted upscaling scenarios for cluster sizes as well as data workload both by factor four. Afterwards, we compared simulation with measurements values. The results suggest accurate predictions for response times and CPU utilization. For batch applications, the relative prediction error was at most 8.76% for response time and 6.38% for CPU utilization, for stream applications 17.03% and 13.43%.

Currently, we are experimenting with applying our approach for stream processing systems that implement an operator-based model. We also intend to represent resource demands for HDD. Furthermore, we plan to automatically derive models and resource demands for big data applications based on measurements. This shall support performance engineers by omitting the manual creation of models and ease the usage of our approach.

Chapter 9

PerTract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop

Authors	Kroß, Johannes[1] (kross@fortiss.org) Krcmar, Helmut[2] (krcmar@in.tum.de) [1]fortiss, Research Institute of the Free State of Bavaria, Guerickestraße 25, 80805 München, Germany [2]Technical University of Munich (TUM), Boltzmannstraße 3, 85748 Garching, Germany
Outlet	Big Data and Cognitive Computing
Status	Accepted
Keywords	Peformance Evaluation, Performance Modeling, Model Extraction, Performance Simulation, Big Data Systems
Individual Contribution	Problem and scope definition, construction of the conceptual approach, prototype development, experiment design, execution and result analysis, paper writing, paper editing

Table 9.1: *Bibliographic details for P6*

PerTract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop

Abstract Evaluating and predicting the performance of big data applications are required to efficiently size capacities and manage operations. Gaining profound insights into the system architecture, dependencies of components, resource demands, and configurations cause difficulties to engineers. To address these challenges, this paper presents an approach to automatically extract and transform system specifications to predict the performance of applications. It consists of three components. First, a system- and tool-agnostic domain-specific language (DSL) allows the modeling of performance-relevant factors of big data applications, computing resources, and data workload. Second, DSL instances are automatically extracted from monitored measurements of Apache Spark and Apache Hadoop (i.e., YARN and HDFS) systems. Third, these instances are transformed to model- and simulation-based performance evaluation tools to allow predictions. By adapting DSL instances, our approach enables engineers to predict the performance of applications for different scenarios such as changing data input and resources. We evaluate our approach by predicting the performance of linear regression and random forest applications of the HiBench benchmark suite. Simulation results of adjusted DSL instances compared to measurement results show accurate predictions errors below 15% based upon averages for response times and resource utilization.

9.1 Introduction

Big data frameworks are specialized to analyze data with high volume, variety, and velocity efficiently (Schermann et al., 2014). By distributing and parallelizing processing, they allow for horizontal scalability. Since the introduction of the MapReduce paradigm, there have been several frameworks released to support different types of applications, such as machine learning and stream processing. For all types, the performance of such software systems in terms of response time, throughput, and resource utilization is essential for a successful application (Brunnert et al., 2014). It is a difficult and complex task to manage and evaluate the performance for different scenarios such as changing data input and hardware resources (Wang/Khan, 2015).

Practical evaluations such as load tests on test systems are expensive. They require multiple experiments and only test a subset of configuration parameters. Additionally, they usually run with a reduced amount of data and resources. Thus, it is not able to draw accurate conclusions about the performance behavior. Performance models, on the other hand, provide an established evaluation approach by depicting performance characteristics of software systems and simulating their behavior or analytically solving them (Brosig et al., 2015). However, there are several challenges: creating models by hand is expensive, error-prone and slow as software systems are complex and continuously evolve (Brunnert et al., 2015). There is a lack of tool support for automatic model extraction. Regarding big data system, most related modeling approaches are also specific to a certain technology (i.e., Apache MapReduce) and only consider the response time of applications but not demands for resources (i.e., CPU).

In order to address these challenges, we propose a specification and model extraction approach for big data systems called PerTract to evaluate and predict the performance. We present a DSL to allow for modeling specifications on an architecture-level in a tool-agnostic way. To demonstrate our approach, we use Apache Spark for the application layer, in particular one random forest and one linear regression application that both use Spark's machine learning library. Additionally, we use Apache Hadoop for data provisioning and resource management. Figure 9.1 illustrates an overview of our approach. We extract execution components and inter-component interactions, resource landscape, and data workload in three separated specifications of a DSL instance using interfaces and logs of these technologies. In addition, we extract monitoring traces of applications (i.e., CPU times) and interrelate these with data workload information to identify parametric dependencies and estimate parametric resource demands of each execution component. On this basis, performance predictions are enabled. Therefore, we transform a DSL instance into a PCM (Becker/Koziolek/Reussner, 2009). Palladio is a model-based performance evaluation tool on the architecture-level that is supported by several analytical solvers and simulation engines.

Our approach provides several benefits. It integrates model-based activities, which are performed during development, and measurement-based activities, which are carried out during operations (DevOps) (Brunnert et al., 2015). The automated extraction process eliminates the effort to create models by hand. As applications are continuously updated, DSL instances can be extracted and tracked for each release as they evolve as well. This also enables engineers to continuously manage and plan required capacities and evaluate the performance for different scenarios (e.g., changing data workload) by adapting model parameters. Finally, it gives detailed insights about resource demands of execution components of an application and can be used to detect performance changes and regressions.

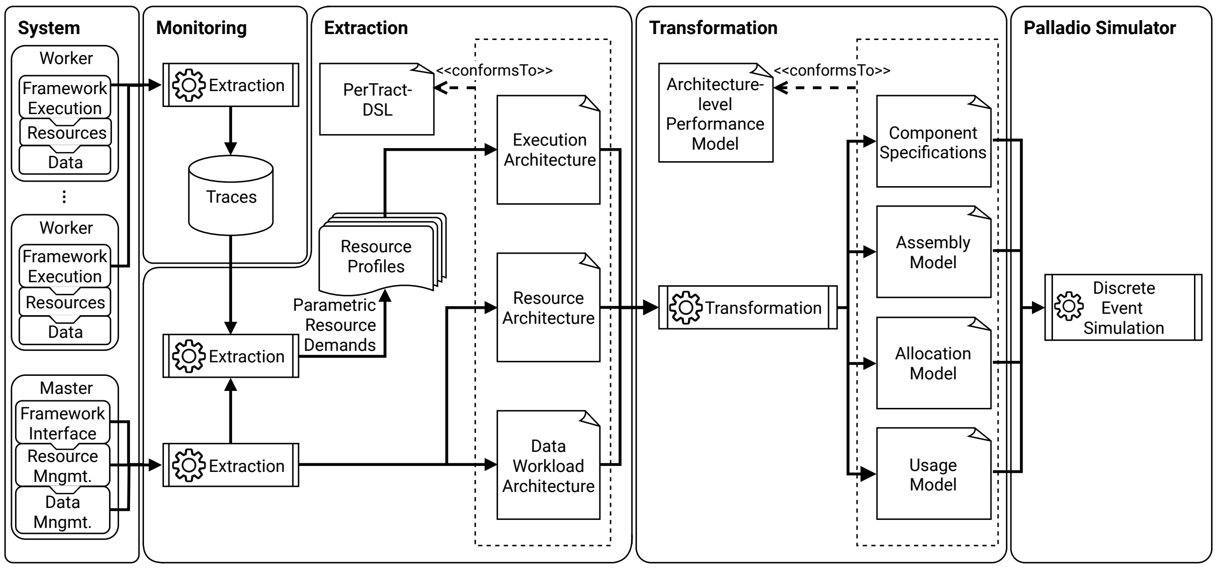


Figure 9.1: *Overview of the extraction and transformation approach.*

To sum up, the contributions of this paper are the following:

1. A DSL for modeling performance-relevant factors of big data systems,
2. An automatic extraction of system structure, behavior, resource demands, and data workload from Apache Spark and Apache Hadoop,
3. Transformations from DSL instances to model- and simulation-based performance evaluation tools,
4. Tool support for this approach.

To the best of our knowledge, our approach is the first white-box approach to extract performance-relevant metrics that allow for performance predictions of response times and resource usage. The developed tools are open source (Kroß) and extendable for extracting DSL instances from other frameworks and for transforming them to other model-based performance evaluation tools.

This paper builds upon our previous work (Kroß et al., 2015b; Kroß/Brunnert/Krcmar, 2015; Kroß/Krcmar, 2016; Kroß/Krcmar, 2017) on modeling and simulating the performance of big data applications and includes the following major improvements and extensions:

1. A formalism and DSL to model big data applications,
2. A lightweight Java agent to sample stack traces and CPU times from applications,
3. Automatic extraction of DSL instances,
4. Detailed evaluation against complex applications of the HiBench benchmark suite.

The remainder of this work is structured as follows: Section 9.2 describes related literature and approaches in the area of modeling and simulating big data applications. Section 9.3 introduces the model formalism as well as the DSL, which are required to understand this paper. Section 9.4 describes the extraction of DSL instances by the example of Apache Spark and Apache Hadoop. Section 9.5 presents the transformation to PCM models to allow for simulating the performance. Section 9.6 evaluates the prediction accuracy of our proposed approach for different upscaling scenarios and describes our assumptions and limitations. Finally, Section 9.7 outlines conclusions of our work and ideas for future activities.

9.2 Related Work

Since the Apache Hadoop family was the first widely-adopted big data framework, initial performance modeling approaches have been concentrating on this technology stack.

Vianna et al. (2013) predict the response time of MapReduce applications by introducing an analytical model, which they validated against an event-driven queuing network simulator. Their approach primarily concentrated on synchronization delays between map and reduce tasks. Verma/Cherkasova/Campbell (2014) introduce another approach for MapReduce. They developed a framework to allow for predicting response times before moving applications to different target platforms. The framework applies multiple benchmarks on source platforms and a regression-based model to relate the performance of source and the target platforms. Zhang/Cherkasova/Loo (2013a; 2013b; 2015) present multiple approaches where most of them are based on the analytical model by Verma/Cherkasova/Campbell (2014). Therefore, they additionally take heterogeneous clusters and configuration optimizations into account.

For other applications of the Hadoop family, Barbierato/Gribaudo/Iacono (2014) developed a language for the description of performance models. As a main component, the model uses the SQL-like query language of Apache Hive, a data warehouse built on top of Apache Hadoop. Ardagna et al. (2016) propose approaches to estimate response times of Hive requirements. Therefore, they presented multiple performance analysis models with increasing complexity and accuracy, such as queueing networks and stochastic well formed nets. They also considered unreliable resources in their experiments. Lehrig (2014) proposes a scalability and elasticity analysis of Software-as-a-Service applications at design time using architectural templates for Palladio. They plan to enhance it for big data paradigms on the processing layer and data layer.

Wang/Khan (2015) propose a prediction model for estimating response times of Apache Spark applications. In their approach, they consider demands for in-memory as well as demands for disk drives but not CPU processing. Another work by Ardagna et al. (2018) explores three modeling approaches for execution time prediction of Spark applications: one queuing network with a fork-join model and one with a task precedence model. Third, they present a discrete event simulation engine dagSim. The evaluation was conducted for different applications such as logistic regression and K-Means running in a public cloud.

Although the variance of the prediction accuracy is low for all approaches, the third approach delivers the most precise results.

Besides analytical and simulation-driven approaches, there are also approaches using machine learning for Apache Spark. Singhal/Singh (2018) evaluated different machine learning algorithms (i.e., multi linear regression and support vector machine) as well as an analytical model to predict execution times of Spark stages in development environments. They include multiple parameters from application logs into their models but only use execution times and do not consider resource demands. They also mention the drawback of machine learning approaches, which require intensive experiments and data collection. Furthermore, Venkataraman et al. (2016) present Ernest, a performance prediction framework for large scale analytics using machine learning kernels. It involves an automatic process to collect training data and to build a non-negative least squared model taking only a few parameters. They evaluate their approach on Amazon EC2 and show accurate predictions of execution times for increasing machine numbers. It is a black-box approach and does not give any insight into components of an application. As Ernest is bound to the structure of machine learning jobs, Alipourfard et al. (2017) present CherryPick, which intends to find best cloud configurations for various applications and use Bayesian optimization to create performance models. A configuration, for instance, contains parameters such as the number of virtual machines, CPU, and cores. In contrast to our work, they support additional types of applications (i.e., Spark SQL). Additionally, Witt et al. (2019) provide an extensive survey on performance prediction of batch processing using black box monitoring and machine learning.

Castiglione et al. (2014) propose a general approach to model the behavior of batch applications and concentrate on cloud infrastructures and evolution dynamics in terms of resource requirements and energy consumption. Therefore, they use an analytic modeling technique based Markovian agents and mean field analysis to describe the behavior of interactive cloud, batch, and time constrained applications. Niemann (2016) also presents an approach in the area of energy consumption. He focuses on Apache Cassandra, a distributed data management system, and uses queueing Petri nets to predict the performance and energy consumption of different workloads and platforms. Casale et al. (2015) propose a model-driven engineering for quality assurance of data-intensive software systems concentrating on Apache Hadoop and MapReduce, NoSQL databases, and stream processing (i.e., Apache Storm). Their approach aims at simulating, verifying, and optimizing architectures of big data applications. The models contain three different model layers including a platform-independent, a technology-specific and a deployment-specific model (Guerriero et al., 2016). Gómez et al. (2016) also shows an approach to transform these models into stochastic Petri nets, which is intended to allow for evaluating performance requirements. Lastly, Ginis/Strom (2010) hold a patent in the area of stream processing. The patent describes a method to model performance characteristics of publish–subscribe systems using queueing theory. However, the method does not include resource demands such as CPU, memory, and disks.

To summarize, the mentioned approaches focus on predicting the metric response time and often only implicitly assume resource demands for service executions per resource but do not link them to software components and operations (Brunnert et al., 2015). To the best of our knowledge, automatic model extraction in the area of big data are only supported by the mentioned machine learning approaches (Venkataraman et al., 2016; Alipourfard et al., 2017). However, these are black-box approaches and the models serve as interpolation of the measurements (Brunnert et al., 2015). Consequently, they do not model detailed information of the system architecture and dependencies and cannot be adapted for further evaluation scenarios. Finally, most of the mentioned models are technology-specific and, thus, are difficult to adapt and generalize them.

9.3 Modeling Approach

In this section, we describe the formalism for specifying big data systems. Afterwards, we present the PerTract-DSL based on the formalism.

9.3.1 Formalism

The specification consists of the following components:

- An *Execution Architecture* of the application, specifying nested directed graphs for execution components,
- A set of *Resource Profiles*, providing demands of different resources with parametric dependencies for the nodes of a graph,
- A *Data Workload Architecture*, specifying the underlying data model and type of data source
- A *Resource Architecture*, specifying a cluster of resource nodes, each with several resource units

9.3.1.1 Application Execution Architecture

The specification of the application Execution Architecture is a 2-tuple (c, n) where $c \in C$ is the application configuration and $n \in N$ specifies an initial node of the application.

A configuration $c \in C$ is represented by the 5-tuple $(p_d, e, ts_e, m_e, m_{ts})$ where p_d is the default parallelism for operations, more specifically tasks, of an application (e.g., join or reduce); e is the number of executors, which manage tasks; ts_e describes the number of tasks slots per executor that can be executed in parallel; m_e is the amount of main memory per executor that is available for tasks; and m_{ts} represents the amount of memory that each task slot requires to be allocated.

Nodes N are composite components. They can represent directed graphs $NG \subset N$ and execution nodes of a directed graph $NE \subset N$. In Figure 9.2, *ScalaWordCount* and *saveAsHadoopFile* represent a directed graph and *map* and *reduce* an execution node.

A directed graph $ng \in NG$ is a 2-tuple (N_{ng}, E_{ng}), in which N_{ng} is a set of nodes (or vertices) of the directed graph ng such that $ng \notin N_{ng}$; and E_{ng} is a set of directed edges. A directed edge $e \in E$ is represented by a 3-tuple (n_t, n_h, t_e), where $n_t \in N$ is the tail of e; $n_h \in N$ is the head of e; and $t_e \in \mathbb{R}_{\geq 0}$ specifies the factor of how many data are transmitted from n_t to n_h dependent on the amount of input data of n_t.

An execution node $ne \in NE$ is a 5-tuple (p_n, s, m, n_{ng}, rp) where p_n is the parallelism of node (e.g., some big data frameworks such as Apache Flink allow for specifying the parallelism for each operation individually); s indicates whether ne is a spout that is the node depending on partitioned data from an external source, such as a file system or messaging system; $m \in M$ is a reference to the dependent data model from the Data Workload Architecture; $n_{ng} \in NG$ references the parent directed node graph; and $rp \in RP$ describes the Resource Profile of ne.

9.3.1.2 Resource Profile

We use Resource Profiles to specify multiple resource demands. A Resource Profile $rp \in RP$ describes an ordered set of parametric resource demands RD. A parametric resource demand $rd \in RD$ is a 3-tuple (rt, f_{rt}, p) in which $rt \in RT$ represents the resource type and $f_{rt} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ is a function to specify the actual value of a resource demand in dependence on a parameter p (e.g., number of partitions of an input data source).

9.3.1.3 Data Workload Architecture

The model to represent the data workload is kept very simple. A Data Workload Architecture $d \in D$ is a singleton containing a set of data models M. A data model $m \in M$ contains one data source $ds \in DS$ element that consists of a parameter p_{ds} to specify the number of partitions.

9.3.1.4 Resource Architecture

A Resource Architecture $ra \in RA$ is a pair (nc, RN) in which $nc \in NC$ is a network channel and RN is a set of resource nodes. A network channel $nc \in NC$ is a 2-tuple (b, l) where b describes its bandwidth and l its latency. A resource node $rn \in RN$ describes a cluster node and is a 2-tuple (cs, RU) in which $cs \in CS$ is a cluster specification and RU is a set of resource units. A cluster specification $cs \in CS$ is described by a 2-tuple (rr, sp) where $rr \in RR$ describes a resource role (i.e., master node or worker node) and $sp \in SP$ the scheduling policy for distributing task across resource nodes (i.e., round robin). A resource unit $ru \in RU$ represents CPU, drive, and memory units.

9.3.2 PerTract-DSL

The PerTract-DSL follows the system model formalism described in the previous subsection and constitutes a language for specifying such models. Figure 9.2 illustrates an exemplary PerTract-DSL instance for a big data application. The PerTract-DSL is implemented as an Ecore-based meta-model using the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009). We use the DSL as an intermediate language to extract model instances and adapt its parameters for different scenarios. Afterwards, we generate architecture-level performance models that we use to simulate and predict the performance.

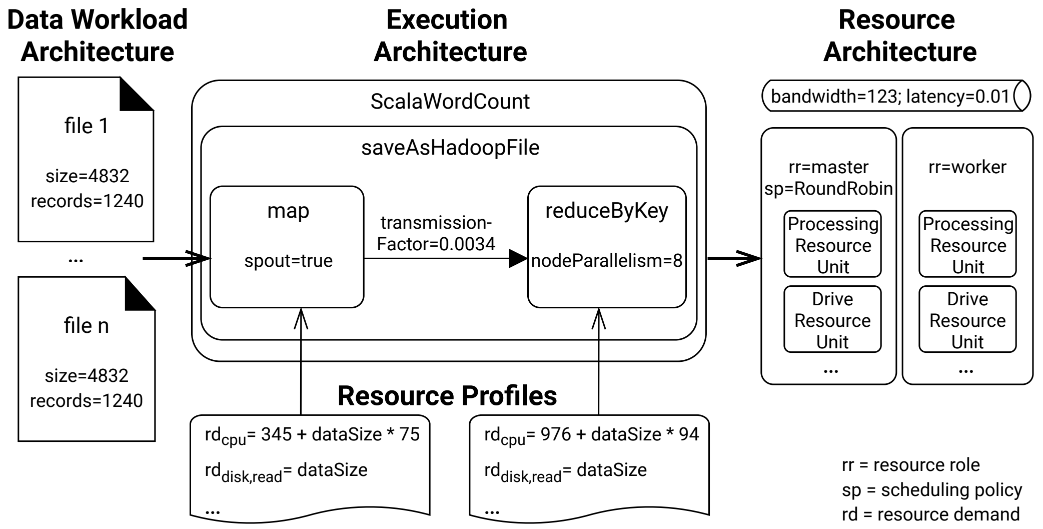


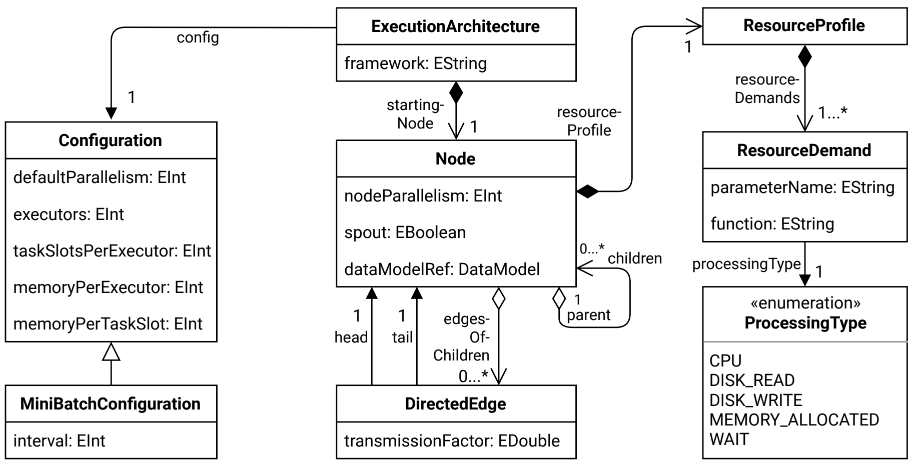
Figure 9.2: *Exemplary PerTract-DSL instance*

Figure 9.3a shows the classes and relationships of the Execution Architecture and Resource Profile. The Execution Architecture includes execution flows and operations on data and a configuration of an application. The configuration includes multiple parameters to specify the application settings. Depending on the application type (i.e., batch, mini-batch, and stream), a corresponding configuration type can be instantiated and may include additional parameters. For instance, a *MiniBatchConfiguration* involves an interval variable to indicate the mini-batch intervals.

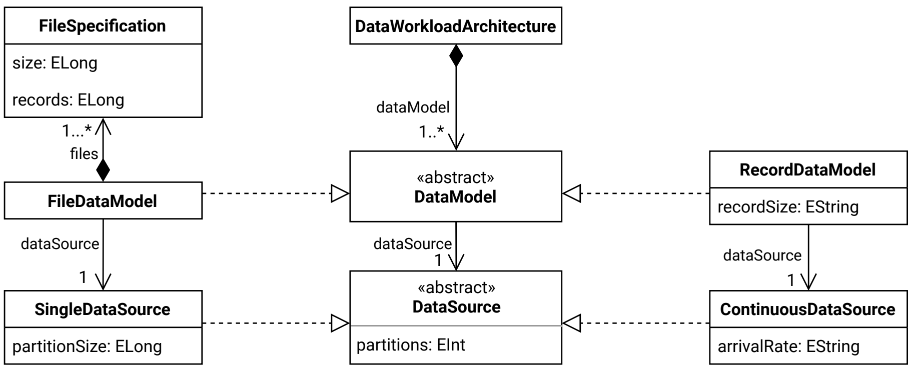
In order to specify operations on data and execution flows, we use nodes and directed edges (for instance, distributed acyclic graphs *DAGs* represent execution flows in Apache Spark, *topologies* in Apache Storm, and *job graphs* and *execution graphs* in Apache Flink). Therefore, a *Node* is a composite that can represent two roles—a directed graph that contains several nodes (*children*) and edges, and an execution node that executes tasks. In the latter case, a node contains a Resource Profile for its tasks.

The term Resource Profile describes a set of resource demands for transactions of an application (King, 2004; Brandl/Bichler/Ströbel, 2007; Brunnert/Krcmar, 2017). This includes resource demands for CPU, disk, memory, and network usage. Resource Profiles have been used for transactions for a specific workload and specific servers (King, 2004; Brandl/Bichler/Ströbel, 2007) but also for component operations within the control flow of each transaction independent of their deployment topology (Brunnert/Krcmar, 2017).

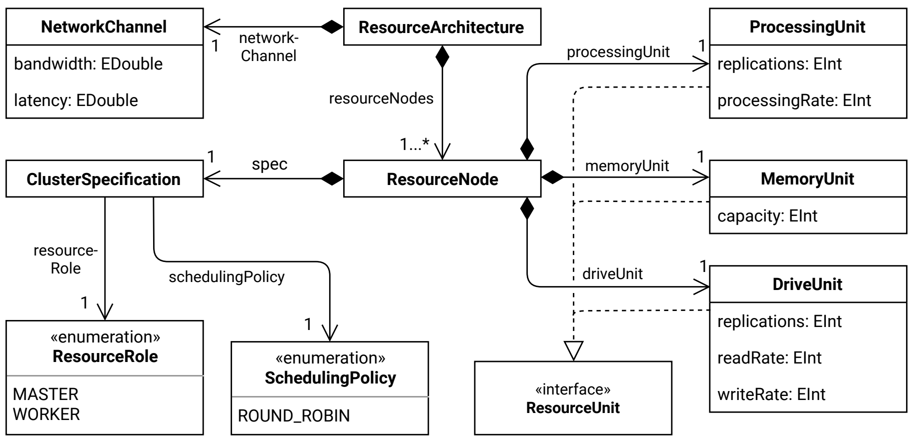
Branches with probabilities for its occurrences represent operation control flows. As yet, the related approaches do not use parametric dependencies and use Resource Profiles in the area of enterprise applications, where the workload is mainly user-driven and the resource demands for operations may remain static for each user. In our case, operations highly depend on incoming data volume either dependent on the data size or number of records.



(a) *Execution Architecture and Resource Profile*



(b) *Data Workload Architecture*



(c) *Resource Architecture*

Figure 9.3: *PerTract-DSL classes and relationships*

We change the notion of Resource Profiles for our purposes in three ways. First, we include parametric dependencies. Second, we do not model the control flow and probability as this information is contained in the directed graph. Third, we do not apply a Resource Profile on the same fine granularity level of operations except for a set of operations and tasks. Big data frameworks chain and group single operations together and transform each grouping into a set of tasks, which will eventually be executed multiple times in a distributed way. The number of executed tasks usually depends on the number of partitions. As we model data and hardware resources as first-class entities in dedicated specifications, the exact number and distribution of operations depends on them. Therefore, we apply a Resource Profile on a group of chained operations. It forms the basis to derive tasks with resource demands and predict the performance by combining them with data workload and Resource Architectures.

While considering data as first-class entities, we focus on specifying only performance-relevant factors of data as presented in Figure 9.3b. A Data Workload Architecture contains one or several data models, which are either file-based (e.g., for batch applications) or record-based (e.g., for stream applications). The former contains multiple file specifications and a single data source, which specifies the partition size of the files and the number of partitions. The latter contains a variable to indicate the mean record size and a continuous data source, which describes the number of partitions of a data stream as well as the arrival rate per second of one record.

Figure 9.3c illustrates an overview of the classes and relationships of the Resource Architecture. It is a simplified version based on the resource environment model of PCM including our extension (Becker/Koziolek/Reussner, 2009; Kroß/Brunnert/Krcmar, 2015). It contains several resource nodes that, combined, represent a cluster. Each resource node contains a processing unit, memory unit, and drive unit with individual processing rates or capacities. The resource demands of one Resource Profile will be performed on the corresponding resource units of one resource node.

9.4 Extracting Model Instances by the Example of Apache Spark, Apache YARN and Apache HDFS

Since creating models for applications, data, and resources requires much effort, we propose an approach to automatically extract PerTract-DSL instances based on monitoring measurements and logs. The remainder of this section describes the approach to extract a DSL instance in detail, comprising the monitoring on application level (Section 9.4.1), the extraction of Execution Architectures from applications (Section 9.4.2), the estimation of Resource Profiles for stages of applications (Section 9.4.3), the derivation of Data Workload Architectures (Section 9.4.4), and the extraction of hardware resources (Section 9.4.5).

9.4.1 Extraction of Resource Demands

Collecting measurement data is necessary in order to extract Resource Profiles, estimate resource demands, and calculate parametric dependencies. Profilers provide a common way to extract fine-grained data such as stack traces and CPU times. We examined multiple Java profilers but found that the performance of big data applications is significantly increased by their overhead. Therefore, we chose a sampling approach and developed a lightweight Java agent for sampling CPU values for either stack traces or thread groups of long-running applications.

Algorithm 1 shows the main procedure of the agent. It collects samples in intervals of 100 milliseconds, which we found to cause only low overhead while still providing high accuracy in our experiments. Therefore, the agent fetches a dictionary of thread identifiers and corresponding stack traces by calling the *getAllStackTraces()* method provided by the Java *Thread* class. The dictionary contains only entries for threads that are in an active state at the point of time requested. The CPU time is collected for each thread by using the ThreadMXBean management interface (i.e., the *getThreadCpuTime(long id)* method) for monitoring of the Java Virtual Machine (JVM). The CPU times for thread groups with the same names will be summed up and sent as a batch to an Apache Cassandra repository. Additionally, the name of the JVM will be transmitted to the repository for each measurement.

Algorithm 1: Sampling thread groups and CPU values

```
Output: $samples \leftarrow$ dictionary containing a timestamp as key and tuples of
        thread groups and CPU times as value
Schedule new thread every 100 milliseconds
    $threadGroups \leftarrow < k : String, v : long >$;
    $sampleTime \leftarrow$ current timestamp;
    /* procedure provided by Java */
    $threads \leftarrow$ getAllStackTraces();
    for $thread$ to $threads$ do
        /* procedure provided by Java */
        $cpuTime \leftarrow$ getThreadCpuTime($thread.id$);
        $threadGroup \leftarrow thread.threadGroup$;
        $threadGroups[threadGroup] \leftarrow cpuTime + threadGroups[threadGroup]$);
    end
    $samples \leftarrow (sampleTime, threadGroups)$;
Until $application\ has\ terminated$;
```

9.4.2 Extraction of Execution Architectures

The Apache Spark framework introduces so-called resilient distributed datasets (RDDs). RDDs are parallel data structures to store intermediate results in memory and offer coarse-grained operations, which can be applied on them and work the same way on all data items (Zaharia et al., 2012a). Spark offers several operations and transformations such as *map* and *reduce*.

A Spark application is executed by forming a DAG based on associated operations and grouping them into stages of tasks. A stage chains operations with narrow dependencies, which means a shuffle operation is not required e.g., a *map* and a subsequent *filter* operation (Zaharia et al., 2012a). The number of tasks of one stage depends on the number of RDD partitions. Stages are executed successively and constitute one job. One or more jobs compose one Spark application. The application is managed by one context. It runs in the main process called the driver program. It allocates executors to worker nodes and schedules and assigns tasks of an application on to executors. An executor is a process that executes the tasks and operations in parallel (Apache Spark, 2015).

In order to automatically extract execution components and inter-component interactions from Apache Spark, we access the interfaces of the embedded history server. We remind readers that we refer to the specification introduced in Section 9.3.1. We use the Spark environment properties to derive an Application Configuration. We set p_d to *spark.default.parallelism*, e to *spark.executor.instances*, ts_e to *spark.executor.cores*, and m_e to *spark.executor.memory*. While a DAG created by Apache Spark models RDDs as nodes and operations as edges, we create nodes on three levels—on application-, job- and stage-level—and data flows as edges (similar to the JobGraph of Apache Flink).

On the application-level, one initial node is created to represent the application itself (i.e., *ScalaWordCount* in Figure 9.2). It contains a set of child nodes and edges for the job-level.

On the job-level, we read the interface for job metrics of the corresponding application and create a set of nodes containing one element for each job entry. As jobs may be executed in parallel, we consider the chronological sequence of jobs by accessing start times and end times in order to create a set of directed edges and connect successive nodes. The data transmission factor of each edge is calculated by bringing the input data of the tail and head in dependence:

$$dt_e = \frac{input_{n_t}}{input_{n_h}}. \tag{9.1}$$

Each job node contains a set of child nodes and edges for the stage-level. On the stage-level, we access the interface for stage metrics of the corresponding application and create a set of nodes containing one element for each stage entry corresponding to one job. In order to derive the parallelism p_n of each node and whether it represents a spout s_n, we obtain the read data metrics of each stage and distinguish between *input* and *shuffle* data:

$$s_n = \begin{cases} true, & \text{for } \mathit{input} > 0 \land \mathit{shuffle} = 0, \quad (9.2a) \\ false, & \text{otherwise}, \quad (9.2b) \end{cases}$$

$$p_n = \begin{cases} p_{ds}, & \text{for } \mathit{input} > 0 \land \mathit{shuffle} = 0, \quad (9.3a) \\ p_d, & \text{otherwise}. \quad (9.3b) \end{cases}$$

In case a stage has read input bytes, the initial RDD of the stage is created by an external data source and contains as many partitions as the data source. This usually applies to each initial stage of a job. For this case, we set s_n to true (Equation (9.2a)) and specify p_n according to the number of partitions of the data source p_{ds} (Equation (9.3a)). In case a stage has read shuffled data, the corresponding RDD of the stage is already transformed based on prior RDDs. Its partitions equal the default parallelism p_d. Therefore, we set s_n to false (Equation (9.2b)) and set p_n to p_a (Equation (9.3b)). The data transmission factor is calculated as in (Equation (9.1)). Finally, we extract one Resource Profile for each node element on the stage-level.

9.4.3 Extraction and Estimation of Resource Profiles

A Resource Profile consists of a set of resource demands where each element may involve a different resource type and a function to specify the value. Our main focus lies on the CPU resource. As Ousterhout et al. (2015) systematically identified by the example of Apache Spark, CPU is the bottleneck of data analytics applications in most cases contrary to the widely-accepted opinion that disk and network are weak points.

We define three different CPU demands for each stage $i \in EN$. The first one represents the actual time to process a task. We define a linear function dependent on the parameter p describing the data size for each task of a stage. The slope of the function is calculated by using aggregated CPU times originating from task-related thread groups across all Spark executors. This CPU time is divided by the total amount of read data for each stage:

$$f_{i,cpu,task}(p) = p\frac{cpuTime_{i,task}}{input_i + \mathit{shuffle}_i}. \tag{9.4}$$

The second CPU demand represents the overhead of coordinating with the driver program, preparing a task before it is actually executed, and postprocessing. These times are provided by the Spark task metrics interface (i.e., they are included in the variables *executorDeserializeTime* and *resultSerializationTime*). As the coordination grows with the number of Spark executors, we define the demand dependent on the configuration parameter e, the number of executors. We observed that this demand varies very strong from task to task, especially for the first tasks of a stage. As averaging the metric is not reasonable, we model this demand by converting the series of time values to a boxed probability density function (PDF) with variable interval sizes as specified by PCM (Becker/Koziolek/Reussner, 2009). In order to box the CPU values, we use the percentiles 5, 25, 50, 75 and 95 as intervals since they are provided by the Spark's interface.

The third CPU demand represents the overhead caused by providing infrastructure services for one task. As it is independent of data input, we define a static demand using aggregated CPU times of traces originating from worker-related thread groups across all Spark executors. We additionally divide the CPU times by the total number of tasks to receive the demand for one task:

$$f_{i,cpu,infra} = \frac{cpuTime_{i,worker}}{numComplTasks + numFailTasks}. \tag{9.5}$$

For the extraction of drive demands, we examined several approaches to estimate read and write demands. As we are not able to measure the drive demands on an appropriate level without adding instrumentation to HDFS (similar to Ousterhout et al. (2015)), we extract only a resource demand for reading data, which equals the dependent parameter p describing the data size for each stage.

Similarly, network demands on a low granularity level are only able to be retrieved by instrumenting Spark in a sophisticated way. In order to compensate and include the time delays caused by network traffic, we extract *wait* demands. We calculate the delays between stages by comparing their start and end times and model the demand accordingly.

Furthermore, we do not extract demands for allocating main memory at the moment. As simulation approaches for memory are still limited and neglect features such as garbage collection, the prediction accuracy of this resource is debatable (Brunnert/Krcmar, 2017).

9.4.4 Extraction of Data Workload Architectures

The Hadoop distributed file system (HDFS) is a distributed, scalable, and fault-tolerant storage system for big data (Apache Hadoop, 2015). Files are split into a sequence of blocks according to a specified block size, which are are replicated to different data nodes to support fault tolerance (Apache Hadoop, 2015). For instance, if Spark applications read a file from HDFS, it will be represented by one RDD with as many partitions as blocks.

In order to extract the Data Workload Architecture, we create a file-based data model and a single data source for a specified folder in HDFS and create a file specification for each file. To access the required information, we use the client library of Apache Hadoop. We access the size of each file as well as calculate the partition size and number of overall partitions p_{ds}.

9.4.5 Extraction of Resource Architectures

Cluster managers, such as Apache Hadoop YARN and Apache Mesos, arbitrate resources for batch and stream applications and provide support to distribute them on cluster nodes. YARN stands for Yet Another Resource Negotiator and follows a master–worker architecture (Apache Hadoop, 2015). This includes one resource manager and multiple node managers. A node manager runs on each worker node and is responsible for executing resource containers. A resource container is an abstract notion for resources such as CPU, memory, and HDD in which application tasks run (Apache Hadoop, 2015). If a new application is submitted, a responsible application master will be executed in a new resource container. It orchestrates application tasks and, therefore, requests resource containers from the resource manager and monitors their state (Dean/Ghemawat, 2008). Apache Spark is able to run in different modes on YARN. In the so-called client-mode, for instance, the driver program and Spark context runs at the client itself, the application master requests resources for executors, and each executor will run in its own resource container (Apache Spark, 2015).

In order to extract Resource Architectures, we use the public interface provided by YARN to retrieve metrics of each cluster node. For each node manager, we create one resource node $rn \in RN$. Therefore, we assign a worker resource role and create a resource unit for each CPU, drive, and memory. The CPU cores and memory capacity are extracted via the interface. As drive information is not available, we set the read and write speed manually (e.g., by testing HDFS with the included DFSIO benchmark).

Besides the set of resource nodes, we create a network channel and also set the bandwidth and latency manually.

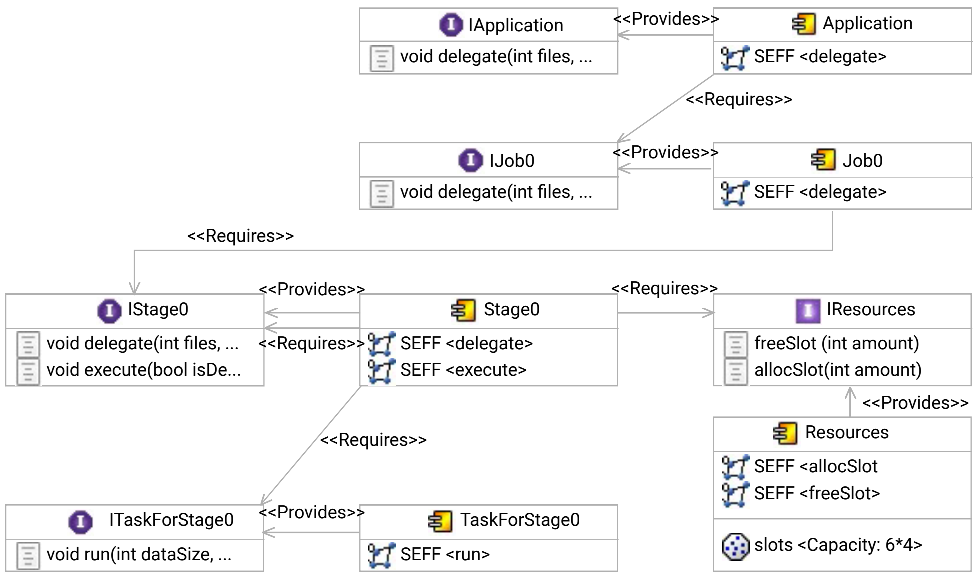
9.5 Transformation to Performance Models

This section describes the concepts of the architecture-level performance model PCM and how we transform DSL instances into PCM models.

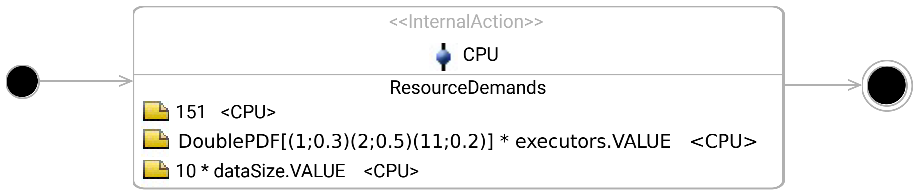
9.5.1 Palladio Component Model

We chose to use PCM (Becker/Koziolek/Reussner, 2009) as a model-based performance evaluation tool as it enables engineers to specify software systems independent of technology, include resource demands for software components, consider resource contention, and predict not only response time, but also resource utilization. Furthermore, the tool support is mature, open source, and continuously maintained with a large community.

In particular, PCM is developed for component-based software systems and enables engineers to describe performance relevant factors of software architectures, resource environments, and usage behavior (Brosig et al., 2015). It is implemented in Ecore from the Eclipse Modeling Framework (EMF) and consists of multiple models (Becker/Koziolek/Reussner, 2009). Software interfaces and components are specified in the Repository Model (Figure 9.4a). Components provide the implementation for signatures of interfaces. Therefore, they contain a resource demanding service effect specification (RDSEFF) in which the activities such as parametric resource demands and external calls of signatures are modeled similar to activity diagrams (Figure 9.4b). Components are additionally assembled in a System Model. In the Resource Environment Model, network and hardware resources are specified such as processing resources (CPU, disk, and delay), processing rates, and scheduling policies. The Allocation Model allows for deploying assembled components from the System Model on resources from the Resource Environment Model. The usage and workload of software components are specified in the Usage Model. Finally, PCM provides a simulator for its models, which is based on a process-oriented discrete event simulation.



(a) *PCM Repository model example*


<<InternalAction>>
CPU
ResourceDemands
151 <CPU>
DoublePDF[(1;0.3)(2;0.5)(11;0.2)] * executors.VALUE <CPU>
10 * dataSize.VALUE <CPU>

(b) *Resource demanding SEFF for a task (PDF probability density function*

Figure 9.4: *Exemplary transformed PCM instances*

9.5.2 Transformation to PCM

We describe the transformation for each DSL component. Table 9.2 shows the mapping of DSL concepts to PCM elements. An Execution Architecture is transformed to a Repository Model (Figure 9.4a). In order to traverse the Edges and Nodes of an Execution Architecture, we use a recursive depth-first search. Upon visiting each Node, we check if it contains child Nodes and Edges. If this is the case, we again traverse this Node and the procedure repeats.

For each Node, we create one Interface with several signatures and a corresponding Basic Component that *provides* the signatures using an RDSEFF. If a Node contains child Nodes, we add a *delegate* signature to the corresponding Interface (i.e., *IJob0*). Additionally, the Basic Component *requires* the Interfaces of the child Nodes.

Parameters of the Configuration and parametric dependencies of the Execution Architecture are transformed into input parameters of each Signature. We consider parameters for the number of files, the data size of one file, the default partition size, the number of partitions, and the number of executors. In order to model and limit the maximum number of concurrent tasks, we separately specify an Infrastructure Component to represent a pool of available task slots. The component contains two SEFFs to acquire and to release a task slot. In order to finally execute a task, a slot must be acquired first. After task completion, the slot is released again. In the case of Apache Spark, the limiting number of task slots is the number of total cores.

Table 9.2: *Mapping of PerTract-DSL to PCM elements.*

***PerTract*-DSL**	***PCM Model Elements***
Execution Architecture	Repository Model
Nodes	Interface, Basic Component
Edges	RDSEFF
Configuration	Parameters, Infrastructure Component
Resource Profile	Distributed Call Action, RDSEFF
Resource Architecture	Resource Environment Model
Resource Node	Resource Container
Cluster Specification	Cluster Specification
Network Channel	Linking Resource
Data Workload Architecture	Usage Model
Data Model	Entry Level SystemCall, Parameters
Data Source	Workload

RDSEFF Resource Demanding Service Effect Specification; *Distributed Call Action, Cluster Specification* PCM extensions (Kroß/Brunnert/Krcmar, 2015).

Edges are represented in the RDSEFF of a Basic Component. Each *delegate* RDSEFF models the flow by using External Call Actions to invoke signatures of required Interfaces in the specified order (i.e., *Job0* invokes the *prepare* signature of *IStage0*). In the course of this, the input parameters are forwarded and altered at specific points to model the data transmission factor t_e of an Edge.

If a Node contains a Resource Profile, we transform it by creating several model elements. In order to call a group of tasks in parallel, we add two signatures to the corresponding Interface of the Node (i.e., *Stage0*). The providing RDSEFF *prepare* is intended to create a set of parallel tasks. It uses a Distributed Call Action to invoke the *execute* signature of the same Interface several times in parallel. The parallelism is either defined by the number of partitions of a data source p_{ds} or the specified parallelism of the Node p_n. The *execute* RDSEFF acquires and releases a task slot before and after prompting a task.

We create an additional Interface and Basic Component (i.e., *TaskForStage*) to model a task. Its behavior *run* is responsible to execute the parametric resource demands of a task (Figure 9.4b). Only the *wait* demand of a Resource Profile will be executed in the prior *prepare* RDSEFF as the demand occurs once at the beginning of each stage and not for each task. We automatically assemble all Basic Components of the Repository Model in order to derive Palladio's System Model.

Since the Resource Architecture follows the concepts of Palladio's Resource Environment Model, the transformation is linear. We transform each Resource Node to a Resource Container and convert the Cluster Specification and Resource Role accordingly. Additionally, we transform each Resource Unit to an equivalent Processing Resource Unit including the specification of processing rates, number of replicas (e.g., the number of cores), and scheduling policies. Finally, all Resource Containers are connected to networks via a Linking Resource.

In order to create the Allocation Model, we deploy all assembled Basic Components from the System Model on the master Resource Container from the Resource Environment Model. Our previous extensions (Kroß/Brunnert/Krcmar, 2015) enable Palladio's simulation framework SimuCom to distribute resource demands to Resource Containers that represent worker nodes with a round robin policy.

Finally, we transform the Data Workload Architecture to a Usage Model. We create one Entry Level System Call that invokes the *delegate* signature of the *Application* Interface. The required input parameters are transformed based on the Data Model and Data Source. We specify the number of files, the data size of one file, the default partition size, and the number of partitions. For the Single Data Source, we create a simple closed Workload with a population of one, which means the Entry Level System Call is triggered once.

All transformed models can be used by Palladio's simulator to predict performance metrics.

9.6 Evaluation

This section evaluates the model extraction and performance simulation approach introduced in this work.

9.6.1 Research Methodology

In order to validate our approach, we conduct three integrated controlled experiments by modeling and simulating the execution of two different exemplary machine learning applications (Hevner et al., 2004). Therefore, we formulate three claims by exemplary problems from a performance management perspective.

First, engineers are interested in the performance behavior of applications and resources in case data workload grows. This experiment evaluates the claim that data workloads can be changed independently of Execution Architectures and Resource Architectures. We initially extract one PerTract-DSL instance for each of the two applications based on monitoring data. Afterwards, we adapt data sizes in Data Workload Architectures and compare predictions for response times and CPU utilization with corresponding monitored measurements in several upscaling scenarios.

Second, engineers need to evaluate the scalability of applications if additional hardware resources are allocated. This experiment evaluates the claim that resources can be altered independently of Execution Architectures and Resource Architectures. We modify and add worker nodes in Resource Architectures without changing Execution Architectures and Data Workload Architectures. Afterwards, we compare predictions results with corresponding monitored measurements.

Third, engineers need to efficiently plan and manage capacities for given data workloads and performance requirements (Brunnert et al., 2015). This experiment evaluates the claim that data workloads as well as resources can be changed independently of Execution Architectures. Similarly, we use the models extracted in the first experiment and conduct several upscaling scenarios regarding data workload and cluster size without modifying Execution Architectures. Afterwards, we compare the simulated prediction results with corresponding measurements.

9.6.2 HiBench Benchmark Suite

In our experiments, we apply the HiBench benchmark suite to run representative and reproducible applications and workloads for Apache Spark (Huang et al., 2010). As the automatic extraction approach shall allow for modeling complex applications, we use two machine learning applications. We chose a random forest classification (RFC) since random forests represent frequently used machine learning models for classification and regression. HiBench implemented the application using Apache Spark's machine learning library MLlib and provides an RFC-specific data generator. Additionally, we chose a linear regression (LR) as it is a common approach for regression analysis and forecasting. Therefore, HiBench's implementation uses a model without regularization using a stochastic gradient descent to predict label values. Similarly, it implements Spark's MLlib and includes its own data generator.

9.6.3 Experiment Setup

Table 9.3 and 9.4 illustrates our testbed and data configurations. The hardware environment includes five servers. Each server is connected to a storage area network (IBM System Storage EXP3512, New York, United States) via fibre channel allowing for eight gigabits per second (GBit/s). The servers are also connected in a local area network (LAN) with one GBit/s.

Table 9.3: *Software and hardware configuration of the test system*

Software platform	4x	Hortonworks Data Platform (2.6.3.0-235) - Apache Spark (2.2.0) - Apache Hadoop (2.7.3) - Apache Ambari (2.6.0)
Java virtual machine		Oracle JDK (1.8.0_60)
Operating system		CentOS Linux (7.2.1511)
Virtualization		VMware ESXi (5.1.0), 8 cores, 36 GB RAM
CPU cores	5x	48 x 2.1 GHz
CPU sockets		4 x AMD Opteron 6172
Random access memory (RAM)		256 gigabyte (GB)
Hardware system		IBM System X3755M3

We virtualized each server using the VMware ESXi hypervisor (VMware, Palo Alto, United States) and configured eight cores and 36-gigabyte (GB) memory for each vir-

Table 9.4: *Data workload scenarios and configurations*

Application	Scenario	File Size	Files	Partitions	Total Size
Random forest classification	Small	1.89 gigabyte	8	128	15.12 gigabyte
	Large	3.58 gigabyte	8	232	28.64 gigabyte
	Huge	5.52 gigabyte	8	360	44.16 gigabyte
Linear regression	Small	1.86 gigabyte	8	120	14.88 gigabyte
	Large	3.49 gigabyte	8	224	27.92 gigabyte
	Huge	5.59 gigabyte	8	360	44.72 gigabyte

tualized machine (VM). On each server, we allocated four VMs. On the first server, we use one VM as a master node for Apache HDFS and YARN, one VM for managing the cluster (i.e., Apache Ambari), one VM for storing monitoring data, and one VM for initiating the benchmark applications. On the remaining four servers, we use each VM as a worker node. We deployed the Hortonworks Data Platform to use Apache Spark, YARN, and HDFS. For HDFS, we kept the default configurations including a replication factor of three and a data block size of 128 megabytes (MB). For YARN, we configured 26 GB and six virtual cores (vCores) per container, for Spark executors 22 GB as well as six cores. Since we experienced that not all cores were utilized when running applications, we changed the resource calculator to be dominant and enabled CPU scheduling to address this issue. For evaluating the prediction accuracy, we compare the metrics response time and CPU utilization. For simulations, we captured the simulated mean response time (MRT) as well as the simulated mean CPU utilization (MCPU) across the cluster. For the benchmark measurements, applications were executed four times for each experiment to avoid any distortions. Similarly, monitored MRT and monitored MCPU on the user-level were calculated. Monitored response times are derived from the Spark monitoring API and monitored CPU measurements from the Ambari Metrics System (2.6.0).

Tables 9.5 and 9.6 list all simulated and monitored MRT and MCPU results, the root mean square errors (RMSE), and the relative prediction errors. They provide the basis for presenting and discussing our experiments in the following.

9.6.4 Collecting Resource Demands and Extracting Execution Architectures

The extraction and transformation process follows the overview illustrated in Figure 9.1. In order to extract an Execution Architecture for one application, we monitor the application using our profiler presented in Section 9.4.1 to extract stack traces and corresponding CPU times. Additionally, the Spark framework itself monitors an application. As described in Section 9.4.2, execution components and inter-component interactions are extracted using Spark's interfaces. For each execution component, CPU resource demands are generated by processing corresponding CPU times and interrelating them with data input information of each component as explained in Section 9.4.3.

In order to evaluate the three proposed claims, we derive one initial PerTract-DSL instance for each of the two machine learning applications that we use throughout all experiments. According to each experiment and scenario, we adapt the PerTract-DSL instance and simulate it to derive predictions.

Table 9.5: *Monitored and simulated mean response times (seconds).*

		Random Forest Classification Application				Linear Regression Application			
Worker Nodes	Data Workload	Monitored MRT	Simulated MRT	RMSE	Prediction Error	Monitored MRT	Simulated MRT	RMSE	Prediction Error
4	Small	264.79	262.71	4.47	0.78%	42.15	43.09	1.19	2.23%
	Large	502.09	462.41	40.26	7.90%	71.96	76.60	4.73	6.45%
	Huge	755.05	696.70	59.65	7.73%	124.21	116.38	13.39	6.30%
8	Small	222.46	199.04	24,92	10.53%	35.28	32.95	2.65	6.59%
	Large	378.31	322.54	56.62	14.74%	52.24	49.74	3.66	4.79%
	Huge	534.12	486.34	48.48	8.94%	76.73	73.54	4.60	4.15%
16	Small	196.62	196.46	4.34	0.08%	37.84	37.33	2.22	1.34%
	Large	287.38	285.20	11.56	0.76%	40.86	45.24	4.48	10.74%
	Huge	373.74	396.38	25.97	6.06%	53.27	56.96	4.05	6.93%

Table 9.6: *Monitored and simulated mean CPU utilization.*

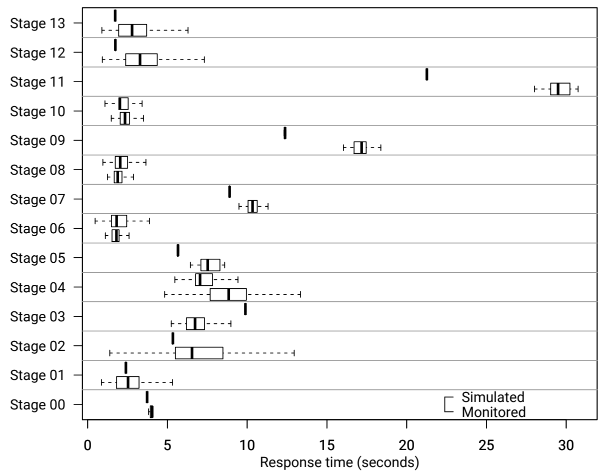
		Random Forest Classification Application				Linear Regression Application			
Worker Nodes	Data Workload	Monitored MCPU	Simulated MCPU	RMSE	Prediction Error	Monitored MCPU	Simulated MCPU	RMSE	Prediction Error
4	Small	48.96%	45.69%	3.31%	6.69%	48.86%	47.43%	2.53%	2.94%
	Large	56.93%	48.70%	8.23%	14.45%	57.55%	52.06%	5.62%	9.53%
	Huge	56.06%	49.66%	6.43%	11.42%	56.32%	55.45%	4.02%	1.54%
8	Small	35.23%	34.83%	0.91%	1,13%	36.03%	32.48%	3.72%	9.86%
	Large	44.64%	39.60%	5.31%	11.29%	46.13%	42.51%	3.85%	7.85%
	Huge	47.27%	40.66%	6.61%	13.98%	52.93%	48.15%	4.81%	9.04%
16	Small	22.65%	22.12%	0.84%	2.32%	22.05%	19.34%	2.91%	12.26%
	Large	31.23%	27.61%	3.65%	11.57%	31.85%	28.99%	3.06%	8.97%
	Huge	34.00%	30.72%	3.39%	9.63%	38.22%	35.59%	3.13%	6.89%

9.6.5 Evaluating Data Workload Changes

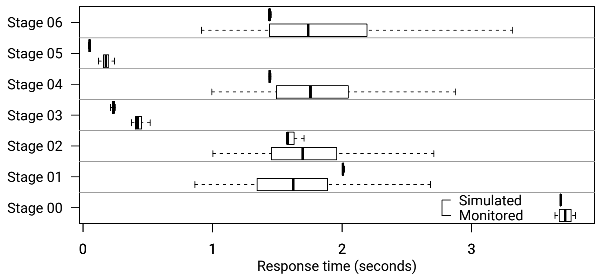
In order to evaluate our first claim that data workload changes can be modified independently, we specified three different scenarios *small*, *large*, and *huge* for both applications. Table 9.4 shows the corresponding number of files, file sizes, total partitions and total sizes for each scenario. The basis for evaluating workload changes of each application provides one initial PerTract-DSL instance each. We extracted this instance from a monitored experiment with a small data workload in a cluster of four worker nodes. Afterwards, we changed the Data Workload Architecture according to the scenarios large and huge and simulated the model instances. The simulation and monitoring results are part of Tables 9.5 and 9.6.

The starting experiment (i.e., four nodes and small workload) shows a response time prediction error of 0.78% for the RFC and 2.23% for the LR application. CPU prediction errors amount to 6.69% and 2.94%. Changing the data workload according to the large and huge scenarios leads to a response time prediction error of 7.90% and 7.73% for the RFC and 6.45% and 6.30% for the LR applications. Similar to the prediction errors, the RMSE increased in both scenarios. For the huge scenario, Figure 9.5 illustrates the response time statistics of simulated and monitored Spark tasks for each stage. For both applications, we predict the median of the tasks for 16 of 21 stages with errors below 30%.

However, the monitored results show an increased deviation compared to the simulation results, especially, for the LR application. This is due to the monitored delays and task processing, which showed great variances. While we model delays with probability distributions, we only use the mean for estimating CPU demands and did not depict this behavior. For the RFC application, tasks for stages *05*, *07*, *09*, and *11* also differ significantly. These stages contain reduce operations for which the input data size does not exactly scale linearly with increasing data workload for this RFC application. However, the error only has a minor effect on the overall application response time as stages for reduce tasks consist of only eight tasks compared to 360 tasks for each of the other stages.

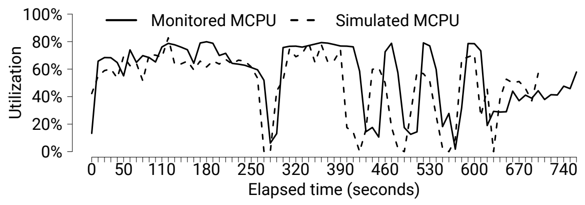


(a) *Random forest classification*

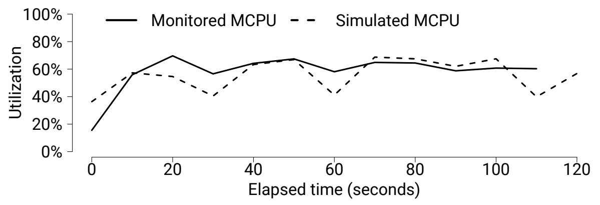


(b) *Linear regression*

Figure 9.5: *Response time statistics of Spark tasks for each stage (four worker nodes, huge data workload)*



(a) *Random forest classification*



(b) *Linear regression*

Figure 9.6: *Mean CPU utilization of four worker nodes (huge data workload)*

For the large and huge workload scenarios, the RMSE for CPU consistently remain below 9%. CPU prediction errors amount to 14.45% and 11.42% for the RFC and 9.53% and 1.54% for the LR application. Figure 9.6 illustrates the CPU utilization over time for one experiment run. In order to avoid illustrating too many lines, we calculated the mean across the worker nodes. Although underestimating the CPU utilization by 6.4% for the RFC application, the graphs of the simulated and monitored values map very closely.

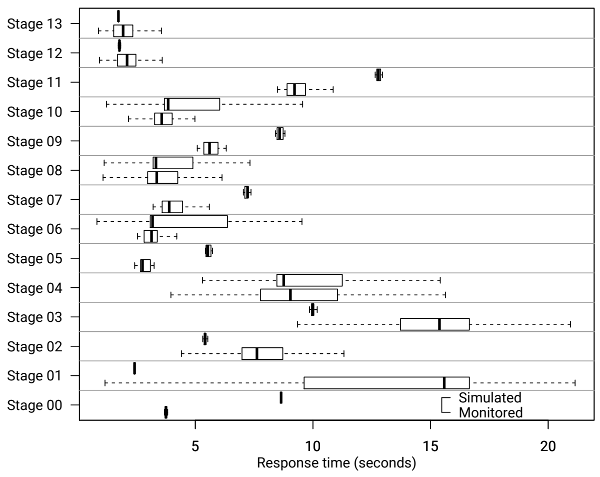
The results for response time and resource utilization show accurate prediction results based upon averages for upscaling workload changes. Therefore, we validated the claim of being able to change data workloads independent of Execution Architectures and Resource Architectures.

9.6.6 Evaluating Resource Changes

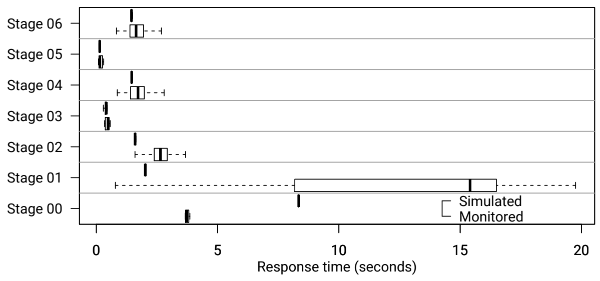
We increased the initial cluster size of four worker nodes by factors two and four in order to evaluate our second claim that hardware resources can be changed independently of Execution Architectures and Data Workload Architectures.

Similarly, the evaluation is based on one initial PerTract-DSL instance for each application, which is the same as for the data workload evaluation and was extracted from a monitored experiment with four worker nodes. Afterwards, we increased the worker nodes to eight and 16 nodes in the Resource Architecture. Additionally, we adapted the number of executors e in the application configuration of the Execution Architecture to match the number of worker nodes. The simulation and monitoring results are part of Tables 9.5 and 9.6.

In the previous subsection, we already discussed the same starting experiment, which does not include any changes. For eight worker and 16 worker nodes, response time prediction errors amount to 10.53% and 0.08% for the RFC application and 6.59% and 1.34% for the LR application, respectively. Compared to the data workload changes, the RMSE is lower throughout the resource changes for both applications. Figure 9.7 additionally shows the detailed response time statistics of Spark tasks for each stage of the applications. Compared to the data workload evaluation, the median values of simulated and monitored results lie closer together. The distance of the first and third quartiles are also predicted more accurately for most stages of both applications. For a few stages such as *Stage 01*, minimum, maximum, and quartiles differ significantly. Nonetheless, response time predictions errors on application-level remain below 15% in total.



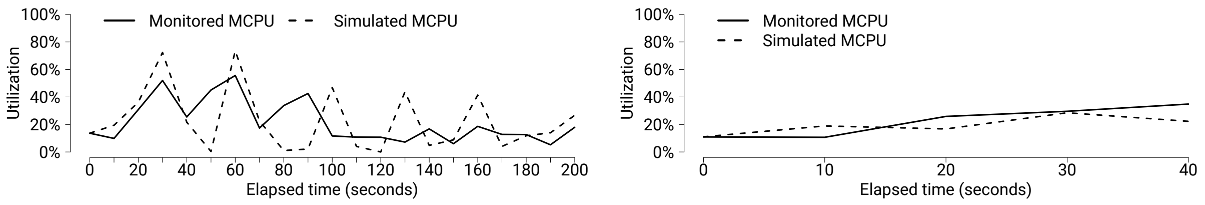
(a) *Random forest classification*



(b) *Linear regression*

Figure 9.7: *Response time statistics of Spark tasks for each stage (16 worker nodes, small data workload)*

For eight worker and 16 worker nodes, CPU prediction errors come to 1.13% and 2.32% for the RFC application and to 9.86% and 12.26% for the LR application, respectively. Figure 9.8 illustrates the CPU utilization over time for one experiment run. For the RFC application, the simulated CPU usage overestimates several peaks and underestimates



(a) *Random forest classification* **(b)** *Linear regression*

Figure 9.8: *Mean CPU utilization of 16 worker nodes (small data workload)*

negative peaks. However, it depicts the progression of the monitored results overall. For the LR application, the predicted CPU utilization is very precise.

In total, the simulation results show accurate prediction results for upscaling hardware resource changes with mean prediction errors below 15% and validate the claim that hardware resource can be modified without changing Execution Architectures and Data Workload Architectures.

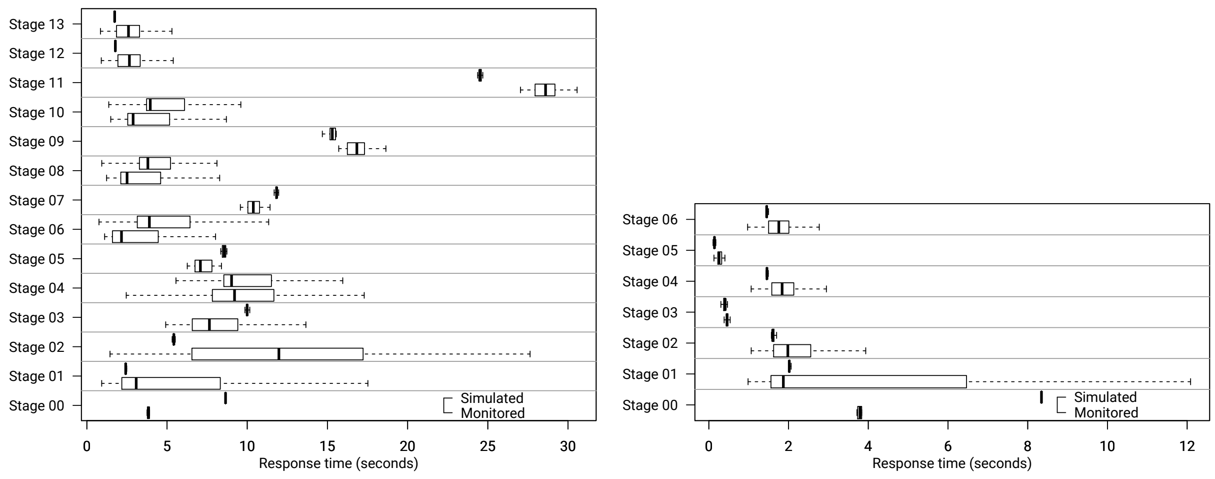
9.6.7 Evaluating Data Workload and Resource Changes

In order to evaluate our claim that data workload and hardware resources can be modified without changing application Execution Architectures, we applied both upscaling scenarios together, regarding data workload as well as worker nodes. The simulation and monitoring results are part of Tables 9.5 and 9.6. Again, the evaluation is based on the same initially extracted PerTract-DSL instance for each application.

For eight worker nodes and a large data workload, response time prediction errors amount to 14.74% for the RFC and 4.79% for the LR application. For huge data workload, the errors are 8.94% and 4.15%, respectively. For 16 worker nodes and a large data workload, response time prediction errors come to 0.76% for the RFC and 10.74% for the LR application. With huge data workload, the errors are 6.06% and 6.93%, respectively. The RMSE results consistently behave similarly to prediction errors. The highest RMSE amounts to 56.62 seconds, which equals 14.97% of the corresponding monitored response times. For all scenarios, prediction errors constantly remain below 15%. Figure 9.9 additionally shows the response time statistics of results with 16 worker nodes and huge workload. Compared to the two previous evaluations, the simulation results depict monitoring results as the closest for both applications.

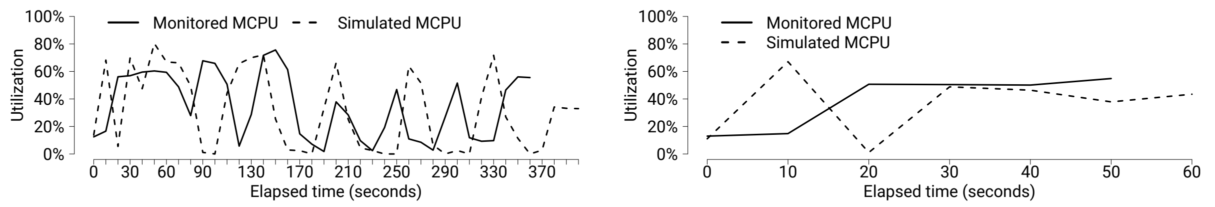
Looking at the CPU results for eight worker nodes and a large data workload, prediction errors amount to 11.29% for the RFC application and 7.85% for the LR application. For a huge workload, the errors remain similarly with 13.98% and 9.04%. For 16 worker nodes and a large data workload, the errors also remain 11.57% and 8.97%. With a huge data workload, they decrease a little to 9.63% and 6.89%, similar to the response time prediction.

Figure 9.10 shows the CPU utilization over time of one run with 16 worker nodes and a huge data workload. In case of the RFC application, the simulation graph depicts the



(a) *Random forest classification* **(b)** *Linear regression*

Figure 9.9: *Response time statistics of Spark tasks for each stage (16 worker nodes, huge data workload)*



(a) *Random forest classification* **(b)** *Linear regression*

Figure 9.10: *Mean CPU utilization of 16 worker nodes (huge data workload)*

progression of the monitored measurements. However, it shifts as the response time differs. In case of the LR application, the simulated CPU utilization is also slightly shifted due to the different response times. Otherwise, it depicts the monitored utilization except for one peak at the beginning. This is due to overestimating the CPU demand for *Stage 00*. Similarly, the task response time also significantly differs for *Stage 00* for both applications throughout all experiments. The reason for the overestimation is that this stage consists of only one task, which does not scale linearly with the dependent data size. This is a case that we intentionally did not consider and could not cover as it requires metaknowledge of the application that we do not expect in an automatic extraction process.

Overall, the simulated results for response times on an application-level as well as CPU utilization show accurate predictions for both data workload changes and hardware resources. The mean prediction errors remained below 15% as well as the RMSE compared to the monitored results. In performance evaluation literature, prediction errors of 30% across cluster sizes are expected (Ardagna et al., 2018). Therefore, we validated the claim of being able to change data workloads and resources' architectures independent of Execution Architectures. Our approach enriches related work by predicting CPU utilization across clusters and over time.

9.6.8 Threats to Validity

Although we applied some sophisticated machine learning applications, we generated data and used only a set of sample applications from one benchmark suite. As they are far more complex applications and have deviating data in praxis, this represents a threat to external validity (Wohlin et al., 2012).

Furthermore, we evaluated our approach only for one technology (i.e., Apache Spark) and one type of application (i.e., batch). In previous work, we showed that our approach is also applicable for Spark Streaming applications (Kroß/Krcmar, 2017). However, we claim that the DSL builds a foundation to specify other technologies as well, such as Apache Flink and Apache Storm. Extensions might be required (e.g., additional parameters) to support modeling and accurate predictions. We plan to evaluate this in our future work.

We used several visualizations and statistical measures such as mean, standard deviation, and relative error to ensure statistical conclusion validity. While the results of one measure can be close to each other (e.g., mean), another measure can differ significantly (e.g., minimum value).

9.6.9 Assumptions and Limitations

We allocated one Spark executor to each node during our experiments. It is also possible to size less cores and memory for Spark executors, which would enable Spark to allocate multiple executors to one node. Although we are also able to model and simulate these scenarios, we did not evaluate such a case. We evaluated our experiments in a virtualized, but exclusive cluster in which no other applications were running in parallel and using any CPU, disk drives, or networks. For data analytics applications, CPU is usually the bottleneck (Ousterhout et al., 2015). As HiBench and other industry benchmarks mainly consist of only compute-intensive applications, we did not evaluate our approach for a wider variety of applications.

Regarding our modeling approach, we specified the input of a subsequent Spark stage probabilistically depending on the output data of a previous stage. Therefore, our prediction error will increase, if the properties of the initial underlying data set change significantly (e.g., the number of distinct words in case of a word count application). Another limitation is that we only include network delays in our models and simulations, but did not simulate network throughput and bandwidth yet. The same applies to disk drives. In addition, we also did not consider rack awareness in our specification. Regarding big data features and PCM, Heinrich/Eichelberger/Schmid (2016) discuss current challenges and potential solutions, for instance, for modeling data structures and continuous data flows.

9.7 Conclusions and Future Work

Modeling and predicting the performance of big data applications are essential for planning capacities and evaluating configurations. Automatically deriving models, specifying applications tool-agnostic, and gaining insights into performance-relevant factors of system architectures and dependencies are complex challenges. We present PerTract, an approach to automatically extract model specifications and transform them to the model-based performance evaluation tool Palladio. A PerTract-DSL allows the specification of (i) application execution architectures including components, parametric dependencies, and resource demands, (ii) computing resources, and (iii) data workloads. It is specifically designed for big data systems, decreases the complexity compared to full performance models, and simplifies the changeability to users. We demonstrated the extraction of DSL instances by the example of Apache Spark applications, Apache YARN resources, and Apache HDFS data. This is the first white-box approach to present an automated way to integrate measurements and estimate resource demands to produce performance models that can be simulated. We used two machine learning applications of the HiBench benchmark suite in the evaluation and upscaled data sizes as well as cluster sizes in different scenarios. We are able to predict mean response times on application-level and CPU usage with accurate predictions errors below 15%.

In our future work, we plan to extract DSL instances from more technologies. We already provide a way to extract the execution architecture of Apache Flink applications, but need further investigations to estimate accurate resource demands. Additional technologies include Apache Mesos for modeling computing resources and Apache Kafka for characterizing data workload. We also plan to implement direct transformations from the DSL to a scalable event-oriented discrete-event simulation as we are reaching the limit for simulating continuous sources (data streams). Finally, we will extend the specification of continuous data sources to include load intensity profiles that model variations in arrival rates (Kistowski et al., 2017).

Part C

Chapter 10

Summary of Results and Discussion of Implications

In this chapter we first summarize the results for the embedded publications. Afterwards, we outline limitations of this dissertation as well as the contributions to research and practice. We conclude with an outlook of future research.

10.1 Summary of Results

This section describes the results of the embedded publications. We refer to the publication numbers as introduced in section 3.3.

Publication *P1* proposes the vision of a model-based performance evaluation approach that allows for predicting the performance and scalability of system-of-systems such as big data and IoT systems. Motivated by IoT use cases in which big data must be processed constantly, we identify the need to consider software performance, especially, at the beginning of developing new systems in order to continuously ensure that performance and scalability requirements can be met. We describe that performance models are well understood in the area of business applications. As these applications are mainly user-driven and often process data on a single hardware node, they do not consider data as a first class entity. As a result, these approaches are not suitable for modeling the data-driven applications as well as distributed and parallel applications. We describe our future research goal to combine and integrate modeling approaches from different areas including business applications and big data applications. Our contributions shall to enable engineers to detect bottlenecks, predict end-to-end response times, and plan required capacities for efficient IT operations.

Publication *P2* introduces an approach that uses PCM to model and simulate the performance of a batch processing application that is part of a lambda architecture in order to minimize the usage of parallel stream processing. We use this model and its prediction results as a decision making model during each processing iteration. We predict if the response time of the batch processing application with given data input and hardware resources is within a certain response time threshold. If this is the case, we determine to switch off the speed layer for one iteration, otherwise we decide to run the speed layer for the iteration in order to meet the response time requirements. Based on a smart energy use case we implemented a batch application that analyzes data from wind power facilities. In order to evaluate our approach, we model and simulate the batch application and examine the accuracy of the response time prediction results for different sizes of input data. In the evaluation we only used a single-node cluster for the batch application and also simulated the application on only one hardware node. The reason for this is that we where not able to model a distributed and parallel setup with PCM. As a result of this publication, we derived a practical and detailed understanding of what features are required and were missing the meta-model.

In publication *P3*, we address the missing features we experienced in *P2* and proposed meta-model extensions to PCM. In contrast to classical component-based software systems, big data frameworks and systems parallelize and distribute data processing in a computer cluster. First, we extend the meta-model to allow for invoking an operation multiple times in parallel. We introduce two parameters, one to specify the simultaneous number of executions (e.g., that may equal the number of partitions of a data stream) and one to specify the total number of executions (e.g., that may be the number of partitions of a data file) with parametric dependencies. Second, we extend the meta-model to allow for specifying resource clusters. This includes to specify the resource role of a resource (e.g., master and worker) and the scheduling policy (e.g., round robin) that is used to determine the distribution of operations on worker nodes. In order to make use of the proposed extensions, we started to extend PCM's simulation engine SimuCom to support the features. This extension is tested in publication *P4* and described as well as finished in publication *P5*.

Publication *P4* presents an approach to model and simulate stream applications on the basis of our meta-model extensions from *P3*. As an exemplary technology we used the Spark Streaming API of Apache Spark and chose as distinct count application of the HiBench benchmark suite. We demonstrate how we depict the concepts of the Apache Spark framework into a performance model and how we use delay and CPU measurements to derive parametric resource demands. We evaluate the approach for an upscaling scenario regarding the amount of cluster nodes. We derive an initial performance model based on two cluster nodes and adapt the model to four, six, and eight cluster nodes. Afterwards, we compare simulated response times with measured response times from corresponding execution runs in order to evaluate the prediction accuracy. During the four scenarios, the mean relative prediction error remained below 22%. During the evaluation of this paper, we already evaluated not only the modeling approach but also our extensions to the simulation engine. We were able to simulate around 450,000 events per second and simulation unit, respectively, and also experienced already the limitations with regard to the scalability of the simulation engine. Unfortunately, simulating more events was not feasible for us.

In publication *P5*, we enhanced our modeling approach of publication *P4* and evaluated it in more detail. We model and evaluate one batch application as well as one stream application. We completed the extension to the simulation engine and describe corresponding model to text transformations. In addition to comparing response times, we also compare the prediction accuracy of CPU utilization. The procedure from *P4* in which we used measurements form the Spark's API to derive CPU resource demands did not result in accurate CPU utilization predictions. The reason was that the calculation of provided measurements involved some abstractions that distorted the measurements. We decided not to change the source code of Spark as we wanted our approach to be reproducible and useable without any modifications. Therefore, we used profilers to monitor the CPU time of applications and used those measurements to extract parametric resource demands. Compared to *P4*, we also considered upscaling scenarios regarding input data sizes for both application types. In order to evaluate the prediction accuracy, we compared the mean response time as well as the mean CPU utilization of simulated results with measured results. For all upscaling scenarios, both metrics, and both application, the prediction error was at most 17.02%.

Publication *P6* presents our approach PerTract to automatically extract models of big data systems. Throughout our former publications we faced several limitations and drawbacks. The adaption of model parameters with regard to data information is complex and involves several limitations as PCM does not consider data as first class entity. We also reached the performance limit of the simulation engine for stream applications. To address these concerns, we introduce an own DSL and modeling formalism to consider only essential characteristics of big data frameworks. Based on this DSL, we provide automated transformations to create PCM model instances. By this abstraction and considering data as a first class entity, we are able to adapt model parameters in an easier way and may also provide transformations to other simulation frameworks in future that may be able to to handle more load.

Furthermore, it was not feasible to model complex big data applications (e.g., machine learning) manually in prior publications. In publication *P6*, we introduce an automatic approach to derive DSL instances. We use provided interfaces by big data frameworks to derive the software execution architecture, data models, and hardware resources. We also implemented a lightweight agent to sample stack traces and CPU times of applications as existing monitoring solutions caused a considerable amount of overhead in our experiments. We use these monitored measurements to interrelate them with components of the software execution architecture and derive parametric resource demands. We automated all our procedures including simulating and analyzing simulation results and provided tool support in order to enable engineers to use our approach without expert knowledge in software performance. We evaluated our approach similar to publication *P5*, but used a random forest classification and linear regression as exemplary batch applications. In addition, we also visualized the CPU utilization over time and provided the response times not only on application-level, but also for task operations. Our evaluation shows that we are able to accurately predict the performance throughout our conducted scenarios.

Table 10.1 summarizes the key results of the embedded publications.

No.	Title	Key Results
P1	Towards a Model-driven Performance Prediction Approach for Internet of Things Architectures	• Demonstration of the necessity to consider performance and scalability when starting the development of big data and IoT systems
P2	Stream Processing On Demand for Lambda Architectures	• Modeling and simulation of a Apache MapReduce application on a single-node cluster • Identification that existing performance modeling approaches on architecture-level are not sufficient for evaluating data-intensive architectures and distributed and parallel processing
P3	Modeling Big Data Systems by Extending the Palladio Component Model	• Extensions for PCM's meta-model to allow for modeling – distributed and parallel operations – clustered resources
P4	Modeling and Simulating Apache Spark Streaming Applications	• Modeling an Apache Spark application and simulating the response time for upscaling hardware resources
P5	Model-Based Performance Evaluation of Batch and Stream Applications for Big Data	• Extensions for PCM's simulation framework SimuCom • Evaluation of batch and stream processing applications • Prediction of response time and CPU utilization
P6	PerTract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop	• A formalism and a DSL to model big data applications • A lightweight Java agent to sample stack traces and CPU times from applications • Automatic extractions of DSL instances from big data frameworks on application-, data-, and hardware-level • Comprehensive evaluation against machine learning applications for response time

Table 10.1: *Key results of embedded publications*

10.2 Limitations

The presented approaches and evaluations come with several assumptions and limitations, which we describe in this section.

In our approaches, we assume that operations always produce the same amount of output data in dependence on their input data. As a result, we model the input data of a subsequent operation as a percentage of the input data of a prior operation. Depending on the underlying data set, this assumption may not be always true. For instance, the output of a distinct word count application may vary depending on the number of distinct words. As a result, our prediction error will decrease or increase in case the data properties change. For test environments, it is desirable to have test data that comprise the same characteristics as production data.

As Apache Spark applications are also usually mainly CPU bound (Ousterhout et al., 2015), we did not explicitly consider demands for disk drives as well as network, but implicitly modeled them within delay demands. Although we tried to extract demands for disk drives, we were not able to retrieve fine-granular measurements without adding sophisticated instrumentation to Apache HDFS. We also did not model demands for allocating and deallocating main memory as simulation engines are not able to simulate such features while taking into account important features such as garbage collection (Brunnert/Krcmar, 2017). Furthermore, we did not regard data locality in our modeling approach. Throughout our evaluations, operations always read data from local disks. In production environments, data may not be always locally available, but may be read via network. Finally, we could not take rack awareness into account due to the size and virtualization of our test environment. Rack awareness is a feature supported by Apache YARN to enhance the performance in very large clusters.

Throughout our experiments, we used an exclusive cluster in which no other applications were running and competing for resources. For the experiments that involved Apache Spark, we always allocated one Spark executor on each node. It is also possible to configure Spark and YARN so multiple Spark executors can be allocated on one node. In order to calculate the response time of a Spark batch application, we did not use the response time provided by the Spark API as it involves the time to request and allocate worker nodes. We considered only the pure processing time as response time.

Big data frameworks involve hundreds of different configuration parameters each. For Apache Spark, Apache MapReduce, Apache YARN, and Apache HDFS, we always used the default settings (if not explicitly described in our embedded publications). We did not evaluate our approach for other configurations.

Finally, simulating Apache Spark Streaming applications with our approach is limited by the scalability of PCM's simulation engine SimuCom. As Spark Streaming implements a mini-batch-model, our approach is not evaluated for stream frameworks with an operator-model (e.g., Apache Flink) yet.

10.3 Contribution to Research

This thesis contributes to research by introducing a formalism to describe performance characteristics of big data systems on architecture-level including execution components, resource demands, data models, and hardware resources. With the introduction of a DSL, we allow for describing big data systems independent of their technology in a tool-agnostic way. DSL instances can be used as an intermediate language and be transformed and integrated with other performance modeling approaches and simulation frameworks.

We provide a white-box approach and demonstrate different procedures how execution components and inter-component interactions from common big data frameworks can be formalized and transferred into DSL instances. By providing an prototype that uses measurement data to automatically extract performance models, this thesis combines model-based approaches, which are usually conducted during development, and measurement-based approaches, which are usually performed during operations, as proposed by Woodside/Franks/Petriu (2007) to manage software performance efficiently (Smith, 2002; Menascė, 2002; Brunnert et al., 2015).

Finally, we provide a simple procedure how stack traces and corresponding CPU times can be sampled and monitored from long running big data applications in a distributed setup. We demonstrate to correlate these distributed traces to single execution components and put CPU times into parametric resource demands.

10.4 Contribution to Practice

Balsamo et al. (2004) describe that automation is a key factor for efficient and effective practice of performance management. This can be achieved through available tooling and support of performance prediction across the software lifecycle. Similarly, Osman/Knottenbelt (2012) argue that performance models are often unexploited as there is not sufficient tool support available and models are complex to create and maintain.

This dissertation contributes to the scope, applicability, and usability of model-based performance evaluations by providing a prototype to create performance models and to analyze big data systems. The automated extraction process eliminates the effort to create models by hand. It supports performance engineers to plan capacities, answer sizing questions as well as to analyze a system's performance and scalability that usually cannot be realized in test systems.

By automating all procedures, depicting only relevant parameters in a DSL, and abstracting performance models from users, this thesis aims also to support software engineers that do not have any knowledge about performance models. As software continuously evolves, DSL instances can be extracted and tracked for each application release. In this way required capacities can be continuously planned.

10.5 Future Research

There are several opportunities to extend the approaches introduced in this dissertation. In this section, we describe different aspects that are worth focussing on and should be pursued in future.

Support for Additional Technologies

At the beginning of this thesis, we experimented with Apache MapReduce and then mainly focussed on Apache Spark and Spark Streaming. While Spark Streaming utilizes a mini-batch model, it would be interesting to use our approach with additional frameworks that use an operated-based model such as Apache Storm and Apache Flink. In general, the support of additional frameworks also on data level (e.g., Apache Kafka) and on resource management level (e.g., Apache Mesos) is desirable. As a result, it would also allow for comparing the performance of different frameworks for certain workloads and support engineers at selecting an appropriate framework.

Another field we approached in this thesis is profiling and sampling the performance by using Java agents. At the moment our extraction approach requires monitoring data from our agent. As they are many different agents available, making our approach adaptable to different vendors would benefit the applicability.

Modeling and Simulating Additional Configuration Parameters

Big data frameworks involve hundreds of different parameters on different level. It is also desirable to identify the main performance influencing parameters. If appropriate, they could be incorporated into our formalism and DSL and evaluated in different scenarios. Modeling demands for disk drives, main memory, and network are additional possibilities to increase the prediction accuracy of our approach.

Simulation Capabilities

As we reached the performance limit when simulating data streams, transformations to other simulation frameworks are required. Similarly, we modeled data sources in a very simple way. In order to mimic realistic workload, load intensity profiles should be considered to model load variations. In this context, our approach could also be extended to allow for extracting DSL instances during runtime for stream applications. This would enable, for instance, to predict when data will queue and additional resources will be required to ensure real-time processing.

DSL Visualizations

The DSL instances represent execution components, inter-component relations and resource demands for different frameworks in a unified, technology-independent way. Visualizing the DSL, for instance, in a web application, would enable engineers to have a comprehensive view of the execution architecture and illustrate performance-characteristics.

References

Alipourfard, O.; **Liu, H. H.**; **Chen, J.**; **Venkataraman, S.**; **Yum, M.**; **Zhang, M. (2017):** CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). Boston, MA, USENIX Association, 469–482.

Alrokayan, M.; **Vahid Dastjerdi, A.**; **Buyya, R. (2014):** SLA-Aware Provisioning and Scheduling of Cloud Resources for Big Data Analytics. In Proceedings of the 2014 IEEE International Conference on Cloud Computing in Emerging Markets. IEEE, 1–8.

Amazon Web Services (2015): Amazon Kinesis. ⟨URL: `http://aws.amazon.com/kinesis/`⟩ last accessed 2019-01-20.

Aniello, L.; **Baldoni, R.**; **Querzoni, L. (2013):** Adaptive Online Scheduling in Storm. In Proceedings of the 7th ACM International Conference on Distributed Event-based Systems. New York, NY, USA, ACM, ISBN 978–1–4503–1758–0, 207–218.

Apache Cassandra (2015): The Apache Cassandra project. ⟨URL: `http://cassandra.apache.org/`⟩ last accessed 2019-01-20.

Apache Flink (2017a): Documentation - Dataflow Programming Model. ⟨URL: `https://ci.apache.org/projects/flink/flink-docs-release-1.3/concepts/programming-model.html`⟩ last accessed 2019-01-20.

Apache Flink (2017b): Documentation - Distributed Runtime Model. ⟨URL: `https://ci.apache.org/projects/flink/flink-docs-release-1.3/concepts/runtime.html`⟩ last accessed 2019-01-20.

Apache Flink (2017c): Documentation - YARN Setup. ⟨URL: `https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/yarn_setup.html`⟩ last accessed 2019-01-20.

Apache Hadoop (2015): Welcome to Apache Hadoop! ⟨URL: `http://hadoop.apache.org/`⟩ last accessed 2019-01-20.

Apache Hadoop (2017a): HDFS Architecture. ⟨URL: `http://hadoop.apache.org/docs/r2.7.5/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`⟩ last accessed 2019-01-20.

Apache Hadoop (2017b): MapReduce Tutorial. ⟨URL: `https://hadoop.apache.org/docs/r2.7.1/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`⟩ last accessed 2019-01-20.

Apache Kafka (2015): A high-throughput distributed messaging system. ⟨URL: `http://kafka.apache.org/`⟩ last accessed 2019-01-20.

Apache Pig (2014): Welcomt to Apache Pig! ⟨URL: `https://pig.apache.org/`⟩ last accessed 2019-01-20.

Apache Samza (2015): Samza. ⟨URL: `http://samza.apache.org/`⟩ last accessed 2019-01-20.

Apache Spark (2015): Lightning-fast cluster computing. ⟨URL: `https://spark.apache.org/`⟩ last accessed 2019-01-20.

Apache Spark (2017): Running Spark on YARN. ⟨URL: `http://spark.apache.org/docs/2.0.0/running-on-yarn.html`⟩ last accessed 2019-01-20.

Apache Storm (2015): Storm, distributed and fault-tolerant realtime computation. ⟨URL: `http://storm.apache.org/`⟩ last accessed 2019-01-20.

Aravantinos, V.; **Voss, S.**; **Teufl, S.**; **Hölzl, F.**; **Schätz, B. (2015):** AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. In Proceedings of the 8th International Workshop on Model-based Architecting of Cyber-physical and Embedded Systems..

Ardagna, D.; **Barbierato, E.**; **Evangelinou, A.**; **Gianniti, E.**; **Gribaudo, M.**; **Pinto, T. B. M.**; **Guimarães, A.**; **Silva, A. P. C. da**; **Almeida, J. M. (2018):** Performance Prediction of Cloud-Based Big Data Applications. In Proceedings of ACM/SPEC International Conference on Performance Engineering. ACM, ICPE '18, 192–199.

Ardagna, D.; **Bernardi, S.**; **Gianniti, E.**; **Karimian Aliabadi, S.**; **Perez-Palacin, D.**; **Requeno, J. I.; Carretero, J.**; **Garcia-Blas, J.**; **Ko, R. K.**; **Mueller, P.**; **Nakano, K. eds. (2016):** Modeling Performance of Hadoop Applications: A Journey from Queueing Networks to Stochastic Well Formed Nets. Springer International Publishing, 599–613, ISBN 978–3–319–49583–5.

Arlitt, M.; **Marwah, M.**; **Bellala, G.**; **Shah, A.**; **Healey, J.**; **Vandiver, B. (2015):** IoTAbench: An Internet of Things Analytics Benchmark. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering., 133–144.

Atzori, L.; **Iera, A.**; **Morabito, G. (2010):** The Internet of things: A survey. *Computer networks*, Vol. 54, No. 15, 2787–2805.

Balsamo, S.; **Di Marco, A.**; **Inverardi, P.**; **Simeoni, M. (2004):** Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, Vol. 30, No. 5, 295–310.

Barbierato, E.; **Gribaudo, M.**; **Iacono, M. (2014):** Performance evaluation of NoSQL big-data applications using multi-formalism models. *Future Generation Computer Systems*, Vol. 37, 345–353.

Becker, S.; **Koziolek, H.**; **Reussner, R. (2009):** The Palladio component model for model-driven performance prediction. *The Journal of Systems and Software*, Vol. 82, No. 1, 3–22.

Brambilla, G.; **Picone, M.**; **Cirani, S.**; **Amoretti, M.**; **Zanichelli, F. (2014):** A Simulation Platform for Large-scale Internet of Things Scenarios in Urban Environments. In Proceedings of the International Conference on IoT in Urban Space., 50–55.

Brandl, R.; **Bichler, M.**; **Ströbel, M. (2007):** Cost accounting for shared IT infrastructures. *WIRTSCHAFTSINFORMATIK*, Vol. 49, No. 2, 83–94.

Brosig, F.; **Meier, P.**; **Becker, S.**; **Koziolek, A.**; **Koziolek, H.**; **Kounev, S. (2015):** Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-Based Architectures. *IEEE Transactions on Software Engineering*, Vol. 41, No. 2, 157–175.

Brunnert, A.; **Hoorn, A. van**; **Willnecker, F.**; **Danciu, A.**; **Hasselbring, W.**; **Heger, C.**; **Herbst, N.**; **Jamshidi, P.**; **Jung, R.**; **Kistowski, J. von**; **Koziolek, A.**; **Kroß, J.**; **Spinner, S.**; **Vögele, C.**; **Walter, J.**; **Wert, A. (2015):** Performance-oriented DevOps: A Research Agenda. SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC) (SPEC-RG-2015-01) – Technical Report.

Brunnert, A.; **Krcmar, H. (2017):** Continuous Performance Evaluation and Capacity Planning Using Resource Profiles for Enterprise Applications. *Journal of Systems and Software*, Vol. 123, 239–262.

Brunnert, A.; **Vögele, C.**; **Danciu, A.**; **Pfaff, M.**; **Mayer, M.**; **Krcmar, H. (2014):** Performance management work. *Business & Information Systems Engineering*, Vol. 6, No. 3, 177–179.

Carbone, P.; **Katsifodimos, A.**; **Ewen, S.**; **Markl, V.**; **Haridi, S.**; **Tzoumas, K. (2015):** Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* Vol. 36, No. 4.

Casado, R.; **Younas, M. (2015):** Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, Vol. 27, No. 8, 2078–2091.

Casale, G.; **Ardagna, D.**; **Artac, M.**; **Barbier, F.**; **Nitto, E. D.**; **Henry, A.**; **Iuhasz, G.**; **Joubert, C.**; **Merseguer, J.**; **Munteanu, V. I.**; **Pérez, J. F.**; **Petcu, D.**; **Rossi, M.**; **Sheridan, C.**; **Spais, I.**; **Vladušič, D. (2015):** DICE: Quality-driven Development of Data-intensive Cloud Applications. In Proceedings of the Seventh International Workshop on Modeling in Software Engineering. IEEE, 78–83.

Castiglione, A.; **Gribaudo, M.**; **Iacono, M.**; **Palmieri, F. (2014):** Modeling performances of concurrent big data applications. *Software: Practice and Experience.*

Chen, C. L. P.; **Zhang, C.-Y. (2014):** Data-intensive applications, challenges, techniques and technologies: a survey on big data. *Information Sciences*, Vol. 275, 314–347.

Chen, H.; **Chiang, R. H. L.**; **Storey, V. C. (2012):** Business Intelligence and Analytics: From Big Data to Big Impact. *MIS Quarterly*, Vol. 36, No. 4, 1165–1188.

Costantino, L.; **Buonaccorsi, N.**; **Cicconetti, C.**; **Mambrini, R. (2012):** Performance analysis of an LTE gateway for the IoT. In Proceedings of the International Symposium on a World of Wireless, Mobile and Multimedia Networks., 1–6.

Daud, M. A.; **Suhaili, W. S. H.; Phon-Amnuaisuk, S.**; **Au, T.-W.**; **Omar, S. eds. (2017):** Internet of Things (IoT) with CoAP and HTTP protocol: A study on which protocol suits IoT in terms of performance. Springer International Publishing, 165–174, ISBN 978–3–319–48517–1.

Dean, J.; **Ghemawat, S. (2008):** MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, Vol. 51, No. 1, 107–113.

Faulstich, S.; **Hahn, B.**; **Tavner, P. J. (2011):** Wind turbine downtime and its importance for offshore deployment. *Wind Energy*, Vol. 14, No. 3, 327–337.

Faulstich, S.; **Lyding, P.**; **Tavner, P. (2011):** Effects of wind speed on wind turbine availability. In Proceedings of the European Wind Energy Conference. EWEA.

Franks, G.; **Al-Omari, T.**; **Woodside, M.**; **Das, O.**; **Derisavi, S. (2009):** Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering*, Vol. 35, No. 2, 148–161.

Ginis, R.; **Strom, R. E. (2010):** Method for predicting performance of distributed stream processing systems., US Patent 7,818,417 ⟨URL: `https://www.google.com/patents/US7818417`⟩ last accessed 2019-01-20.

Gómez, A.; **Merseguer, J.**; **Di Nitto, E.**; **Tamburri, D. A. (2016):** Towards a UML Profile for Data Intensive Applications. In Proceedings of the 2Nd International Workshop on Quality-Aware DevOps. ACM, 18–23.

Guerriero, M.; **Tajfar, S.**; **Tamburri, D. A.**; **Di Nitto, E. (2016):** Towards a Model-driven Design Tool for Big Data Architectures. In Proceedings of the 2nd International Workshop on BIG Data Software Engineering. ACM, 37–43.

Hashem, I. A.; **Yaqoob, I.**; **Anuar, N. B.**; **Mokhtar, S.**; **Gani, A.**; **Ullah Khan, S. (2015):** The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, Vol. 47, 98–115.

Heinrich, R.; **Eichelberger, H.**; **Schmid, K. (2016):** Performance Modeling in the Age of Big Data - Some Reflections on Current Limitations. In Proceedings of the 3rd International Workshop on Interplay of Model-Driven and Component-Based Software Engineering., 37–38.

Herbst, N. R.; **Huber, N.**; **Kounev, S.**; **Amrehn, E. (2014):** Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and Computation: Practice and Experience*, Vol. 26, No. 12, 2053–2078.

Hesse, G.; **Lorenz, M. (2015):** Conceptual Survey on Data Stream Processing Systems. In Proceedings of the 21st IEEE International Conference on Parallel and Distributed Systems. IEEE, 797–802.

Hevner, A. R.; **March, S. T.**; **Park, J.**; **Ram, S. (2004):** Design Science in Information Systems Research. *MIS Quarterly*, Vol. 28, No. 1, 75–105.

Hindman, B.; **Konwinski, A.**; **Zaharia, M.**; **Ghodsi, A.**; **Joseph, A. D.**; **Katz, R. H.**; **Shenker, S.**; **Stoica, I. (2011):** Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In Proceedings the 8th USENIX Symposium on Networked Systems Design and Implementation. Boston, MA, USA, USENIX, 295–308.

Huang, S.; **Huang, J.**; **Dai, J.**; **Xie, T.**; **Huang, B. (2010):** The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In Proceedings of the 26th International Conference on Data Engineering Workshops. IEEE, 41–51.

Huber, N.; **Brosig, F.**; **Spinner, S.**; **Kounev, S.**; **Bähr, M. (2017):** Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language. *IEEE Transactions on Software Engineering*, Vol. 43, No. 5, 432–452.

Hunt, P.; **Konar, M.**; **Junqueira, F. P.**; **Reed, B. (2010):** ZooKeeper: Wait-free Coordination for Internet-scale Systems. In Proceedings of the 2010 USENIX Annual Technical Conference. USENIX, 145–158.

Jain, R. (1991): The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling. John Wiley & Sons, Wiley professional computing, ISBN 0471503363.

King, B. (2004): Performance assurance for IT systems. Boston, MA, USA, Auerbach Publications, ISBN 0849327784.

Kistowski, J. V.; **Herbst, N.**; **Kounev, S.**; **Groenda, H.**; **Stier, C.**; **Lehrig, S. (2017):** Modeling and Extracting Load Intensity Profiles. *ACM Transactions on Autonomous and Adaptive Systems*, Vol. 11, No. 4, 23:1–23:28.

Kistowski, J. von; **Herbst, N. R.**; **Kounev, S. (2014):** LIMBO: A Tool For Modeling Variable Load Intensities. In Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. New York, NY, USA, ACM, 225–226.

Kreps, J. (2014): Questioning the Lambda Architecture. ⟨URL: `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`⟩ last accessed 2019-01-20.

Kreps, J.; **Narkhedem, N.**; **Rao, J. (2011):** Kafka: a Distributed Messaging System for Log Processing. In Proceedings of the 6th International Workshop on Networking Meets Databases. ACM, 145–158.

Kroß, J.: PerTract. `https://github.com/johanneskross/pertract` (accessed on 7 August 2019).

Kroß, J.; **Brunnert, A.**; **Krcmar, H. (2015):** Modeling Big Data Systems by Extending the Palladio Component Model. *Softwaretechnik-Trends* Vol. 35, No. 3.

Kroß, J.; **Brunnert, A.**; **Prehofer, C.**; **Runkler, T. A.**; **Krcmar, H. (2015a):** Model-based performance evaluation of large-scale smart metering architectures. In Proceedings of the 4th International Workshop on Large-Scale Testing. New York, NY, USA, ACM, 9–12.

Kroß, J.; **Brunnert, A.**; **Prehofer, C.**; **Runkler, T. A.**; **Krcmar, H. (2015b):** Stream Processing on Demand for Lambda Architectures. In **Beltrán, M.**; **Knottenbelt, W.**; **Bradley, J. eds.:** Computer Performance Engineering. Vol. 9272, Springer International Publishing, 243–257.

Kroß, J.; **Krcmar, H. (2016):** Modeling and Simulating Apache Spark Streaming Applications. *Softwaretechnik-Trends* Vol. 36, No. 4.

Kroß, J.; **Krcmar, H. (2017):** Model-based Performance Evaluation of Batch and Stream Applications for Big Data. In IEEE 25th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. IEEE, MASCOTS, 80–86.

Lehrig, S. (2014): Applying Architectural Templates for Design-Time Scalability and Elasticity Analyses of SaaS Applications. In Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability. ACM, 2:1–2:8.

Levy, Y.; **Ellis, T. J. (2006):** A systems approach to conduct an effective literature review in support of information systems research. *Informing Science: International Journal of an Emerging Transdiscipline*, Vol. 9, No. 1, 181–212.

Lilja, D. J. (2005): Measuring computer performance: a practitioner's guide. Cambridge university press, ISBN 0521646707.

Liu, X.; **Iftikhar, N.**; **Xie, X. (2014):** Survey of Real-time Processing Systems for Big Data. In Proceedings of the 18th International Database Engineering & Applications Symposium. New York, NY, USA, ACM, 356–361.

March, S. T.; **Smith, G. F. (1995):** Design and natural science research on information technology. *Decision Support Systems*, Vol. 15, No. 4, 251–266.

Marcu, O.; **Costan, A.**; **Antoniu, G.**; **Pérez-Hernández, M. S. (2016):** Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks. In Proceedings of the IEEE International Conference on Cluster Computing. IEEE, 433–442.

Martínez-Prieto, M. A.; **Cuesta, C. E.**; **Arias, M.**; **Fernánde, J. D. (2015):** The Solid architecture for real-time management of big semantic data. *Future Generation Computer Systems*, Vol. 47, 62 – 79, Special Section: Advanced Architectures for the Future Generation of Software-Intensive Systems.

Marz, N.; **Warren, J. (2015):** Big data: principles and best practices of scalable real-time data systems. Manning Publications Co., ISBN 9781617290343.

Medvedev, A.; **Hassani, A.**; **Zaslavsky, A.**; **Jayaraman, P.**; **Indrawan-Santiago, M.**; **Delir Haghighi, P.**; **Ling, S.; Podnar Žarko, I.**; **Broering, A.**; **Soursos, S.**; **Serrano, M. eds. (2017):** Data ingestion and storage performance of IoT platforms: Study of OpenIoT. Springer International Publishing, 141–157, ISBN 978–3–319–56877–5.

Menascė, D. A. (2002): Load Testing, Benchmarking, And Application Performance Management For The Web. In Proceedings of the 2002 Computer Management Group Conference., 271–281.

Menascė, D. A.; **Almeida, V. A. F. (2002):** Capacity Planning for Web Services: Metrics, Models, and Methods. Upper Saddle River, New Jersey, USA, Prentice Hall, ISBN 0130659037.

Menascė, D. A.; **Almeida, V. A. F.**; **Dowdy, L. W. (2004):** Performance by Design: Computer Capacity Planning by Example. Upper Saddle River, New Jersey, USA, Prentice Hall, ISBN 0130906735.

Nabi, Z.; **Wagle, R.**; **Bouillet, E. (2014):** The best of two worlds: integrating IBM InfoSphere Streams with Apache YARN. In Proceedings of the 2014 IEEE International Conference on Big Data. IEEE, 47–51.

Niemann, R. (2016): Towards the Prediction of the Performance and Energy Efficiency of Distributed Data Management Systems. In Proceedings of ACM/SPEC International Conference on Performance Engineering. ACM, 23–28.

Osman, R.; **Knottenbelt, W. J. (2012):** Database system performance evaluation models: A survey. *Performance Evaluation*, Vol. 69, No. 10, 471–493.

Ousterhout, K.; **Rasti, R.**; **Ratnasamy, S.**; **Shenker, S.**; **Chun, B.-G. (2015):** Making Sense of Performance in Data Analytics Frameworks. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation. Oakland, CA, USENIX Association, 293–307.

Peffers, K.; **Tuunanen, T.**; **Rothenberger, M. A.**; **Chatterjee, S. (2007):** A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, Vol. 24, No. 3, 45–77.

Rabl, T.; **Gómez-Villamor, S.**; **Sadoghi, M.**; **Muntés-Mulero, V.**; **Jacobsen, H.-A.**; **Mankovskii, S. (2012):** Solving Big Data Challenges for Enterprise Application Performance Management. *Proceedings of the VLDB Endowment*, Vol. 5, No. 12, 1724–1735.

Reussner, R.; **Becker, S.**; **Burger, E.**; **Happe, J.**; **Hauck, M.**; **Koziolek, A.**; **Koziolek, H.**; **Krogmann, K.**; **Kuperberg, M. (2011):** The Palladio Component Model. Karlsruhe (14) – Technical Report.

Rychlý, M.; **Škoda, P.**; **Smrž, P. (2015):** Heterogeneity-aware scheduler for stream processing frameworks. *International Journal of Big Data Intelligence*, Vol. 2, No. 2, 70–80.

Schäfer, A. M.; **Zimmermann, H. G. (2006):** Recurrent neural networks are universal approximators. In **Kollias, S.**; **Stafylopatis, A.**; **Duch, W.**; **Oja, E. eds.:** Artificial Neural Networks - ICANN 2006. Vol. 4131, Springer Berlin Heidelberg, ISBN 978–3–540–38625–4, 632–640.

Schermann, M.; **Hemsen, H.**; **Buchmüller, C.**; **Bitter, T.**; **Krcmar, H.**; **Markl, V.**; **Hoeren, T. (2014):** Big Data - an interdisciplinary opportunity for information systems research. *Business & Information Systems Engineering*, Vol. 6, No. 5, 261–266.

Sequeira, H.; **Carreira, P.**; **Goldschmidt, T.**; **Vorst, P. (2014):** Energy Cloud: Real-Time Cloud-Native Energy Management System to Monitor and Analyze Energy Consumption in Multiple Industrial Sites. In Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. IEEE, 529–534.

Shukla, A.; **Chaturvedi, S.**; **Simmhan, Y. (2017):** RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms. – Technical Report.

Shvachko, K.; **Kuang, H.**; **Radia, S.**; **Chansler, R. (2010):** The Hadoop Distributed File System. In 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)., 1–10.

Simon, H. A. (1996): The sciences of the artificial. MIT press, ISBN 0262193744.

Singhal, R.; **Singh, P. (2018):** Performance Assurance Model for Applications on SPARK Platform. In **Nambiar, R.**; **Poess, M. eds.:** Performance Evaluation and Benchmarking for the Analytics Era. Cham, Springer International Publishing, Lecture Notes in Computer Science, 131–146.

Skala, K.; **Davidovic, D.**; **Afgan, E.**; **Sovic, I.**; **Sojat, Z. (2015):** Scalable Distributed Computing Hierarchy: Cloud, Fog and Dew Computing. *Open Journal of Cloud Computing*, Vol. 2, No. 1, 16–24.

Skerrett, I. (2016): Profile of an IoT Developer: Results of the IoT Developer Survey. ⟨URL: `https://ianskerrett.wordpress.com/2016/04/14/profile-of-an-iot-developer-results-of-the-iot-developer-survey/`⟩ last accessed 2019-01-20.

Smith, C. U.; Marciniak, J. J. ed. (2002): Software Performance Engineering. John Wiley & Sons, ISBN 9780471028956.

Smith, C. U.; Bernardo, M.; **Hillston, J. eds. (2007):** Introduction to Software Performance Engineering: Origins and Outstanding Problems. Berlin, Heidelberg, Springer Berlin Heidelberg, 395–428, ISBN 978–3–540–72522–0.

Spinner, S.; **Casale, G.**; **Zhu, X.**; **Kounev, S. (2014):** LibReDE: a library for resource demand estimation. In Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014). New York, NY, USA, ACM, 227–228.

Steinberg, D.; **Budinsky, F.**; **Paternostro, M.**; **Merks, E. (2009):** EMF: Eclipse Modeling Framework. 2. Edition. Addison-Wesley, ISBN 9780321331885.

Taylor, J. W. (2008): An evaluation of methods for very short-term load forecasting using minute-by-minute British data. *International Journal of Forecasting*, Vol. 24, No. 4, 645 – 658.

Toshniwal, A.; **Taneja, S.**; **Shukla, A.**; **Ramasamy, K.**; **Patel, J. M.**; **Kulkarni, S.**; **Jackson, J.**; **Gade, K.**; **Fu, M.**; **Donham, J.**; **Bhagat, N.**; **Mittal, S.**; **Ryaboy, D. (2014):** Storm@Twitter. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. New York, NY, USA, ACM, SIGMOD '14, 147–156.

Vavilapalli, V. K.; **Murthy, A. C.**; **Douglas, C.**; **Agarwal, S.**; **Konar, M.**; **Evans, R.**; **Graves, T.**; **Lowe, J.**; **Shah, H.**; **Seth, S.**; **Saha, B.**; **Curino, C.**; **O'Malley, O.**; **Radia, S.**; **Reed, B.**; **Baldeschwieler, E. (2013):** Apache Hadoop YARN: Yet Another Resource Negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing. New York, NY, USA, ACM, SOCC '13, 5:1–5:16.

Venkataraman, S.; **Yang, Z.**; **Franklin, M.**; **Recht, B.**; **Stoica, B. (2016):** Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation. Santa Clara, CA, USENIX Association, 363–378.

Verma, A.; **Cherkasova, L.**; **Campbell, R. H. (2011):** ARIA: automatic resource inference and allocation for mapreduce environments. In Proceedings of the 8th ACM International Conference on Autonomic Computing. New York, NY, USA, ACM, 235–244.

Verma, A.; **Cherkasova, L.**; **Campbell, R. H. (2014):** Profiling and evaluating hardware choices for MapReduce environments: An application-aware approach. *Performance Evaluation*, Vol. 79, 328 – 344.

Vianna, E.; **Comarela, G.**; **Pontes, T.**; **Almeida, J.**; **Almeida, V.**; **Wilkinson, K.**; **Kuno, H.**; **Dayal, U. (2013):** Analytical Performance Models for MapReduce Workloads. *International Journal of Parallel Programming*, Vol. 41, No. 4, 495–525.

Vögele, C.; **Hoorn, A. van**; **Schulz, E.**; **Hasselbring, W.**; **Krcmar, H. (2016):** WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems. *Software & Systems Modeling*,, 1–35.

Wang, J.; **Shi, X.**; **Alhussein, M.**; **Peng, L.**; **Hu, Y. (2016):** SPSIC: Semi-Physical Simulation for IoT Clouds. *Mobile Networks and Applications*, Vol. 21, No. 5, 856–864.

Wang, K.; **Khan, M. M. (2015):** Performance Prediction for Apache Spark Platform. In Proceedings of the 17th International Conference on High Performance Computing and Communications (HPCC). IEEE, 166–173.

Webster, J.; **Watson, R. T. (2002):** Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Quarterly*, Vol. 26, No. 2, xiii–xxiii.

White, T. (2015): Hadoop: The Definitive Guide. 4. Edition. O'Reilly Media, Inc., ISBN 9781491901632.

Witt, C.; **Bux, M.**; **Gusew, W.**; **Leser, U. (2019):** Predictive performance modeling for distributed batch processing using black box monitoring and machine learning. *Information Systems*, Vol. 82, 33 – 52, ISSN 0306–4379.

Wohlin, C.; **Runeson, P.**; **Höst, M.**; **Ohlsson, M. C.**; **Regnell, B.**; **Wesslén, A. (2012):** Experimentation in software engineering. Springer Berlin Heidelberg, ISBN 9783642290435.

Woodside, M.; **Franks, G.**; **Petriu, D. C. (2007):** The Future of Software Performance Engineering. In 2007 Future of Software Engineering. Washington, DC, USA, IEEE Computer Society, FOSE '07, 171–187.

Zaharia, M.; **Chowdhury, M.**; **Das, T.**; **Dave, A.**; **Ma, J.**; **McCauly, M.**; **J., F. M.**; **Shenker, S.**; **Stoica, I. (2012a):** Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation. San Jose, CA, USA, USENIX, 15–28.

Zaharia, M.; **Das, T.**; **Li, H.**; **Hunter, T.**; **Shenker, S.**; **Stoica, I. (2012b):** Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing. EECS Department, University of California, Berkeley (UCB/EECS-2012-259) – Technical Report.

Zaharia, M.; **Xin, R. S.**; **Wendell, P.**; **Das, T.**; **Armbrust, M.**; **Dave, A.**; **Meng, X.**; **Rosen, J.**; **Venkataraman, S.**; **Franklin, M. J.**; **Ghodsi, A.**; **Gonzalez, J.**; **Shenker, S.**; **Stoica, I. (2016):** Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, Vol. 59, No. 11, 56–65.

Zhang, Z.; **Cherkasova, L.**; **Loo, B. T. (2013a):** Benchmarking Approach for Designing a Mapreduce Performance Model. In Proceedings of the ACM/SPEC International Conference on Performance Engineering. New York, NY, USA, ACM Press, ICPE '13, 253–258.

Zhang, Z.; **Cherkasova, L.**; **Loo, B. T. (2013b):** Performance Modeling of MapReduce Jobs in Heterogeneous Cloud Environments. In Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing. Washington, DC, USA, IEEE, CLOUD '13, 839–846.

Zhang, Z.; **Cherkasova, L.**; **Loo, B. T. (2015):** Exploiting Cloud Heterogeneity to Optimize Performance and Cost of MapReduce Processing. *SIGMETRICS Perform. Eval. Rev.* Vol. 42, No. 4, 38–50.

Zoitl, A.; **Strasser, T.**; **Valentini, A. (2010):** Open source initiatives as basis for the establishment of new technologies in industrial automation: 4DIAC a case study. In Proceedings of the IEEE International Symposium on Industrial Electronics., 3817–3819.

Appendix: Published/Accepted Version of Included Publications

Open Access

Open Journal of Internet of Things (OJIOT)
Volume 3, Issue 1, 2017

http://www.ronpub.com/ojiot
ISSN 2364-7108

Towards a Model-driven Performance Prediction Approach for Internet of Things Architectures

Johannes Kroß [A], Sebastian Voss [A], Helmut Krcmar [B]

[A] fortiss GmbH, Model-based Systems Engineering, Guerickestr. 25, 80805 Munich, Germany, {kross, voss}@fortiss.org
[B] Technische Universität München, Chair for Information Systems, Boltzmannstr. 3, 85748 Garching, Germany, krcmar@in.tum.de

Abstract

Indisputable, security and interoperability play major concerns in Internet of Things (IoT) architectures and applications. In this paper, however, we emphasize the role and importance of performance and scalability as additional, crucial aspects in planning and building sustainable IoT solutions. IoT architectures are complicated system-of-systems that include different developer roles, development processes, organizational units, and a multilateral governance. Its performance is often neglected during development but becomes a major concern at the end of development and results in supplemental efforts, costs, and refactoring. It should not be relied on linearly scaling for such systems only by using up-to-date technologies that may promote such behavior. Furthermore, different security or interoperability choices also have a considerable impact on performance and may result in unforeseen trade-offs. Therefore, we propose and pursue the vision of a model-driven approach to predict and evaluate the performance of IoT architectures early in the system lifecylce in order to guarantee efficient and scalable systems reaching from sensors to business applications.

Type of Paper and Keywords

Visionary paper: *performance, model, simulation, prediction, evaluation, Internet of Things, IoT, architectures*

1 Introduction

Since several years Internet of Things (IoT) constitutes one of the main future topics for industries [3]. Information and communication technologies for small devices continuously become not only more affordable, but also more powerful regarding processing. This enables these devices to be connected to the Internet. Additionally, big data technologies emerged and enabled organizations to store huge amounts of data and to analyze incoming data streams with sophisticated algorithms in real-time [11]. This has facilitated the evolution of IoT and enables organizations to build solutions for a highly diverse range of use case scenarios in different domains. Therefore, IoT may be considered as an umbrella term for different disciplines that already have longer histories (e.g., industry automation) and, additionally, promotes the integration of these different disciplines, for instance, the automatic combination of

This paper is accepted at the *International Workshop on Very Large Internet of Things (VLIoT 2017)* in conjunction with the VLDB 2017 Conference in Munich, Germany. The proceedings of VLIoT@VLDB 2017 are published in the Open Journal of Internet of Things (OJIOT) as special issue.

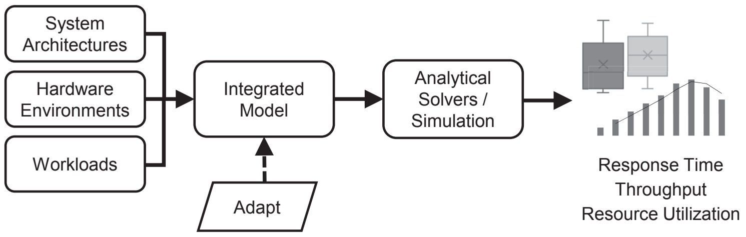


Figure 1: Model-based prediction approach

sensor data with enterprise resource planning data.

Although being promoted very much, only a few IoT use cases are implemented in industry yet. On the contrary, there are already hundreds of IoT platforms and technologies that are waiting to be exploited. In addition, there are several initiatives to define standards [3]; however, their establishment progresses slowly and an oversupply of vendor-specific implementations hamper the development of integrated solutions. For instance, an IoT developer survey with 528 participants conducted by the Eclipse IoT Working Group, IEEE IoT, and AGILE-IoT suggests that security, interoperability, and connectivity represent the three major concerns among all participants for developing IoT solutions [14]. However, for developers and organizations that have already deployed IoT solutions, performance becomes the third concern over connectivity. This reflects our comprehension that performance is not considered sufficiently when building architectures and finalized developments become very costly to counteract on late in the software life cycle.

We emphasize the role and importance of performance in terms of response time, throughput, and resource utilization. It is a vital aspect in planning and building sustainable IoT solutions as they involve multi-domain environments including constrained devices, gateways, and platforms of which the latter combines big data technologies and business applications. All these levels can have a direct impact on the performance of an overall system. Furthermore, evaluating the impact of design choices (e.g., regarding security, interoperability, and platforms) at development time is difficult, especially, for large-scale operations. These are only some of the factors that complicate IoT performance management.

In order to address and solve these issues, we propose the vision of a model-based approach for representing components and performance-influencing factors of IoT architectures and allow for *performance-by-design*. These models shall serve as input for analytical solvers or simulation engines and allow for predicting different performance metrics (Figure 1) [6]. In this way, architectures can proactively be evaluated regarding bottlenecks and scalability. Required resource demands can be planned and the throughput and response time behavior of subsystems can be estimated. The derived performance metrics and predictions shall also contribute to support communication and collaboration between developers (e.g., embedded developers and developers for business applications) and operations.

2 MODEL-DRIVEN PERFORMANCE PREDICTION

Our vision and its realization is driven by the following three research questions, which we use to explain our proposal and intentions:

1. What resource requirements and performance difficulties occur and are relevant on different levels of IoT architectures?
2. Which existing approaches and technologies can be used for implementing the integrated modeling concept?
3. How can existing meta-models and simulation approaches of different levels be integrated and combined?

In order to address the first research question the different levels and developer roles of architectures must be considered. Figure 2 shows a very basic IoT architecture that is reduced to the essential three layers. First, constrained devices and controllers represent the things in IoT. Second, gateways, routers, and smart devices enable fog computing at the edge and may integrate as well as pre-analyze data from devices [13]. Third, platforms process, store, and aggregate data from different sources and enable business applications to analyze and report data to end users. The connectivity and communication among the levels is not limited to one direction. In addition, non-functional requirements

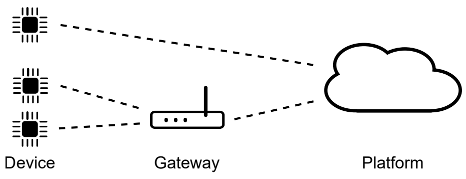


Figure 2: Abstract IoT architecture

such as performance, security, and interoperability are topics that influence all levels. For performance engineers, for instance, the following questions occur:

1. How shall computing resources (e.g., CPU, disk, memory, network) be sized on each level?
2. Shall gateways pre-analyze data and of how many devices per gateway?
3. What is the impact of protocol, security, and interoperability choices on the overall performance?
4. Does the architecture scale linearly with increasing number of devices and gateways?

Since the IoT stack is highly diverse, different developers and engineers are involved in the implementation. Embedded and systems developers are responsible on the device level and also partly on the gateway level. As gateways continuously expand, become smarter, and are able to run sophisticated operating systems, application developers also constitute a part on the gateway level. On the platform level, a mix of data scientists, application developers, and web developers implement the integration and visualization of data. Due to this mix of interests and engineering disciplines, we see the need to investigate the performance requirements on each level and for each role in order to understand influencing factors in a holistic view that need to be included in our model approach.

Similarly, previous and present related modeling approaches consider these disciplines in separated ways. As mentioned, IoT provides and increases the opportunity of combining existing approaches. Use case scenarios arise, for instance, that require capacity planning for devices and gateways based on formal models which are already well understood in the domain of business information systems. Since there is a tremendous number of modeling approaches, the second research question addresses reviewing existing methods and technologies for different levels with regards to our vision. In the following we list one example technology for each level.

For the device level, for instance, AutoFOCUS3[1] represents an integrated model-based tool for the development process of embedded systems [1]. It includes the activities for modeling requirements, software architectures, hardware platforms, and deployments as well as for generating code. The software architecture is built up by different software components that may be connected to each other to allow for interactions and may also be decomposed into multiple hierarchical subcomponents. The hardware architecture includes resources such as processors and memory that can be linked. It also involves platform architectures for execution and runtime environments such as operating systems or Java virtual machines. The integration and combination of these models enables developers to apply different analysis and synthesis methods such as testing, model checking, deployment, and automated scheduling [1].

The Eclipse Framework for Distributed Industrial Automation and Control (4diac)[2][3] is part of the Eclipse IoT ecosystem and represents an instance for modeling the gateway level. It provides an open source infrastructure for distributed industrial process measurement and control systems based on the IEC 61499 standard [17]. In order to model software architectures, 4diac includes an application editor that allows for representing function block networks consisting of one or multiple function blocks and their interaction via events. Similarly, a separate editor is included to model the specification of hardware by modeling devices and resources. By the means of several more editors and an own runtime environment, 4diac supports the development of industrial IoT applications and facilitates portability, interoperability, configurability, and scalability as promoted by IEC 61499 [17].

For the platform level, the performance management work tools (PMWT)[4] provide several integrated approaches to automate, support and integrate performance engineering activities across the software lifecycle [6]. This includes the automatic generation of models for enterprise applications based on performance measurements [5], modeling complex user behavior of applications [15], and simulating the performance of big data applications [9].

In order to address the third research question, we will examine similarities of model-based approaches for the different levels and domains of IoT architectures. For instance, models on architecture-level may often separate their meta-model as illustrated in Figure 1.

[1] http://af3.fortiss.org
[2] https://eclipse.org/4diac/
[3] http://fortiss.org/research/projects/4diac/
[4] https://pmw.fortiss.org/tools/pmwt/

One or several models are used to describe the software and system architecture, its components and its activity flow. Another model is used to describe the hardware and resource environment such as computing nodes with processors, disks, and memory connected via a network. An additional usage model is used to describe the use case scenarios of the software architecture and the workload.

Although implicitly considering performance aspects, present solutions focusing on the device and gateway level usually concentrate on guaranteeing functionality and safety [1]. In contrast, there are a lot of performance models to predict and analyze behavior on the platform level. Existing models on architecture-level that provide the benefits we seek with our vision, however, only focus on classical business applications and involve different requirements. For instance, the workload of business applications is mostly user-driven such as the number of parallel user accesses, whereas IoT applications are mostly data-driven such as the volume, velocity, and variety of incoming data. In addition, massive distributed and parallel computing and resource clusters are properties that are usually not found in business applications. Therefore, we aim at combining existing model approaches and additionally implementing missing functionalities so we are able to model the performance requirements we identified in the first research questions.

In order to be able to predict the performance behavior of the architectures, we will also implement simulators and solvers for deriving different metrics such as response time, throughput, and resource utilization. To evaluate our approach (Figure 1), we plan to model applications from IoT benchmark suites and adapt these models for various scenarios such as increasing number of things and increasing resource capacities. After simulating the models, we will compare simulation results with measurement results of applications from the benchmark that we adapted in the same way.

3 Related Work

As already mentioned, IoT involves and combines a variety of application domains and development stacks. Consequently, there are a lot of related, but also highly diverse approaches of which we try to refer to examples to the best of our knowledge in this Section. There are also several solutions available to model constrained devices, simulate networks within IoT architectures on a very detailed level, and evaluate them for throughput and latency issues. For instance, Wang et al. [16] apply the network simulator OPNET for IoT cloud solutions; Brambilla et al. [4] propose a simulation methodology to test large-scale IoT systems with interconnected devices in urban environments and include several network protocols and different mobility, network, and energy consumption models.

Furthermore, many related approaches specifically analyze and compare the performance of different protocols or technologies on different layers. For scenarios in which devices and gateways do not have a wired connection, for instance, Costantino et al. [7] investigate LTE as a suitable interconnection in terms of its efficiency, bandwidth, and coverage. In contrast, Daud and Suhaili [8] provide a performance evaluation of protocols for the application layer in IoT architectures. Therefore, they compare the hypertext transfer protocol (HTTP) and the constrained application protocol (CoAP) for message formatting, communication, and request handling on different test beds.

There are several developments of performance benchmarks for IoT, however, mostly on the platform level. Arlitt et al. present an analytics benchmark called IoTAbench [2]. It allows for generating, loading, repairing and analyzing synthetic data and was evaluated by the example of a smart metering use case and using a HP Vertica database. Shukla et al. [12] propose another benchmark for distributed stream processing platforms (i.e., Apache Storm) called RIoTBench. They provide different data workloads and generators as well as a set of 27 common IoT tasks for different domains. Furthermore, Medvedev et al. [10] provide an evaluation of different IoT platforms with regards to performance characteristics.

4 Conclusion and Future Work

This paper proposes and pursues the vision of an model-based approach for predicting and evaluating the performance of IoT architectures and systems. It shall support developers and engineers at examining design choices early in the system lifecycle, finding potential bottlenecks, planning and sizing required resources on different levels, and predicting response times from sensors to visual results. We will start our future research and work with combining and integrating modeling approaches for embedded systems with approaches for big data systems as well as for business information systems. Therefore, we are currently developing a first prototype for an integrated modeling environment.

References

[1] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, and B. Schätz, "AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems," in *Proceedings of the 8th International Workshop on Model-based Architecting of Cyber-physical and Embedded Systems*, 2015.

[2] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver, "IoTAbench: An Internet of Things Analytics Benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 133–144.

[3] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[4] G. Brambilla, M. Picone, S. Cirani, M. Amoretti, and F. Zanichelli, "A Simulation Platform for Large-scale Internet of Things Scenarios in Urban Environments," in *Proceedings of the International Conference on IoT in Urban Space*, 2014, pp. 50–55.

[5] A. Brunnert and H. Krcmar, "Continuous Performance Evaluation and Capacity Planning Using Resource Profiles for Enterprise Applications," *Journal of Systems and Software*, vol. 123, pp. 239–262, 2017.

[6] A. Brunnert, C. Vögele, A. Danciu, M. Pfaff, M. Mayer, and H. Krcmar, "Performance Management Work," *Business & Information Systems Engineering*, vol. 6, no. 3, pp. 177–179, 2014.

[7] L. Costantino, N. Buonaccorsi, C. Cicconetti, and R. Mambrini, "Performance Analysis of an LTE Gateway for the IoT," in *Proceedings of the International Symposium on a World of Wireless, Mobile and Multimedia Networks*, June 2012, pp. 1–6.

[8] M. A. Daud and W. S. H. Suhaili, "Internet of Things (IoT) with CoAP and HTTP Protocol: A Study on Which Protocol Suits IoT in Terms of Performance," in *Proceedings of the Computational Intelligence in Information Systems Conference*, S. Phon-Amnuaisuk, T.-W. Au, and S. Omar, Eds., 2017, pp. 165–174.

[9] J. Kroß and H. Krcmar, "Modeling and Simulating Apache Spark Streaming Applications," *Softwaretechnik-Trends*, vol. 36, no. 4, 2016.

[10] A. Medvedev, A. Hassani, A. Zaslavsky, P. Jayaraman, M. Indrawan-Santiago, P. Delir Haghighi, and S. Ling, "Data Ingestion and Storage Performance of IoT Platforms: Study of OpenIoT," in *Preceedings of Second International Workshop on Interoperability and Open-Source Solutions for the Internet of Things, Stuttgart, Germany, November 7*, I. Podnar Žarko, A. Broering, S. Soursos, and M. Serrano, Eds., 2017, pp. 141–157.

[11] M. Schermann, H. Hemsen, C. Buchmüller, T. Bitter, H. Krcmar, V. Markl, and T. Hoeren, "Big Data - An Interdisciplinary Opportunity for Information Systems Research," *Business & Information Systems Engineering*, vol. 6, no. 5, pp. 261–266, 2014.

[12] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: A real-time IoT benchmark for distributed stream processing platforms," Tech. Rep., 2017. [Online]. Available: http://arxiv.org/abs/1701.08530

[13] K. Skala, D. Davidovic, E. Afgan, I. Sovic, and Z. Sojat, "Scalable Distributed Computing Hierarchy: Cloud, Fog and Dew Computing," *Open Journal of Cloud Computing*, vol. 2, no. 1, pp. 16–24, 2015. [Online]. Available: https://www.ronpub.com/ojcc/OJCC_2015v2i1n03_Skala.html

[14] I. Skerrett, "Profile of an IoT Developer: Results of the IoT Developer Survey," 2016. [Online]. Available: https://ianskerrett.wordpress.com/2016/04/14/profile-of-an-iot-developer-results-of-the-iot-developer-survey/

[15] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar, "WESSBAS: Extraction of Probabilistic Workload Specifications for Load Testing and Performance Prediction — A Model-Driven Approach for Session-based Application Systems," *Software & Systems Modeling*, pp. 1–35, 2016.

[16] J. Wang, X. Shi, M. Alhussein, L. Peng, and Y. Hu, "SPSIC: Semi-Physical Simulation for IoT Clouds," *Mobile Networks and Applications*, vol. 21, no. 5, pp. 856–864, 2016.

[17] A. Zoitl, T. Strasser, and A. Valentini, "Open Source Initiatives as Basis for the Establishment of New Technologies in Industrial automation: 4DIAC a Case Study," in *Proceedings of the IEEE International Symposium on Industrial Electronics*, 2010, pp. 3817–3819.

AUTHOR BIOGRAPHIES

Johannes Kroß received a B.Sc. degree in Business Information Systems from Middlesex University in London and a M.Sc. degree in Information Systems from Technical University of Munich. At the fortiss institute he is responsible for the Performance Management Group within the Model-based Systems Engineering competence field. The group focuses on all aspects required to ensure that given performance requirements for application systems can be met.

Sebastian Voss has done his Ph.D. in the avionic context at EADS Innovation Works in the department Sensors, Electronics & Systems Integration. Previously, he worked one year for Daimler research & development. At fortiss he is heading the Model-based Systems Engineering competence field. His research interests include techniques and tools for the professional development of software-intensive systems, design space exploration methods and model-based (tool) development in AutoFOCUS3.

Helmut Krcmar studied business management in Saarbrücken and obtained his doctorate in 1983. Afterwards, he worked as a postdoctoral fellow at the IBM Los Angeles Scientific Center and as assistant professor of information systems at the New York University and the City University of New York. From 1987 to 2002, he held the Chair for Information Systems at the University of Hohenheim, Stuttgart. Since 2002 he holds the Chair for Information Systems at the Technical University of Munich (TUM). From 2010 to 2013, he served as Dean of the Faculty of Computer Science.

Stream Processing On Demand for Lambda Architectures

Johannes Kroß[1], Andreas Brunnert[1], Christian Prehofer[1], Thomas A. Runkler[2], and Helmut Krcmar[3]

[1] fortiss GmbH, Guerickestr. 25, 80805 Munich, Germany
{kross,brunnert,prehofer}@fortiss.org
[2] Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, 81739 Munich, Germany
thomas.runkler@siemens.com
[3] Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
krcmar@in.tum.de

Abstract. Growing amounts of data and the demand to process them within time constraints have led to the development of big data systems. A generic principle to design such systems that allows for low latency results is called the lambda architecture. It defines that data is analyzed twice by combining batch and stream processing techniques in order to provide a real time view. This redundant processing of data makes this architecture very expensive. In cases where process results are not continuously required to be low latency or time constraints lie within several minutes, a clear decision whether both processing layers are inevitable is not possible yet. Therefore, we propose stream processing on demand within the lambda architecture in order to efficiently use resources and reduce hardware investments. We use performance models as an analytical decision-making solution to predict response times of batch processes and to decide when to additionally deploy stream processes. By the example of a smart energy use case we implement and evaluate the accuracy of our proposed solution.

Keywords: Lambda Architecture, Big Data, Performance, Model, Evaluation

1 Introduction

With the increasing ubiquity of information and communication technology (ICT) and the emergence of the Internet of things (IoT) the available data amount is growing exponentially. Simultaneously, technologies have been developed to store, manage and analyze these diverse and high volumes of data, also known as *big data* [30]. These circumstances allow for applying analytics in order to gain knowledge and support decision-making. For more and more usage scenarios, these analytical capabilities must also meet specific time requirements such as real-time [17]. One common approach to design big data systems that can cover many use cases is the lambda architecture [26]. It mainly consists of a batch layer and a speed layer. The former iteratively processes a set of historical data in batches while the latter processes the arriving data stream in parallel to incrementally analyze latest data. By joining the output of both layers query results always reflect current data.

Nowadays, various complementary technologies with different characteristics exist to build a big data system and there is hardly one technology solution that fits most use cases of an organization. Although the lambda architecture simply is a generic design framework which offers a solution for many use cases, nonetheless, a variety of technologies can be applied for the batch or speed layer. Examples for the batch layer are Hadoop MapReduce [5], Apache Pig [7], and Apache Spark [9] and for the speed layer Apache Storm [10], Apache Spark Streaming [9], Apache Samza [8], or Amazon Kinesis [2]. This multitude leads to the development of complex system of systems, which often results in performance issues and high resource requirements [14]. Furthermore, the lambda architecture intends to process all data twice in both layers. Batch processes also analyze data from the ground up in each iteration to ensure fault tolerance in case of hardware failures or human mistakes [26]. These fundamental ideas require costly resources. For use cases where time constraints are not continuously needed or lie between several minutes, it can be often an important question whether a speed layer is really required or not. However, this question can usually not be answered during system development nor in test systems under realistic workload. As stream processing heavily utilizes main memory, the speed layer can also become an expensive investment [24].

Therefore, we propose a speed layer or stream processing, respectively, on demand. The idea is to exclusively use batch processes as often as possible and switch on stream processing only when batch processes are likely to exceed response time constraints. In this way, computing power is utilized more efficiently and resources can be saved as well as be available for other processes. In case of virtualized environments, investments can be directly decreased by reducing cloud service resources. In order to switch on stream processing at the right time, it is inevitable to predict the response time of succeeding batch iterations. For this purpose, we use performance models. They allow to describe performance influencing factors of software systems and to predict performance metrics such as response time, throughput and utilization by means of analytical solvers or simulation engines [13]. Therefore, we integrate estimated resource demands into the model based on measurements from batch processes to simulate an accurate system behavior. This enables us to efficiently schedule stream processes.

In this paper, we first give a detailed description of our proposed approach in Section 2 and how we use performance models to support decision-making. In Section 3, we validate our approach in an experiment. We describe the selected use case, the setup and sample algorithm for the batch layer, and the prototype performance model to predict batch processes. Afterwards, we discuss the experimental results we derived for different workload scenarios. In Section 4, we reflect related work in the area of the lambda architecture and, finally, conclude our paper with providing an outlook for future work in Section 5.

2 Stream Processing On Demand

In order to make decisions about when to switch on stream processing, we use performance models as an analytical solution. As illustrated in Figure 1, the iterative process is divided into two main steps in which the following Sections 2.1 and 2.2 are structured. First, one batch iteration and, potentially, a concurrent stream process are started within the lambda architecture. Second, after the batch process has ended, a decision-making model is used to decide whether stream processing is required in the next batch process iteration or not. Basis of decision-making is a performance model which is used to predict the response time of a batch process. Afterwards, the procedure is repeated.

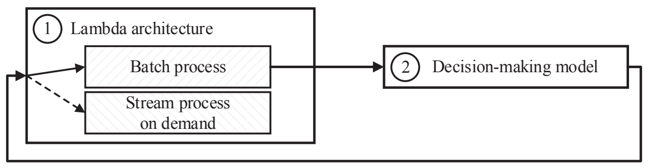


Fig. 1: Stream Processing On Demand Process

2.1 Data Processing in the Lambda Architecture

As already mentioned, our focus is on data processing, namely batch and stream processing, within lambda architecture and not storing data sets or results. Figure 2 illustrates the data flow and structure of batch and speed layer that differ from each other. Starting point is a shared data source which either streams the same data into each processing layer or gets accessed by each layer to retrieve data. Within the batch layer, all data are stored in a data set. A special characteristic of the data set is that it is append-only and data are not updated or removed [26]. Batch processes use the data set to operate on. In doing so, each batch process usually analyzes a huge set of historical data which leads to response times of minutes or hours for one batch job. The results are written to separate views, which is also considered as serving layer by Marz and Warren [26] for batch results. Batch processes constantly run iteratively and start from the beginning once a batch job has finished. If a batch process starts, only data that have been created before are included. Consequently, data that arrive during the current batch process are only included in the next new batch process. Since all data are analyzed in each cycle, each new result view can replace its predecessor. As the batch layer does not rely on incremental processing, it has the advantage of being a robust system where everything can be recomputed and reconstructed in case of hardware or software failures or human mistakes [26].

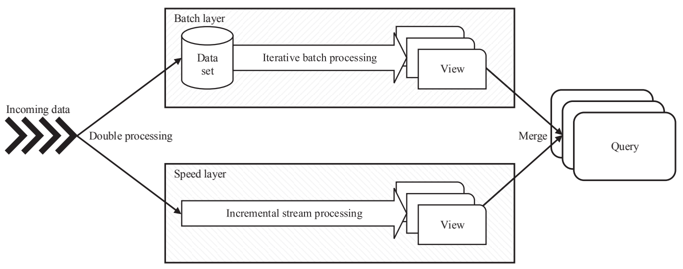


Fig. 2: Composition and data flow of batch and speed layer of the lambda architecture (adapted from Marz and Warren [26])

In contrast to the batch layer, the speed layer does not keep a record of historical data and solely uses main memory. As of today, stream processes run permanently and analyze each incoming message. They incrementally calculate and immediately update their result views. Thus, both layers include separated views and, in practice, usually different technologies are used as underlying databases because of their distinct requirements regarding read and write operations. In order to receive a holistic result, the view of both layers have to be merged in a query.

Although both layers process the same data, the results of queries that merge views only reflect data that are processed once at the time of the query. The purpose of the speed layer is to analyze the data prior to the batch layer and enable low latency by incremental updated result views. As a result, a past view of the speed layer can be discarded as soon as a subsequent batch job has finished.

A typical implementation of the lambda architecture as illustrated in Figure 2 would be to use Apache Kafka [6] - a publish-subscribe messaging system - as shared source for incoming data. For the batch layer, HDFS can be used as data set and Hadoop MapReduce for batch processing. For storing batch results, which Marz and Warren [26] also describe as serving layer, ElephantDB[4] represents a specialized database for this purpose. For the speed layer Apache Storm [10] is an example of an appropriate technology and Apache Cassandra [4] of a database.

[4] https://github.com/nathanmarz/elephantdb

2.2 Decision-making Model

To decide when to switch on stream processing, we predict the response time of succeeding batch processes and build a decision-making model. To comprehend why it is necessary to predict the succeeding batch processes, the chronological sequence of batch and stream processes as intended by the lambda architecture is illustrated in Figure 3. As already mentioned, results of batch processes are not available until they finish, while results of stream processes are incremental and can be queried at any time. Supposing one *batch process i* has ended and a decision must be made at time y on whether additional stream processes are needed afterwards or not, the earliest point in time where results of stream processes can be reasonably used is at time z. *Stream process j* considers only data newer than time y. Therefore, a batch process is required that has analyzed data before time y. However, the corresponding *batch process j* will only start after time y and end at a given time z. Thus, a decision must already be made at time y, if *batch process k* violates time-constraints so stream processes are switched on at time y. Consequently, query results after time z will have consistently incorporated all data.

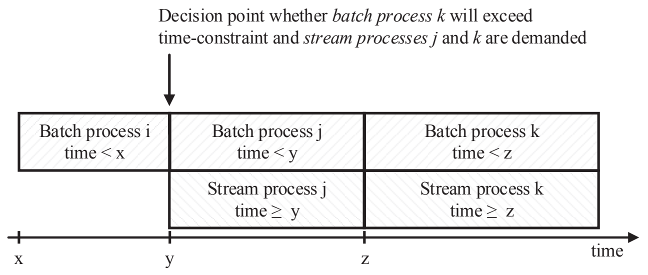

Decision point whether batch process k will exceed time-constraint and stream processes j and k are demanded
Batch process i time < x
Batch process j time < y
Batch process k time < z
Stream process j time ≥ y
Stream process k time ≥ z
x
y
z
time

Fig. 3: Chronological sequence of batch and stream processes

The above mentioned response time prediction is part of our decision-making model. Its procedure is depicted in Figure 4. Starting point is a finished batch process iteration. The response time of the second next batch iteration is predicted by using a performance model, which takes two inputs - the time constraint for the duration of a batch process and the load intensity. The latter means information about the incoming data of the batch layer. For instance, this can be in the form of a variable distribution as modeled by the LIMBO tool [22]. The prediction can be accomplished by means of simulation or analytical solving. If the predicted response time does not lie within the specified time limitation, the model tries to start batch processing in parallel with stream processing, otherwise the model considers batch processing only as sufficient.

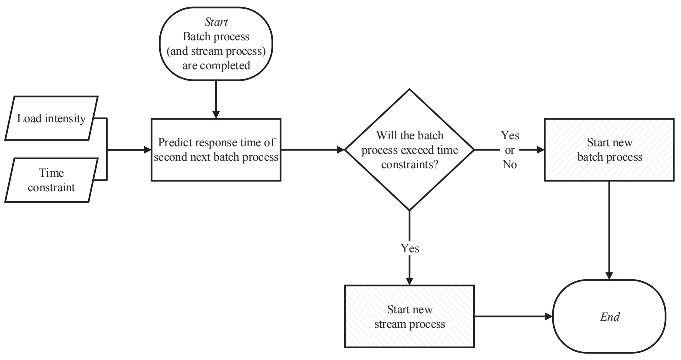


Fig. 4: Decision-making model

3 Experimental Validation

For the evaluation of our proposed approach, we conduct a controlled experiment which is described in the following Subsections. First, we discuss the selected use case. Second, we list the used setup and technologies of our exemplary batch layer as well as the sample algorithm for data processing. Afterwards, the performance model prototype to support decision-making is presented. Finally, we evaluate the accuracy of the inferred decision-making on the basis of three selected scenarios and discuss results from our observed measures.

3.1 Use Case and Design Options

To represent incoming data and their distribution, we pick the example of a common smart energy use case as illustrated in Figure 5.

Here, several hundred wind turbines are positioned in several wind farms in different geographic locations with long distances onshore or offshore. In order to operate efficiently, they measure several thousand parameters per turbine such as pressure, temperature or vibrations of rotor blades. As they are subject to various influences, wind turbines are not always in operation and do not measure data, for instance, if they are defect or are maintained. While onshore wind

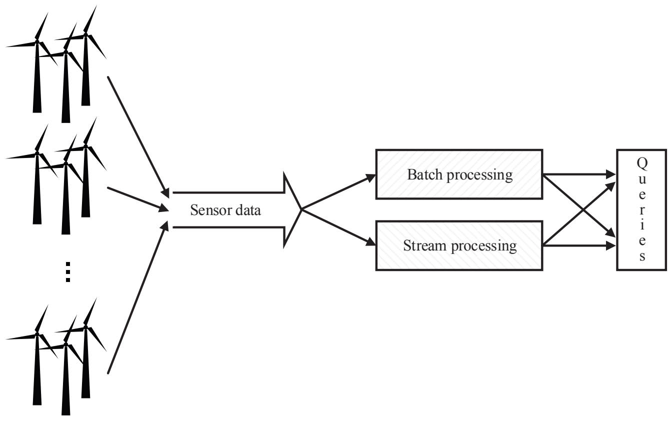


Fig. 5: Data processing of wind power facilities

turbines and wind farms, respectively, tend to have a time-based availability between 95-99%, the values for offshore wind farms with distance less than 12km range from 67.4% to 90.4% [19]. However, wind turbines include also downtimes, if wind is too strong or too weak which is described by the metric energy-based availability. Faulstich et al. [20] compared time-based and energy-based availability of wind turbines. In an extreme case where the downtime due to defects and the downtime due to wind speed does not overlap, the energy-based availability lies within 90.4-95,2%.

Dependent on a wind turbine's availability, we assume it either produces a set of measurement data with constant volume or does not produce any output data. As a result, wind turbines generate not only immense amount of heterogeneous data, but also variable load which makes it difficult to predict the production rate of data. As soon as data are generated, they flow into a central data center where they are processed. Dependent on the use case, data are handled in different ways. They can be gathered and stored in a central repository where batch processing can be used to extract, transform, and load (ETL) data and to apply complex analytics. This procedure usually lies in the range of minutes or hours and is not suitable for real-time requirements. For this purpose, stream processing can be used to directly process data as they stream in. Here, analytical algorithms may be designed in a simpler and less complex way than at batch processing as well as implemented in slightly different way as they produce incremental results.

In scenarios where low latency results are required and normally stream processing is chosen, but also analysis of historical data by batch processing need to be incorporated for conclusive results, the lambda architecture is an appropriate solution that allows for serving such use cases. Therefore, on both processing layers, stream and batch, the same kind of algorithm is implemented and results are joined.

Sensor data can be used for a variety of analytical scenarios such as for condition monitoring, diagnostics, predictive analytics or maintenance, and load forecasting. For our experiment, we concentrate on the latter example. Since the introduction of energy exchange such as the continuous intraday spot market of the European power exchange (EPEX), power can be bargained in 15-minute intervals up to 45 minutes before delivery which enables providers as well as consumers to efficiently act on short notice. In this case, the time-constraint is within 15 minutes. Typical forecast methods for short-term load forecasting include different exponential smoothing methods such as an autoregressive integrated moving average (ARIMA) model [33] or recurrent neural networks [29]. Furthermore, these algorithms are often applied on a sliding window of historical data.

Therefore, we will use this smart energy scenario as an example for our proposed approach and generate sensor data that are processed by one central system in similarly way as we have modeled it in a previous work [23]. The generator produces comma-separated values (CSV) files that represent measurements from wind turbines of one wind farm. Listing 1 shows the file structure and syntax.

Listing 1: Example of generated monitoring data from wind turbines

```
id,   timestamp,                 power,   param1, ...  paramN
12,   2015-04-01 08:23:04.125,   12.67,   value1, ...  value1
15,   2015-04-01 08:23:03.973,   13.49,   value2, ...  value2
13,   2015-04-01 08:23:04.096,   12.59,   value3, ...  value3
...
```

Each line represents a measurement of one wind turbine consisting of a *id*, *timestamp*, a *power* value and several hundred more parameters which we generated randomly and do not include in our succeeding analytic algorithms.

3.2 Implementation of the Batch Layer

To examine the accuracy of response time prediction for batch processes, we setup the batch layer using HDFS to store data sets and Hadoop MapReduce for batch processing. For simplicity, we installed a single node cluster in pseudo-distributed mode so Apache Hadoop runs only on one machine, but their daemons have their own Java processes. In order to do load forecasting and apply the data generator as mentioned in Section 3.1, we implemented a simple moving average algorithm in a Hadoop MapReduce job. It is based on an example algorithm[5].

[5] https://github.com/jpatanooga/Caduceus/

The MapReduce programming model intends to implement one map and one reduce function. The former takes a key/value pair as input and produces a set of key/value pairs, whereas the latter takes a key and set of associated values and combines the values to another smaller set [18]. In our case the map function is implemented as

Listing 2: Map function pseudo code

```
map(Object key1, String value1):
   // key1:   file name
   // value1: measurements of wind turbines of one farm
   for each line l in value:
      kv = parse(l)
      emit({kv.id, kv.timestamp}, {kv.timestamp, kv.power})
```

The function is called for each file within a given folder. It receives one CSV file and its value, which are multiple rows of measurement data of wind turbines. The algorithm reads every line and parses it in order to filter the *id* of a wind turbine, the *timestamp* of the measurement and the *power* value that describes the generated power to that time. Afterwards it releases a composite key containing the *id* and *timestamp*, and the values *timestamp* and *power*. By using a composite key Hadoop sorts the ids of wind turbines and, in a secondary sort, the timestamp for each id. Subsequently, the reduce method results in a simpler design as displayed in Listing 3.

Listing 3: Reduce function pseudo code

```
reduce(Object key, Iterator<object> values):
   // key:    an object containing id and timestamp
   // values: power values ordered by timestamp
   result = simpleMovingAverage(values)
   emit(id, result)
```

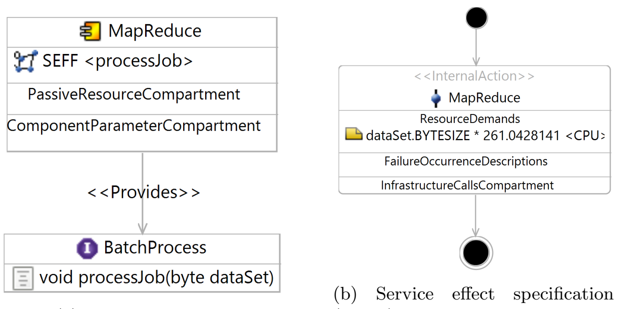
The reduce function is called for each different wind turbine and calculates the actual simple moving average. It receives the key object and a list of values as input which contains timestamps and power values sorted by the former. The function itself calculates the *result* and emits it with the corresponding wind turbine *id*.

3.3 Performance Model Prototype

We use the Palladio component model (PCM) [12] for our performance model. PCM is an annotated software architecture model that allows for describing performance relevant factors of software architecture, execution environment and usage profile [13]. Such performance models enable software architects and performance engineers to predict performance metrics such as response time, utilization or throughput by means of simulation or analytical solving.

PCM is divided into several sub-models. In the repository model, we specify a batch process as a software component with its service effect specification

(SEFF) to describe the resource demands of the provided service. In the resource environment model, we describe the hardware resources and processing rates on which a batch process will be executed. The concrete assignment of modeled batch processes to resources is determined in the allocation model. Finally, we specify the load intensity from wind turbine measurements in the usage model.


MapReduce
SEFF <processJob>
PassiveResourceCompartment
ComponentParameterCompartment
<<Provides>>
BatchProcess
void processJob(byte dataSet)
<<InternalAction>>
MapReduce
ResourceDemands
dataSet.BYTESIZE * 261.0428141 <CPU>
FailureOccurrenceDescriptions
InfrastructureCallsCompartment

(a) Repository model

(b) Service effect specification (SEFF) <processJob>

Fig. 6: Modeling a batch process with the Palladio component model

Figure 6 shows the substantials of modeling the batch process in our performance model. As shown in Figure 6a, we specify one interface *BatchProcess* with the method *processJob* to analyze an input data set. The implementation of the interface and its method is modeled by the component *MapReduce* with the corresponding SEFF. As illustrated in Figure 6b the SEFF itself solely consists of a CPU resource demand in dependence on an incoming data set size. The data set size is specified in the usage model, in our case, in gigabyte.

In order to define the CPU resource demand and simulate a realistic system behavior we integrated measurements into our performance model. Therefore, we measured response times of the MapReduce job described in Section 3.2 while running it. Afterwards, we used an approximation with response times, which is also implemented by the LibReDe library [32], to estimate the required CPU time each process takes per transaction. One transaction means exactly one batch process that analyzes a set of messages. In our case, the resulting resource demand we estimated is 261 as represented in Figure 6b.

In order to predict results, PCM instances must be first transferred to be either simulated or solved analytically. Available model transformations are a model-to-text transformation like SimuCom [12], queuing Petri nets (QPN) transformations as well as a transformation to layered queuing networks (LQN). Brosig et al. [13] evaluated these model transformations with regards to their efficiency and accuracy. In our application scenario, time is critical and the model need to be solved as efficiently as possible so resulting predictions are available at an early opportunity and the next batch process can be initiated. Therefore, we recommend the use of a model transformation to LQNs. It showed to be the most efficient solution as it is an analytical solver [13].

The performance model prototype has the limitation that is does not reflect the scheduling of processes itself within a cluster, for instance, as accomplished by Apache Hadoop YARN. Therefore, we assume sufficient available resources so batch and stream processes always run without interference.

3.4 Controlled Experiment

To conduct our experiments we run the mentioned data generator to produce CSV files for 10 wind farms with 100 wind turbines each, whereas one wind turbine approximately produces one measurement every second. Afterwards, we run the implemented Hadoop MapReduce job which reads only data measured within a sliding window of 24 hours. While the batch process is running, meanwhile we determine the incoming data volume. After the batch process is finished, we predict the response time of the second next batch process using our performance model. For the immediate succeeding batch process, we exactly know the data volume it will process as we know the historical data distribution and tracked new arrived data. For the batch process to be predicted, the data volume must be estimated. Therefore, a variety of specialized tools and algorithms exist to classify and forecast workload such as the approach by Herbst et al. [21]. As we target an efficient solution and a short-term forecast is required, namely, only the next point, we only use a naïve forecast in this study. It does not involve any computational overhead and simply takes the value of the latest observation as next forecast point in contrast to other methods such as cubic smoothing splines or ARIMA 101 that are more appropriate for scenarios with strong trends or noises [21]. In our case, the next forecast point equals the arrived data volume which has not been absorbed by the last batch process yet. Afterwards, we trigger the performance models with the predicted load intensity as input, and compare the predicted response time with the eventual measured response time.

As already mentioned, the aim is to minimize the usage of the speed layer. The level of potential resource reductions and costs savings that can be achieved depends on the characteristics of the underlying workload and variations in data distributions. The effectiveness of our solution itself, however, depends on how well the data volume is predicted and, especially, how accurate batch processes are predicted. Therefore, we concentrate on the latter in this controlled experiment and perform three selected scenarios with different load intensities by assuming different availabilities of wind turbines based on Faulstich et al. [19,20] to evaluate the accuracy of our solution.

In the first scenario, we assume the wind turbine availability (WTA) is constant during two following batch iterations. Consequently, the measurement data wind turbines produce do also not fluctuate so the predicted load intensity using a naïve forecast is very close to the actual measured load intensity. In the second scenario, we assume an increase of the WTA of 5 % for the subsequent batch process and, vice versa, we assume a decrease in a final third scenario. For each

Table 1: Measured and predicted results of batch processes

Scenario	WTA	Fluctuation	PRT	MRT	RE
1	85 %	± 0 %	12.78 minutes	12.17 minutes	5.01 %
	90 %	± 0 %	13.53 minutes	13.60 minutes	0.51 %
	95 %	± 0 %	14.28 minutes	15.47 minutes	7.69 %
2	85 %	+ 5 %	12.78 minutes	13.82 minutes	7.53 %
	90 %	+ 5 %	13.53 minutes	15.03 minutes	9.98 %
3	90 %	− 5 %	13.53 minutes	12.58 minutes	7.55 %
	95 %	− 5 %	14.28 minutes	13.17 minutes	8.43 %

scenario, we conduct several experiments with different WTA to also validate the prediction accuracy under different load intensities. Afterwards we compare predicted response times (PRT) with eventual measured response times (MRT) of the batch process and calculate the relative error (RE) of the PRT. The results are listed in Table 1.

For a WTA of 85% and no fluctuation during the following batch process, we predict the response time for the batch process to be 12.78 minutes. We measured a MRT of 12.17 minutes which leads to a RE of 5.01%. For a WTA of 90%, the RE of the predicted response time is only 0.51 % and 7.69% for a WTA of 95%.

In the second scenario, for a 85% WTA and a 5% increase of available wind turbines during the following batch iteration, the PRT is 12.78 minutes and the MRT 13.82 minutes with a 7.53% RE. Here, the PRT equals the same PRT as in the experiment for first scenario with a 85% WTA since the naïve forecast, as already mentioned, uses the last observation point, namely 85%, as next prediction point. The same occurrence also applies for the following experiments. The highest RE with 9.98% appeared for a WTA of 90% with +5% fluctuation at which the PRT is 13.53 minutes and the MRT 15.03 minutes.

For a decrease of the 5% WTA in the last scenario, we measured REs in the range similar to the former scenario. With a starting point of 90% WTA, the PRT is 13.53 minutes and the MRT 12.58 minutes. For 95% WTA, the PRT equals 14.28 minutes and MRT 13.17 minutes.

In our experiments, we showed that we are able to predict the response times of a batch process or MapReduce job, respectively, with RE between 0.51% and 9.98%. With regards to our exemplary use case, power can be traded every quarter of an hour in the intraday spot market. Assuming a fluctuating workload and a maximum acceptable response time of 14 minutes remaining one minute buffer, we would be able to accurately schedule stream processing in the second scenario, namely, not to switch on in the first experiment and to switch on stream processing in the second experiment as the MRT exceeds the time-constraint with 15.03 minutes. For a decreasing fluctuation, we would proper schedule stream processing for a starting WTA of 90%. However, for the last experiment in Table 1, we would have left the speed layer switched on as the PRT lies over 14 minutes in contrast to the MRT which is mainly caused by the naïve forecast.

4 Related Work

Similar to our use case, Sequeira et al. [31] propose a system based on the lambda architecture to analyze energy consumption. Martínez-Prieto et al. [25] adapted the architecture for semantic data and Casado and Younas [15] give an extensive review about technologies for the lambda architecture. Regarding optimization or efficient resource usage of the architecture, however, related research mainly focuses on the processing layers itself. For instance, Aniello et al. [3] and Rychl et al. [28] specify on scheduling stream processes, while Alrokayan et al. [1] concentrate on scheduling batch processes.

Regarding predicting batch processes, there is comprehensive research available, for instance, specialized for MapReduce jobs [11], [34], [35] as well as for big data applications in cloud infrastructures [16].

To overcome redundancy regarding software development and infrastructure complexity, approaches such as storm-yarn[6] or by Nabi et al. [27] exist to integrate stream processing in the Apache Hadoop environment. Summingbird[7] is an open source library that allows to write algorithms that can be used for batch as well as stream processing.

5 Conclusion and Future Work

This paper introduced a novel approach to use resources more efficiently when implementing the lambda architecture. It is applicable for usage scenarios where time constraints of queries are not permanently required to be low or lie within several minutes. To reduce processing power, we propose to switch on stream processing on demand in cases where batch processes are likely to exceed time requirements. By using historical information of incoming data and naïve forecasting to classify workload, we predicted the response time of succeeding batch iterations. Therefore, we used performance models in which we integrated estimated resource demands based on measurements. The results allow us to make decisions when additional stream processes are required or, vice versa, can be saved to reduce resource usage. If hardware provision is used in a as-a-service manner, it allows for reducing costs directly.

For future work we plan to automate the process illustrated in Figure 1. This involves to automatically measure incoming data during each batch iteration, apply workload forecasting techniques and trigger solving the performance model. Another challenge is to also integrate the speed layer into our test environment. This will enable us to examine our approach and its efficiency for successive batch iterations for a lengthy period of time. Furthermore, we will integrate other workload forecasting techniques besides the naïve forecast to evaluate possible prediction enhancements and scheduling optimizations.

[6] https://github.com/yahoo/storm-yarn
[7] https://github.com/twitter/summingbird

References

1. Alrokayan, M., Vahid Dastjerdi, A., Buyya, R.: Sla-aware provisioning and scheduling of cloud resources for big data analytics. In: Proceedings of the 2014 IEEE International Conference on Cloud Computing in Emerging Markets. pp. 1–8. IEEE (2014)
2. Amazon Web Services: Amazon Kinesis (2015), `http://aws.amazon.com/kinesis/`, accessed: 2015-04-28
3. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In: Proceedings of the 7th ACM International Conference on Distributed Event-based Systems. pp. 207–218. ACM, New York, NY, USA (2013)
4. Apache Cassandra: The Apache Cassandra project (2015), `http://cassandra.apache.org/`, accessed: 2015-04-28
5. Apache Hadoop: Welcome to Apache Hadoop! (2015), `http://hadoop.apache.org/`, accessed: 2015-04-28
6. Apache Kafka: A high-throughput distributed messaging system (2015), `http://kafka.apache.org/`, accessed: 2015-04-28
7. Apache Pig: Welcomt to Apache Pig! (2014), `https://pig.apache.org/`, accessed: 2015-04-28
8. Apache Samza: Samza (2015), `http://samza.apache.org/`, accessed: 2015-04-28
9. Apache Spark: Lightning-fast cluster computing (2015), `https://spark.apache.org/`, accessed: 2015-04-28
10. Apache Storm: Storm, distributed and fault-tolerant realtime computation (2015), `http://storm.apache.org/`, accessed: 2015-04-28
11. Barbierato, E., Gribaudo, M., Iacono, M.: Performance evaluation of nosql big-data applications using multi-formalism models. Future Generation Computer Systems 37(0), 345 – 353 (2014)
12. Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. The Journal of Systems and Software 82(1), 3–22 (2009)
13. Brosig, F., Meier, P., Becker, S., Koziolek, A., Koziolek, H., Kounev, S.: Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures. IEEE Transactions on Software Engineering 41(2), 157–175 (2015)
14. Brunnert, A., Vögele, C., Danciu, A., Pfaff, M., Mayer, M., Krcmar, H.: Performance management work. Business & Information Systems Engineering 6(3), 177–179 (2014)
15. Casado, R., Younas, M.: Emerging trends and technologies in big data processing. Concurrency and Computation: Practice and Experience 27(8), 2078–2091 (2015)
16. Castiglione, A., Gribaudo, M., Iacono, M., Palmieri, F.: Modeling performances of concurrent big data applications. Software: Practice and Experience (2014)
17. Chen, C.L.P., Zhang, C.Y.: Data-intensive applications, challenges, techniques and technologies: a survey on big data. Information Sciences 275(0), 314–347 (2014)
18. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
19. Faulstich, S., Hahn, B., Tavner, P.J.: Wind turbine downtime and its importance for offshore deployment. Wind Energy 14(3), 327–337 (2011)
20. Faulstich, S., Lyding, P., Tavner, P.: Effects of wind speed on wind turbine availability (2011)

21. Herbst, N.R., Huber, N., Kounev, S., Amrehn, E.: Self-adaptive workload classification and forecasting for proactive resource provisioning. Concurrency and Computation: Practice and Experience 26(12), 2053–2078 (2014)
22. von Kistowski, J., Herbst, N.R., Kounev, S.: LIMBO: A tool for modeling variable load intensities. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering. pp. 225–226. ACM, New York, NY, USA (2014)
23. Kroß, J., Brunnert, A., Prehofer, C., Runkler, T.A., Krcmar, H.: Model-based performance evaluation of large-scale smart metering architectures. In: Proceedings of the 4th International Workshop on Large-Scale Testing. pp. 9–12. ACM, New York, NY, USA (2015)
24. Liu, X., Iftikhar, N., Xie, X.: Survey of real-time processing systems for big data. In: Proceedings of the 18th International Database Engineering & Applications Symposium. pp. 356–361. ACM, New York, NY, USA (2014)
25. Martínez-Prieto, M.A., Cuesta, C.E., Arias, M., Fernánde, J.D.: The solid architecture for real-time management of big semantic data. Future Generation Computer Systems 47, 62 – 79 (2015), special Section: Advanced Architectures for the Future Generation of Software-Intensive Systems
26. Marz, N., Warren, J.: Big data: principles and best practices of scalable real-time data systems. Manning Publications Co. (2015)
27. Nabi, Z., Wagle, R., Bouillet, E.: The best of two worlds: integrating ibm infosphere streams with apache yarn. In: Proceedings of the 2014 IEEE International Conference on Big Data. pp. 47–51. IEEE (2014)
28. Rychlý, M., Škoda, P., Smrž, P.: Heterogeneity-aware scheduler for stream processing frameworks. International Journal of Big Data Intelligence 2(2), 70–80 (2015)
29. Schäfer, A.M., Zimmermann, H.G.: Recurrent neural networks are universal approximators. In: Kollias, S., Stafylopatis, A., Duch, W., Oja, E. (eds.) Artificial Neural Networks - ICANN 2006, Lecture Notes in Computer Science, vol. 4131, pp. 632–640. Springer Berlin Heidelberg (2006)
30. Schermann, M., Hemsen, H., Buchmüller, C., Bitter, T., Krcmar, H., Markl, V., Hoeren, T.: Big data - an interdisciplinary opportunity for information systems research. Business & Information Systems Engineering 6(5), 261–266 (2014)
31. Sequeira, H., Carreira, P., Goldschmidt, T., Vorst, P.: Energy cloud: Real-time cloud-native energy management system to monitor and analyze energy consumption in multiple industrial sites. In: Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. pp. 529–534. IEEE (2014)
32. Spinner, S., Casale, G., Zhu, X., Kounev, S.: LibReDE: a library for resource demand estimation. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014). pp. 227–228. ACM, New York, NY, USA (2014)
33. Taylor, J.W.: An evaluation of methods for very short-term load forecasting using minute-by-minute british data. International Journal of Forecasting 24(4), 645 – 658 (2008)
34. Verma, A., Cherkasova, L., Campbell, R.H.: Aria: automatic resource inference and allocation for mapreduce environments. In: Proceedings of the 8th ACM International Conference on Autonomic Computing. pp. 235–244. ACM, New York, NY, USA (2011)
35. Vianna, E., Comarela, G., Pontes, T., Almeida, J., Almeida, V., Wilkinson, K., Kuno, H., Dayal, U.: Analytical performance models for mapreduce workloads. International Journal of Parallel Programming 41(4), 495–525 (2013)

Modeling Big Data Systems by Extending the Palladio Component Model

Johannes Kroß,
Andreas Brunnert
fortiss GmbH
Guerickestr. 25
80805 Munich, Germany
{kross,brunnert}@fortiss.org

Helmut Krcmar
Chair for Information Systems
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
krcmar@in.tum.de

ABSTRACT

The growing availability of big data has induced new storing and processing techniques implemented in big data systems such as Apache Hadoop or Apache Spark. With increased implementations of these systems in organizations, simultaneously, the requirements regarding performance qualities such as response time, throughput, and resource utilization increase to create added value. Guaranteeing these performance requirements as well as efficiently planning needed capacities in advance is an enormous challenge. Performance models such as the Palladio component model (PCM) allow for addressing such problems. Therefore, we propose a meta-model extension for PCM to be able to model typical characteristics of big data systems. The extension consists of two parts. First, the meta-model is extended to support parallel computing by forking an operation multiple times on a computer cluster as intended by the single instruction, multiple data (SIMD) architecture. Second, modeling of computer clusters is integrated into the meta-model so operations can be properly scheduled on contained computing nodes.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

Keywords

Palladio Component Model, Performance Model, Big Data

1. INTRODUCTION

Exponentially growing volumes of data of various formats—referred to as big data—and the necessity of organizations to gain benefits have led to the development of big data systems [6], [10]. These systems are specialized for storing and processing this data. A common example includes Apache Hadoop[1] consisting of a distributed file system called *HDFS*, a scheduler and cluster resource manager called *YARN* and the *MapReduce* model for parallel data processing [8].

Although Apache Hadoop is originally built for commodity hardware, other systems such as Apache Spark (Streaming)[2] and Apache Storm[3] have emerged that enable low latency results on big data by also using in-memory computing [13]. Therefore, big data systems are able to meet continuously increasing performance requirements and to serve several additional use cases. Consequently, up-front performance evaluations for these systems and capacity planning for building an appropriate cluster become not only inevitable, but also difficult and costly [4, 5].

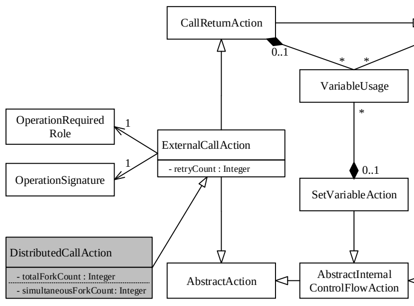
One way to approach these challenges are performance models such as the Palladio component model (PCM) that focuses on component-based software architectures [2]. It allows to model factors influencing system performance and predict performance metrics such as resource utilization, response time, and throughput by analytical solving or simulation [3]. As the PCM meta-model does not allow to model some specific requirements of big data systems yet, we propose and contribute a meta-model extension in this paper. This includes to specify an external call of an action to be executed multiple times in parallel while limiting the number of concurrent actions. It also includes to model a resource cluster consisting of several resource containers with different resource roles such as found in distributed computing architectures.

[1]http://hadoop.apache.org/
[2]http://spark.apache.org/
[3]http://storm.apache.org/

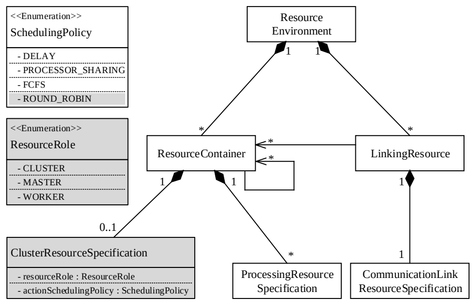
2. MODELING BIG DATA SYSTEMS

Comparing big data systems to ordinary component-based software systems (e.g., for web applications), they make use of specialized processing paradigms. Casado and Younas [6] list two main techniques that are common for big data systems, namely, batch and stream processing. They have in common that they utilize parallel and distributed computing on a distributed system in the form of a computer cluster. For this purpose, software developers implement components with operation signatures, for instance by using software libraries such as Apache Hadoop, and combine these components to build a task job that will be deployed on a computer cluster. In order to distribute this task job across linked resources, the components can be assembled in the form of a directed acyclic graph (DAG) [13]. For instance, the *MapReduce* paradigm consists of two vertices *map* and *reduce* that are linked by a directed edge. By this means, such systems are able to make use of all distributed computing resources and achieve horizontal scalability for increased workloads in terms of data volume or velocity.

Despite their shared characteristics, batch and stream processing adopt distinct approaches and are designed for different use cases. Batch processing is intended to be used on data sets with high volume [6]. In doing so, a specified operation is applied on splits of a non changing distributed data set multiple times in parallel. For instance, implemented Hadoop MapReduce operations are applied on distributed



(a) Service effect specification (SEFF) actions



(b) Resource environment

Figure 1: Meta-model extension for the Palladio component model (PCM, Version 3.4.1)

files on the HDFS. Implemented operations using Apache Spark are applied on so called resilient distributed datasets (RDD). The amount of parallelism for one specified operation is usually limited by the split rate of a dataset. The amount of simultaneously running parallel operations is usually limited by the amount of available resources or by specified user configurations.

Stream processing, on the other hand, is designated for handling high velocity data streams with low latency and is also referred to as real-time processing [6]. It distinguishes itself from batch processing by not operating on a data set, but rather operating on each data (e.g., Apache Storm) or a mini-batch (e.g., Apache Spark Streaming) that are kept in-memory. Therefore, data are continuously received from an unbounded data stream (e.g., in a message queue manner) and immediately processed by an operation. Similar to batch processing, the number of simultaneously running operations is limited by the amount of available resources or by specified configurations.

In previous work [9] we already modeled one *MapReduce* job on a single computer and predicted its response time. As we had to simplify several features and take limitations into account, we identified the need to extend PCM. Based on these findings and the above mentioned characteristics of batch and stream processing, we derive the following requirements of big data systems that we propose to extend PCM in order to allow for modeling typical big data systems:

1. **Distribution and parallelization of operations**
   Component developers specify reusable software components consisting of operations using software frameworks like Apache Spark. In doing so, they may specify, but also may not know the definite number of simultaneous and/or total executions of an operation.

2. **Clustering of resource containers**
   System deployers specify resource containers with resource roles (e.g., master or worker nodes), link them to a mutual network and logically group them to a computer cluster.

On this basis, we propose the following extensions for the PCM meta-model, which are shown in gray in Figure 1 (note that we only depict the relevant parts of the meta-model regarding our approach). The PCM meta-model consists of several partial models according to different developer roles [2]. Figure 1a shows the actions of the service effect specification (SEFF) model. A SEFF describes the behavior of an implemented operation. The element we propose to extend is the *ExternalCallAction* that is used to call a required service [2]. Therefore, we introduce a *DistributedCallAction*. It contains the two additional input parameters *simultaenousForkCount* and *totalForkCount* that can be used to specify the simultaneous and/or total number of executions of an external call as mentioned in the first requirement. Since these parameters depend on the workload and resource environment, component developers can describe the two input parameters as well as the resource demand of an operation as dependencies in parameterized form. In this way, domain experts are able to specify the usage of the component afterwards as proposed by Becker et al. [2].

Figure 1b shows the meta-model extension for the resource environment. Here, a *ResourceContainer* may or may not have several *ProcessingResourceSpecifications* to specify e.g., processors and hard disks. A *ResourceContainer* can also have a set of nested *ResourceContainers*. We propose to complement the *ResourceContainer* by a *ClusterResourceSpecification* which contains references to one *ResourceRole* as well as one *SchedulingPolicy*. These are both part of a *ResourceRepository*, that is intended to contain types of resources such as for middleware and operating system resources [2]. A *ResourceRole* is used to describe whether a *ResourceContainer* represents a cluster, a master or a worker. A *SchedulingPolicy* is used to describe how actions are distributed on a cluster.

An example for a modeled computer cluster is shown in Figure 2. An outer *ResourceContainer* is used to connect

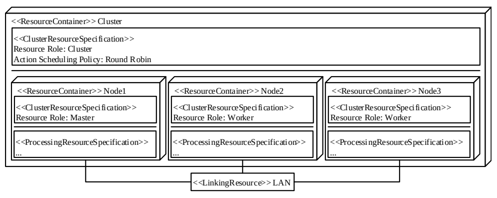

<<ResourceContainer>> Cluster
<<ClusterResourceSpecification>>
Resource Role: Cluster
Action Scheduling Policy: Round Robin
<<ResourceContainer>> Node1
<<ClusterResourceSpecification>>
Resource Role: Master
<<ProcessingResourceSpecification>>
...
<<ResourceContainer>> Node2
<<ClusterResourceSpecification>>
Resource Role: Worker
<<ProcessingResourceSpecification>>
...
<<ResourceContainer>> Node3
<<ClusterResourceSpecification>>
Resource Role: Worker
<<ProcessingResourceSpecification>>
...
<<LinkingResource>> LAN

Figure 2: Example for a resource environment diagram

computing nodes to a cluster and includes a round robin strategy to schedule actions on its nested *ResourceContainer*. This enables system deployers to simply allocate components to a cluster. Furthermore, each nested *ResourceContainer* includes a *ResourceRole*. If only workers are specified, the cluster will represent a shared-nothing architecture. If one master is specified, it will be responsible for distributing actions. Therefore, its *ResourceContainer* operates usually special middleware with additional resource demands that can be modeled with *InfrastructureCalls* in PCM.

3. RELATED WORK

Most of the existing performance modeling approaches for big data systems concentrate only on one technology, namely Apache Hadoop. Barbierato et al. [1] introduce a performance modeling language to evaluate the performance of queries using Apache Hive which is a data warehouse software on top of Apache Hadoop with a SQL-like language. Vianna et al. [12] propose an analytical model, which combines a precedence graph model and a queuing network model, to model MapReduce workloads concentrating on the pipeline parallelism between *map* and *reduce* operations. Verma et al. [11] propose a framework consisting of micro benchmarks and a regression-based model to predict and evaluate response times of MapReduce processes for different cluster resource choices.

A more general approach regarding big data technologies is, for instance, introduced by Castiglione et al. [7] which use Markovian agents and mean field analysis to model big data batch applications and to provide information about performance of cloud-based data processing architectures. However, there is no approach available to the best of our knowledge that tries to enable modeling of general batch and stream processes, and to predict the response time and cluster resource utilization for their concurrent execution.

4. CONCLUSION AND FUTURE WORK

In this paper we introduced a generic performance modeling formalism to model essential characteristics of data processing as found in big data systems. For this purpose, we presented two meta-model extensions for PCM that enable performance engineers to model a computer cluster and to apply distributed and parallel operations on this cluster. This allows to model general stream processing as well as batch processing techniques independent of their technology and to realize up-front performance evaluations for response times, throughputs, and resource utilizations of CPU and memory of big data systems.

We already implemented the meta-model extensions, graphical modeling editors, model-2-code transformations and basic functionalities of the associated simulation framework SimuCom [2] to support our extensions. Although we do not consider network traffic between resource containers yet, first experimental results already look promising. In future, we plan to complete the SimuCom extension as well as integrate network traffic. Afterwards, we intend to comprehensively evaluate our meta-model extension in controlled experiments. This includes up- and downscaling scenarios regarding workload as well as resource capacities. Our long-term goal is to automatically derive performance models for batch and stream processes based on measurement data from middleware like Apache YARN.

5. REFERENCES

[1] E. Barbierato, M. Gribaudo, and M. Iacono. Performance evaluation of NoSQL big-data applications using multi-formalism models. *Future Generation Computer Systems*, 37(0):345 – 353, 2014.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *The Journal of Systems and Software*, 82(1):3–22, 2009.

[3] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev. Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures. *IEEE Transactions on Software Engineering*, 41(2):157–175, 2015.

[4] A. Brunnert and H. Krcmar. Continuous performance evaluation and capacity planning using resource profiles for enterprise applications. *Journal of Systems and Software*, 2015. doi: 10.1016/j.jss.2015.08.030.

[5] A. Brunnert, C. Vögele, A. Danciu, M. Pfaff, M. Mayer, and H. Krcmar. Performance management work. *Business & Information Systems Engineering*, 6(3):177–179, 2014.

[6] R. Casado and M. Younas. Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, 27(8):2078–2091, 2015.

[7] A. Castiglione, M. Gribaudo, M. Iacono, and F. Palmieri. Modeling performances of concurrent big data applications. *Software: Practice and Experience*, 2014.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] J. Kroß, A. Brunnert, C. Prehofer, T. Runkler, and H. Krcmar. Stream processing on demand for lambda architectures. In M. Beltrán, W. Knottenbelt, and J. Bradley, editors, *Computer Performance Engineering*, volume 9272 of *Lecture Notes in Computer Science*, pages 243–257. Springer International Publishing, 2015.

[10] M. Schermann, H. Hemsen, C. Buchmüller, T. Bitter, H. Krcmar, V. Markl, and T. Hoeren. Big data - an interdisciplinary opportunity for information systems research. *Business & Information Systems Engineering*, 6(5):261–266, 2014.

[11] A. Verma, L. Cherkasova, and R. H. Campbell. Profiling and evaluating hardware choices for MapReduce environments: An application-aware approach. *Performance Evaluation*, 79:328 – 344, 2014.

[12] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, and U. Dayal. Analytical performance models for MapReduce workloads. *International Journal of Parallel Programming*, 41(4):495–525, 2013.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

Modeling and Simulating Apache Spark Streaming Applications

Johannes Kroß
fortiss GmbH
Guerickestr. 25
80805 Munich, Germany
kross@fortiss.org

Helmut Krcmar
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
krcmar@in.tum.de

Abstract

Stream processing systems are used to analyze big data streams with low latency. The performance in terms of response time and throughput is crucial to ensure all arriving data are processed in time. This depends on various factors such as the complexity of used algorithms and configurations of such distributed systems and applications. To ensure a desired system behavior, performance evaluations should be conducted to determine the throughput and required resources in advance. In this paper, we present an approach to predict the response time of Apache Spark Streaming applications by modeling and simulating them. In a preliminary controlled experiment, our simulation results suggest accurate prediction values for an upscaling scenario.

1 Introduction

Big data systems enable organizations to store and analyze data with high volume, velocity, and variety [4]. Corresponding processing techniques can be categorized into batch and stream processing [5]. Stream processing systems receive broad concentration nowadays as algorithms, transformations, and windowing mechanisms for streaming data constantly become more sophisticated and libraries for machine learning are available. They are mainly applied to process data and provide results in real time [5]. Therefore, the performance of such systems is particularly significant, for instance, to ensure high throughput for different workload scenarios and prevent queueing up of input stream data. However, planning the requirements of such applications and systems is complicated since environments and conditions to evaluate the performance for different scenarios, system configurations, and realistic workloads are usually not met as in productive environments [3, 8].

We propose a modeling and simulation approach to predict the response time of stream processing applications i.e., Apache Spark Streaming. Therefore, we use the Palladio component model (PCM) [1] and an extension for big data systems that we have presented in previous work [7]. The extension is open source[1] and includes a distributed call action to model parallel and distributed external calls in a service effect specification (SEFF) and a cluster resource specification to model a cluster of master and worker nodes and distribute resource demands to worker nodes.

[1]https://git.fortiss.org/pmwt/bd.pcm.extension

2 Related Work

Regarding modeling, simulation, or analytical solving the performance of big data systems, most of prior research focuses on Apache Hadoop and its MapReduce paradigm and, therefore, on batch processing. There is one approach by Wang and Khan [9], that focuses on predicting the response time of Apache Spark applications, however, only batch applications. Regarding stream processing, there is one patent by Ginis and Strom [2] that describes a method on predicting the performance of publish-subscribe middleware messaging systems using queueing theory that, however, does not take resource demands for CPU, memory, or hard disk drives into account.

3 Modeling and Simulation Approach

There are several known stream processing systems available such as Apache Samza, Apache Storm, Apache Spark Streaming and Apache Flink [6]. We focus on Apache Spark Streaming[2] in this paper as it is one of the sophisticated technologies with a large community and supported by known benchmarks. It comprises a micro-batch model, in contrast to other technologies that use an operator-based model [6].

A Spark Streaming application is constructed as follows[3]: it receives incoming data from streaming sources using a discretized stream called *DStream*. This data is fetched in the form of a micro-batch *job* that is iteratively executed in stream intervals. A *DStream* is represented by several resilient distributed datasets (*RDDs*). Afterwards, transformations such as *map* or *reduce* operations can be applied on a *DStream*. Spark builds a distributed acyclic graph (DAG) based on these related operations and splits them into *stages* of *tasks*. The number of parallel *tasks*

[2]http://spark.apache.org/streaming/

[3]https://spark.apache.org/docs/1.6.0/streaming-programming-guide.html

is limited by the number of partitions of an *RDD*. Furthermore, transformations with narrow dependencies are consolidated in one *stage*, for instance, *map* and *filter* operations that do not require to shuffle data. *Stages* are executed sequentially and finally make up one *job*.

In order to model a Spark Streaming application, we specify one *job* executor component with a SEFF that involves a loop to start several asynchronous forked behaviours as displayed in Figure 1.

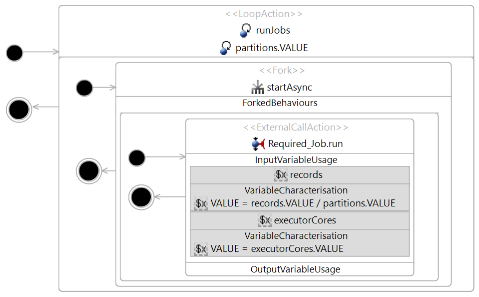
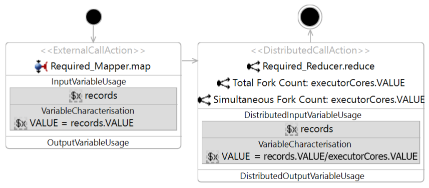


Figure 1: SEFF of job executor component

The loop length and, therefore, the number of executed behaviors depends on the value of the parameter *partitions* that is used to describe the number of topic partitions. In the forked behavior, we call the SEFF of the *stage* executor component with the parameters *records* and *executorCores*. The former parameter describes the number of records for each partition, the latter the number of cores that is configured when starting a Spark application with the equivalent parameter *spark.executor.cores*. The SEFF is illustrated in Figure 2.



<<ExternalCallAction>>
Required_Mapper.map
InputVariableUsage
records
VariableCharacterisation
VALUE = records.VALUE
OutputVariableUsage
<<DistributedCallAction>>
Required_Reducer.reduce
Total Fork Count: executorCores.VALUE
Simultaneous Fork Count: executorCores.VALUE
DistributedInputVariableUsage
records
VariableCharacterisation
VALUE = records.VALUE/executorCores.VALUE
DistributedOutputVariableUsage

Figure 2: SEFF of stage executor component

For each *stage*, we model an external call or distributed call [7], respectively, with the number of *records* as input parameter. Our sample application involves two stages *map* and *reduce*. The first SEFF *map* is invoked once since there is one *DStream* for each partition. The second SEFF is invoked with a distributed call of which the parallelism depends on the *executorCores* value. The *map* and *reduce* SEFFs involve three consecutive internal actions each with one parametric resource demands to specify the scheduler delay, serialization time, and computing time.

In order to model the hardware environment, we specify a resource container with a cluster resource specification [7] for each node. For the master node, we model one parent resource container that includes a round robin action scheduling policy and a master resource role. Dependent on the number of worker nodes, we specify several nested resource containers with a worker resource role. In the usage model, we invoke the *job* executor component with its three input parameters, model a closed workload with one user, and specify the think time according to the stream interval.

4 Controlled Experiment

In our controlled experiment, we use the HiBench benchmark suite[4] of which we use the *distinct count* application. It involves two *stages map* and *reduce*. Therefore, data are streamed to a so-called *topic* in an Apache Kafka[5] cluster, a distributed publish-subscribe messaging system. The application is connected to that *topic* and applies a direct stream to query data from Apache Kafka using *DStreams*. Thereby, the level of parallel streams is defined by the number of partitions of one *topic* which, consequently, equals the number of *map stages*.

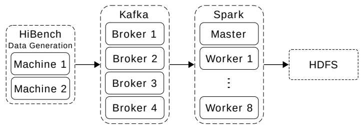


Figure 3: Testbed setup

Our experimental setup is shown in Figure 3. For the workload, we setup 2 virtual machines (VMs) that we use to generate data and a cluster consisting of 4 VMs for Apache Kafka brokers. For the application, we setup 1 VM for the master node, and 8 VMs for the worker nodes where the benchmark application will be executed. We use Apache YARN (2.7) as cluster manager and Apache Spark (1.6) as processing framework. We specified one Spark executor per worker node with 6 cores and 24 gigabytes memory.

We conducted four scenarios with a stream interval of 10 seconds as listed in Table 1.

Table 1: Conducted experiments

Scenario	**Workload** (events/second)	**Kafka broker**	**Topic partitions**	**Spark worker**
2 nodes	~ 450,000	1	2	2
4 nodes	~ 450,000	2	4	4
6 nodes	~ 450,000	3	6	6
8 nodes	~ 450,000	4	8	8

Based on the 2 nodes scenario, we measured the delay and CPU resource demands for all *tasks* using the Spark monitoring API, adjusted the demands in dependence of the number of records, and included them into our repository model as listed in Table 2. We used this repository model for all upscaling scenarios. We adapted the number of workers in the resource

[4]https://github.com/intel-hadoop/HiBench
[5]http://kafka.apache.org/

environment model and the partition parameter in the usage model for each scenario.

Table 2: Parametric resource demands

Map SEFF	
scheduler delay	10
deserialization	$0.000078549 * \text{records.VALUE}$
computing	$Norm(0.003320415 * \text{records.VALUE}, 0.0001553647 * \text{records.VALUE})$
Reduce SEFF	
scheduler delay	10
deserialization	$0.000013227 * \text{records.VALUE}$
computing	$0.000023370 * \text{records.VALUE}$

A boxplot of the measured response time (MRT) and the simulated response time (SRT) is illustrated in Figure 4. For the 2 nodes scenario, the mean MRT is 7.88 seconds and the mean SRT is 7.94 seconds, which gives a relative reponse time prediction error (RTPE) of 0.67%. In the 4 nodes scenario, the values deviate more with a RTPE of 21.14%. Our analysis of the measurements suggests that the processing time for each task did not behave as linear as in the other scenarios. In the 6 nodes and 8 nodes scenarios, the RTPE result in 3.41% and 2.26%.

Our models, simulation and measurements results, and analysis script are publicly available online [10].

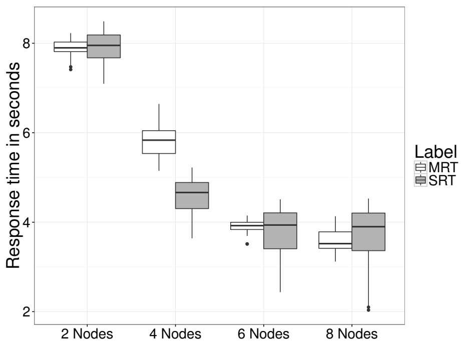


Figure 4: Measured and simulated response times

5 Conclusion and Future Work

In this paper, we proposed a modeling approach for stream processing systems using the example of Apache Spark Streaming. In a small controlled experiments, we simulated an upscaling scenario in which we increased the cluster size. Our predicted response times approach the measured ones closely.

At the moment, our extension for the simulation framework is for PCM 3.4.1 and we only consider delay and CPU demands. Therefore, we plan to incorporate our extension in the up to date PCM version and to additionally evaluate resources such as memory and network. Furthermore, we plan to extend our approach for operator-based processing frameworks such as Apache Flink and Apache Storm. Our long-term goal is to automatically derive performance models based on monitoring data, e.g., provided by APIs of processing frameworks.

References

[1] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: *The Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[2] Roman Ginis and Rober E. Strom. *Method for predicting performance of distributed stream processing systems.* US Patent 7,818,417. Oct. 2010.

[3] Andreas Brunnert et al. “Performance management work”. English. In: *Business & Information Systems Engineering* 6.3 (2014), pp. 177–179.

[4] Michael Schermann et al. “Big Data - an interdisciplinary opportunity for information systems research”. In: *Business & Information Systems Engineering* 6.5 (2014), pp. 261–266.

[5] Ibrahim A.T. Hashem et al. “The rise of “big data” on cloud computing: Review and open research issues”. In: *Information Systems* 47 (2015), pp. 98–115.

[6] Guenter Hesse and Martin Lorenz. “Conceptual Survey on Data Stream Processing Systems”. In: *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on.* Dec. 2015, pp. 797–802.

[7] Johannes Kroß, Andreas Brunnert, and Helmut Krcmar. “Modeling Big Data Systems by Extending the Palladio Component Model”. In: *Softwaretechnik-Trends* 35.3 (Nov. 2015).

[8] Johannes Kroß et al. “Stream Processing on Demand for Lambda Architectures”. English. In: *Computer Performance Engineering.* Ed. by Marta Beltrán, William Knottenbelt, and Jeremy Bradley. Vol. 9272. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 243–257.

[9] Kewen Wang and Mohammad M.H.K. Khan. “Performance Prediction for Apache Spark Platform”. In: *Proceedings of the 17th International Conference on High Performance Computing and Communications (HPCC).* New York, NY: IEEE, Aug. 2015, pp. 166–173.

[10] Johannes Kroß and Helmut Krcmar. *Models, Simulations, Measurements, and Analysis for Modeling and Simulating Apache Spark Streaming Applications.* Available: http://dx.doi.org/10.5281/zenodo.61243. Aug. 2016.

Model-based Performance Evaluation of Batch and Stream Applications for Big Data

Johannes Kroß
Model-based Systems Engineering
fortiss GmbH
Guerickestr. 25
80805 Munich, Germany
Email: kross@fortiss.org

Helmut Krcmar
Chair for Information Systems
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
Email: krcmar@in.tum.de

***Abstract*—Batch and stream processing represent the two main approaches implemented by big data systems such as Apache Spark and Apache Flink. Although only stream applications are intended to satisfy real-time requirements, both approaches are required to meet certain response time constraints. In addition, cluster architectures continuously expand and computing resources constitute high investments and expenses for organizations. Therefore, planning required capacities and predicting response times is crucial. In this work, we present a performance modeling and simulation approach by using and extending the Palladio component model. We predict performance metrics of batch and stream applications and its underlying processing systems by the example of Apache Spark on Apache Hadoop. Whereas most related work concentrates on one specific processing technique and focuses on the metric response time, we propose a general approach and consider the utilization of resources as well. In different experiments we evaluated our approach using applications and data workloads of the HiBench benchmark suite. The results indicate accurate predictions for upscaling cluster sizes as well as workloads with errors less than 18%.**

I. Introduction

The emergence of big data systems enabled organizations to store and process data with high volume, variety and velocity [1]. The Apache Hadoop family and the MapReduce paradigm paved the way for big data applications to be implemented in various areas across all industries [2], [3]. Whereas these technologies were first designed to run on commodity hardware, frameworks such as Apache Spark arose and increased the performance of long-running applications. Their primary focus is to process a historical set of data in batches. Since there was also a need to analyze emerging data as they arrive, stream processing systems such as Apache Storm and Spark Streaming were developed in recent years.

The performance in terms of metrics such as response time, throughput, and resource utilization is a crucial aspect for both types of applications and depends on a variety of factors [4]. It is vital, but also complicated to estimate the behavior and evaluate the impact of different scenarios such as changing data workload and resources [5]. When deploying a big data application from a test to a production environment, for instance, data scientists are confronted with the challenge on how to size resource capacities in order to guarantee certain response times. Performance models represent an established way in order to address these challenges [6]. They depict software systems, analytical solve or simulate their behavior, and predict different metrics [7]. Regarding big data applications, however, most related approaches focus on specific technologies (i.e., MapReduce) and processing types (i.e., batch). Furthermore, most efforts only consider the response time of applications in their approaches leaving out demands for resources. We propose and contribute a modeling and simulation approach for batch and stream processing systems by the example of Apache Spark. It includes resource demands and allows for predicting response times as well as resource utilization. Therefore, we use and extend the Palladio component model (PCM), a model designed for component-based software systems that represents performance-influencing factors on architecture-level [8]. Our extension and approach allows for simulating parallel operations as well as distributing them on a cluster of hardware resources. It supports big data architects to plan required capacities and examine the performance behavior under different conditions such as changing data workload.

In this paper, we first describe related literature in Section II. In Section III, we give an overview of batch and stream processing by the example of Apache Spark. Afterwards, we describe our modeling and simulation approach in Section IV. In Section V, we assess the prediction accuracy of our approach and outline assumptions and limitations. Finally, we conclude our work and describe future activities in Section VI.

II. Related Work

Most of the former related work concentrates on the MapReduce paradigm or complementary database technologies such as Apache Hive or HBase. Many approaches also focus on the metric response time and do not consider resource demands and utilizations. Vianna et al. [9] present a hierarchical model which combines a precedence graph mode as well as a queuing network model to predict the response time of MapReduce applications. They specifically focus on the intra-job synchronization delays between map and reduce tasks. Verma et al. [10] present a framework to predict the response time of MapReduce applications before migrating to a different cluster with different hardware. Therefore, they use micro-benchmarks on the initial cluster and a regression-based approach to model hardware differences between the initial and new cluster. Zhang et al. [11] present a framework including a platform performance model to depict different phases of a MapReduce application and predict the execution time in dependence on a new data set. For their approach they apply the model of the ARIA framework by Verma et al. [10].

Barbierato et al. [12] developed a language for the description of performance models which includes MapReduce applications. The approach allows to predict the response time. As main component the model uses the SQL-like query language of Apache Hive. Ardagna et al. [13] proposed approaches to estimate response times of Hive requires. Therefore, they presented multiple performance analysis models with increasing complexity and accuracy such as queueing networks and stochastic well formed nets. Lehrig [14] proposes an early design-time scalability/elasticity analysis of Software-as-a-Service (Saas) applications using architectural templates for Palladio. They plan to enrich it by big data technologies on the data layer such as replicable NoSQL databases and a MapReduce programming model. One general approach to model the behavior of batch applications is proposed by Castiglione et al. [15]. They use Markovian agents and mean field analysis to predict the behavior of concurrent interactive cloud, batch, and time constrained applications. However, they focus on cloud infrastructures and evolution dynamics of applications rather than on predicting performance metrics. Niemann [16] present another approach to predict the performance and energy consumption of Apache Cassandra, a distributed data management system. They use queueing Petri nets for various workload as well as cluster sizes.

As part of the DICE EU project, Casale et al. [17] propose a model-driven engineering for quality assurance of data-intensive software systems concentrating on Apache Hadoop, NoSQL databases, and stream processing (i.e., Apache Storm). Their approach aims at simulation, verification, and optimization for big data applications. The models contain three different model layers including a platform-independent model, a technology-specific model and a deployment-specific model [18]. Gómez et al. [19] also propose a strategy to transform the models into stochastic Petri nets. It shall enable engineers to asses performance requirements and they are currently validating their approach. For Apache Spark, Wang and Khan [5] propose a simulation-driven prediction model that focuses on estimating response times. They also include read and write operations for hard disk drives (HDD) and the allocation of memory. Venkataraman et al. [20] presented the framework Ernest for predicting the performance for analytical jobs using e.g., Apache Spark based on a optimal experiment design. Therefore, they predict the response time of applications ins dependence of the number of cluster nodes.

Regarding stream processing, there is one patent by Ginis and Strom [21] that describes a method on predicting the performance of publish-subscribe middleware messaging systems using queueing theory that, however, does not take resource demands for CPU, memory, or hard disk drives into account.

III. Big Data Applications and Systems

There is a huge variety of big data solutions available that use different computing techniques [22]. In the following, we give an overview over batch and stream processing systems by the example of Apache Spark.

A. Batch Processing

Batch applications are designed to process a huge amount of historical data in a distributed and parallel way [22]. The Apache Spark framework is example for such applications and introduces so called resilient distributed datasets (RDDs) to keep and reuse data in memory. RDDs are parallel data structures to store intermediate results in memory and offer coarse-grained operations that can be applied on them [23]. An application is executed by forming a distributed acyclic graph (DAG) based on associated operations and grouping them into stages of tasks. A stage chains up operations with narrow dependencies in case a shuffle is not required [23]. The number of tasks of one stage depends on the number of RDD partitions. Stages are executed successively and constitute one job. One or more sequential jobs compose one Spark application. The application is orchestrated by one context, which runs in the main process called the driver program. It is responsible for allocating executors to worker nodes as well as scheduling tasks of an application on executors. An executor is a process that runs tasks in parallel. An application has always its own executors assigned in order to be isolated from other applications [24].

B. Stream Processing

For applications that require to continuously analyze huge volumes of live data with low latency, stream processing systems are specialized for this purpose [22]. There are mainly two approaches - one mini-batch model and one continuous operator-based model [25], [26]. The former divides data streams into mini-batches and allows for batch processing, whereas the latter fetches and processes each record (e.g., Apache Flink) [25]. Apache Spark provides an extension module called Spark Streaming to apply the mini-batch model on data streams and reuse its core functionality. Therefore, Spark introduces discretized streams (DStreams). They allow for representing stream computations as a series of batch computations on mini-batch intervals and are represented as an ordered series of RDDs - one RDD for each interval [26]. Starting point of the data processing workflow is an input data stream that may be partitioned to increase parallel computing. Spark Streaming receives incoming data from such a stream source using a DStream and creates one RDD for each interval with the same amount of partitions as the input stream. Afterwards, transformations such as map or reduce operations can be applied on a DStream and RDD, respectively. As before, Spark builds a DAG based on related operations and splits these into stages of tasks. In contrast to batch processing, one job is created for each mini-batch. Jobs are continuously executed sequentially and always contain the same set of stages and tasks.

IV. Modeling and Simulation Approach

This section first describes the extension for PCM. Afterwards, the derivation of the models and resource demands is outlined for batch applications followed by stream applications. Subsequently, the specification of cluster resources is described as well as the representation of data workloads.

A. Extending the Palladio Component Model

PCM enables engineers to describe performance relevant factors of software architectures [7]. It is implemented using the Eclipse Modeling Framework and consists of several models [8]. Software interfaces and components are specified in the repository model. Components provide the implementation for signatures of interfaces. Therefore, they contain a service effect specification (SEFF) in which the activities such as parametric resource demands and external calls of signatures are modeled. In the resource environment model, network and hardware resources are specified. The allocation model allows for deploying components on resources. The usage and workload is specified in the usage model.

In previous work, we already modeled and simulated big data applications [27], [28]. Since PCM was not developed to support distributed and parallel computing as well as cluster architectures, we propose two meta-model extensions [29]. A *DistributedCallAction* is added to the SEFF meta-model. It extends the *ExternalCallAction* that is used to invoke a remote signature of a required service. The *DistributedCallAction* includes two variables *totalForkCount* and *simultaneousForkCount*. These specify the total number of executions of a remote signature call and the level of parallelism. Both variables can be specified using parametric dependencies . Furthermore, a *ClusterResourceSpecification* is introduced to complement a *ResourceContainer*. A *ResourceContainer* may represent a physical or virtual machine that hosts resources (e.g., CPU). The *ClusterResourceSpecification* contains two variables to reference a *ResourceRole* and a *SchedulingPolicy*. A *ResourceRole* is used to describe whether a *ResourceContainer* represents a cluster, a master or a worker. A *SchedulingPolicy* is used to describe how actions are distributed on a cluster.

For simulating models, PCM applies model to text (M2T) transformations to generate code that is used by the simulation framework SimuCom [8]. We reuse existing Palladio concepts to implement the M2T transformation of the *DistributedCallAction* as the following algorithm demonstrates.

```
1:  $forks$ {array of length $simultaneousForkCount$}
2:  $actionsPerForkCount \leftarrow totalForkCount/simultaneousForkCount$
3:  for $i \leftarrow 0, simultaneousForkCount$ do
4:      $actions$ {array of length $actionsPerForkCount$}
5:      for $j \leftarrow 0, actionsPerForCountk$ do
6:          $actions[j] = createExternalCallAction$
7:      end for
8:      $forks[i] = actions$
9:  end for
10: return $forks$
```

PCM supports modeling parallel calls of signatures from required services by using an *ExternalCallAction* inside a so-called *ForkedBehavior*. First, we create an array *forks* of type *ForkedBehavior* with length *simultaneousForkCount*. The parallel actions (or calls) per fork (*actionsPerForkCount*) are calculated by dividing *totalForkCount* by *simultaneousForkCount*. We fill each index of the array $forks$ with an array called *actions*. This array consists of *ExternalCallActions* according to the number of *actionsPerForkCount*. For example, if *totalForkCount* equals eight and *simultaneousForkCount* equals two, there will be two *ForkedBehaviors* and each will contain four consecutive *ExternalCallActions*.

For the *ClusterResourceSpecification* and its components, we implemented corresponding Java classes in the scheduler and SimuCom plugin of PCM. We also adapted the existing implementation of a simulated resource container to apply the scheduling of calls (i.e., round robin) on nested resource containers.

B. Modeling Batch Applications

We compose our components in the repository model similar to the DAG of an application, in this case Spark's DAG. We specify one application component as a starting point. It includes input parameters for the number of files, the size of one file, the default block size, and the number of executors. We model job components according to the number of Spark jobs. They are invoked sequentially by the application component with the same input parameters. Similarly, we model stage components corresponding with the number of Spark stages for each job. They are called sequentially by each job component and also receive the same parameters.

For each stage, we model one associated task component that will be invoked multiple times in parallel for which we use a *DistributedCallAction* [29]. Therefore, we model the first stage and the number of task executions different from the remaining stages. For the first stage, the number of task executions depends on the number of RDD partitions since input files are read from the storage layer. Spark will create a RDD for each input file and each RDD involves as many partitions as data blocks and splits, respectively. We use the above mentioned input parameters to specify the number of tasks and blocks n_{block} as the sum of data blocks for all files. In order to calculate the blocks for one file, we divide the size of a file x_{file} by the default block size x_{block} and take the ceiling in case there is a remainder ($x \in \mathbb{N}_{>0}$).

$$n_{block} = \sum_{i=1}^{n_{files}} \lceil x_{i,file} \div x_{block} \rceil \tag{1}$$

Furthermore, the input size for tasks of the first stage either match the default block size of the storage layer or the remainder split. Therefore, we specify a branch to include both cases and determine the probability $p_{defaultSplit}$ for a default block by dividing the amount of default blocks by the total amount of blocks.

$$p_{defaultSplit} = \frac{\sum_{i=1}^{n_{files}} \lfloor x_{i,file} \div x_{block} \rfloor}{n_{block}} \quad (2)$$

In contrast to the first stage, the data input for subsequent stages equals the output set of predecessor stages. We model these as a percentage of the dataset of the predecessor. Consequently, we do not need to specify two branches unlike for the first stage, but can directly call the task component. Additionally, the total number of tasks for subsequent stages depends on a fixed value configured by application developers. Since the configurable number of cores constitutes the limiting factor for concurrent tasks on a Spark executor, we specify an additional infrastructure component to model a pool of available cores. The component contains two SEFFs to acquire as well as release one core. In order to finally execute a task, a core must be acquired first and released after the task execution. The SEFF of the task component includes two consecutive resource demands, one for delay and one for CPU. In this work, we do not consider HDD demands and concentrate on CPU.

In order to estimate the function of the CPU demand, we profiled applications using the Java Management Extension (JMX) on the Java virtual machine of each Spark executor. We aggregated the CPU measurements of operations originating from *org.apache.hadoop.net.unix.DomainSocketWatcher.-run* for each stage during the application lifetime. We divided the combined measurements by the number of tasks of each stage to get the intercept of the function. In the same way, we transformed measurements for operations called by *org.-apache.spark.scheduler.Task.run*. We additionally divided the latter value by the mean block size of the underlying dataset in order to the derive the slope of the function in dependence of the data size.

Regarding the delay demand, we used the Spark monitoring interface of the history server to calculate the mean response time of each task of a stage. We then subtracted the CPU demand per task to derive the delay. The monitoring interface also provides several metrics by itself. Although we also experimented to incorporate these metrics, the approach we described delivered more accurate prediction results for CPU and response time.

C. Modeling Stream Applications

The repository model for stream applications is also kept similar to the DAG of a Spark Streaming application as well as to our approach for batch applications. We model one application component as a starting point, which is intended to be triggered for each mini-batch interval. In contrast to the batch approach, we do not specify parameters in dependence on data sizes (i.e., megabytes), but in dependence on records. Therefore, the application component includes parameters for the number of records, the number of partitions of the data stream, and the number of executors. Since Spark Streaming creates one and the same job for each mini-batch, we create one job component. It is invoked by the application component using an asynchronous *ForkedBehavior*. In this way, the application component does not wait until the job component is finished and can be continuously triggered in time. According to the number of Spark stages, we model stage components that are called sequentially by the job component.

For each stage, we model one task component that will be invoked multiple times in parallel using a *Distributed-CallAction* [29]. Similar to the batch approach, we model the first stage and the number of task executions different from the remaining stages. The initial number of stream partitions defines the number of RDD partitions and, therefore, the number of task executions for the first stage. For subsequent stages, a fixed value is used as parameter since it can be configured by engineers in the application configuration. The record input for subsequent stages equals the output set of predecessor stages. As before, we model these as a percentage of the dataset of the predecessor. For all stages, the final task will be invoked after a core is acquired, which will be released afterwards. Therefore, we also specify an infrastructure component to acquire and release available cores. The SEFF of the task component includes one delay demand and one for CPU demand. Both demands are calculated as for the batch approach but in dependence of the number of records.

D. Modeling Cluster Resources

In the resource environment model, we specify one parent *ResourceContainer* and multiple nested *ResourceContainer* depending on the number of workers. All containers are connected to a network via a *LinkingResource*. For each *ResourceContainer*, we model a *ClusterResourceSpecification*. For the parent *ResourceContainer*, we set a *MASTER* role and a *ROUND_ROBIN* policy. For the nested *ResourceContainer*, we configured a *WORKER* role. Additionally, processing resources (e.g., CPU) are added for each nested container.

E. Modeling Data Workload

The data workload is modeled in the usage model. For batch applications, the application component is invoked with four parameters. They specify the number of files that shall be processed, the size of each file, the default block size of the storage layer, and the number of Spark executors. A closed workload is used without any think time and with a population of one since there shall only one application to be executed. For stream applications, the SEFF of the application component is called with three parameters describing the sum of records within the stream interval, the number of stream partitions, and the number of Spark executors. Since the amount of records usually deviates slightly (e.g., due to network circumstances), a normal distribution was used to address this factor. We also specify a closed workload with a population of one. However, the think time is used to represent the time of mini-batch intervals. The application component is continuously invoked after the think time has elapsed.

V. Evaluation

In order to evaluate our approach we used the HiBench benchmark suite[1] to run sample applications in a test environment [30]. Afterwards, we modeled and simulated these applications for selected scenarios and compared the measured and simulated response time and CPU utilization. We conducted four different scenarios - one upscaling scenario regarding cluster size and one upscaling scenario regarding data workload for batch as well as for stream processing each. In the subsequent Subsections, we describe our test environment setup, the evaluation of the batch scenarios followed by the stream scenarios.

A. Test Environment Setup

The hardware environment includes five IBM System X3755M3 servers, each consisting of four CPU sockets, 48 cores at 2.1 GHz in total, and 256 gigabyte (GB) random access memory (RAM). Each server is connected to a storage area network via Fibre Channel allowing for 8 gigabit per second (GBit/s). IBM System Storage EXP3512 is used for storing data. We virtualized each server using the VMware ESXi (5.1.0) hypervisor. We configured eight cores and 36 GB RAM for each virtualized machine (VM). On four servers, we allocated four VMs each that are used as worker nodes. On the remaining server, we allocated two VMs. One is used as master node and one for managing the cluster and initiating the benchmark applications. The following software is used on the VMs:

- CentOS Linux, 7.2.1511
- Oracle JDK, 1.8.0_60
- Apache Ambari, 2.4.2.0
- Hortonworks Data Platform, 2.5.3.0-37
- HiBench Suite, 6.0

Regarding HDFS we kept the default configurations including a replication factor of three and a data block size of 128 megabytes (MB). For YARN, we configured 26 GB and six virtual cores (vCores) per container, for Spark executors 22 GB as well as six cores.

B. Evaluating Batch Applications

We used the word count application of HiBench. It parses a set of input data and counts the appearance of each word [30]. We conducted four upscaling experiments regarding cluster nodes and, similarly, four regarding data workload. Therefore, we created one base repository model for the application. This model including its resource demands is used for all batch experiments. According to each experiment, the resource environment model and the usage model is adjusted. In order to evaluate the prediction accuracy of our approach, we consider the metrics response time and CPU utilization. For the simulation, we captured the simulated mean response time (SMRT) as well as the simulated mean CPU utilization (SMCPU) across the cluster. For the benchmark measurements, the applications were executed three times for each experiment and the measured mean response time (MMRT) as well as the measured mean CPU utilization (MMCPU) on user-level were calculated. The former is derived from the Spark monitoring API, the latter from the Ambari Metrics System. The results for all batch experiments are listed in Table I and the corresponding response times are illustrated in Figure 1.

[1] https://github.com/intel-hadoop/HiBench

TABLE I
Measurement and simulation results for batch applications

Cluster nodes	**Workload** [gigabyte]	**Response time** [milliseconds] MMRT	SMRT	RTPE	**CPU utilization** MMCPU	SMCPU	CPUPE
4	28.72	104,599	103,218	1.32%	61.24%	61.95%	1.16%
8	28.72	68,205	62,233	8.76%	53.06%	55.77%	5.10%
12	28.72	54,984	52,632	4.28%	47.21%	47.5%	0.62%
16	28.72	50,140	47,181	5.90%	40.31%	42.18%	4.65%
16	57.36	74,657	69,583	6.80%	48.87%	48.22%	1.33%
16	86.08	101,675	93,292	8.24%	51.11%	49.66%	2.84%
16	114.72	119,977	122,740	2.30%	55.62%	52.07%	6.38%

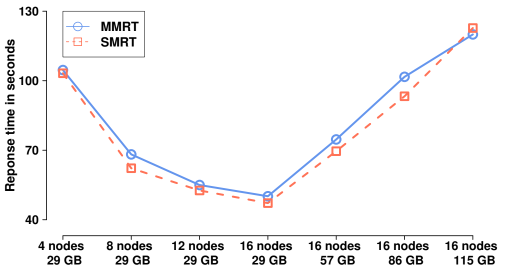


Fig. 1. Response times for batch applications

The starting experiment with four nodes resulted in a MMRT of 104,599 milliseconds (ms) and a SMRT of 103,218 ms leading to a relative response time prediction error (RTPE) of 1.32%. The MMCPU amounts to 61.24%, whereas the SMCPU lies at 61.95%, which gives a relative CPU utilization prediction error (CPUPE) of 1.16%. We resized the amount of nodes up to 16 nodes. Throughout, RTPE and CPUPE remained relatively low and were at most 8.76% and 5.10% both for the eight node scenario. On a cluster of 16 nodes, we additional increased the data workload. Similarly, we approximately resized the dataset by factors two, three, and four. Here, the RTPE was highest for a workload of 86.08 GB (8.24%), while the CPUPE deviated at most for 114.72 GB (6.38%).

Our approach showed to deliver relative prediction errors no more than 10% for both batch scenarios. While we slightly overestimated the CPU utilization values in the first scenario, we slightly underestimated them for the second scenario. Regarding response time, the prediction values were little lower than the measurement values in both scenarios except for one case.

C. Evaluating Stream Applications

We likewise used a word count application from the HiBench benchmark suite. The application involves stateful operators as well as checkpoints and acknowledgements. The application repeatedly fetches data from Kafka in a configured time interval of five seconds. We configured the environment so Kafka as well as Spark run exclusively on VMs. During the experiments, we adapted the number of Kafka brokers according to the number of Spark workers. Similarly, we adapted the number of Kafka partitions to always have six partitions on each Kafka broker to allow for optimal parallel processing of Spark (i.e., one partition per core [31]). For our experimental results, we run the benchmark application and captured performance measurements for ten minutes leaving a ramp-up and ramp-down phase of five minutes.

We conducted four upscaling experiments regarding the cluster and four regarding data workload. Similar to the batch experiments, we derived a repository model including resource demands from the starting experiment and used this model for all simulations. The results are illustrated in Table II and the response times in Figure 2. For an interval of five second, the MMRT for the starting experiment is 3,006 ms and 31.81% MMCPU. The SMRT resulted in 3,029 ms and the SMCPU in 33.29%, which gives a RTPE of 1.21% and a CPUPE of 4.65%. With additional nodes, the relative prediction errors increased for both metrics and were at most for eight nodes (a scaling factor of four compared to the starting experiment). Here, the RTPE results in 17.02% and the CPUPE 13.43%. With increasing data workload, the RTPE and CPUPE decreased. In these experiments, we were not able to scale the workload by factor four since the input data could not be processed within the five second interval. While the SMCPU is constantly slightly lower than the MMCPU and the CPUPE behaves consistently, the SMRT is slightly too low for the last experiment as the MMRT increases abruptly. We conducted multiple experiments to further investigate the response time behavior of the application. We observed that the response times tend to rise rapidly as they converge to the time interval. Our prediction results still showed to provide accurate results with relative errors around 17% [32].

Our extension[2] as well as our models, simulation results, and measurements results are publicly available online[3].

TABLE II
MEASUREMENT AND SIMULATION RESULTS FOR STREAM APPLICATIONS

Cluster nodes	**Workload** [events/second]	**Response time** [milliseconds] MMRT	SMRT	RTPE	**CPU utilization** MMCPU	SMCPU	CPUPE
2	100,000	3,066	3,029	1.21%	31.81%	33.29%	4.65%
4	100,000	2,363	2,515	6.43%	20.89%	19.91%	4.69%
6	100,000	2,124	2,358	11.02%	17.02%	15.44%	9.28%
8	100,000	1,956	2,289	17.02%	15.26%	13.21%	13.43%
8	150,000	2,754	2,820	2.40%	17.55%	16.16%	7.92%
8	200,000	3,296	3,350	1.64%	20.43%	19.10%	6.51%
8	250,000	4,614	3,880	15.91%	22.79%	22.04%	3.29%

[2]http://git.fortiss.org/pmwt/bd.pcm.extension
[3]http://pmw.fortiss.org/research/ieee-mascots/

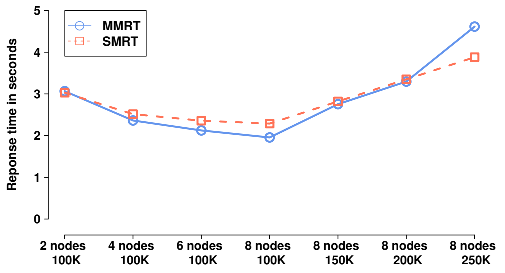


Fig. 2. Response times for stream applications

D. Assumptions and Limitations

In our experiments, we allocated one Spark executor on each node. It is also possible to size less cores and memory for spark executors and to allow for deploying multiple ones on one node. Although we are also able to model and simulate these scenarios, we did not evaluate such a case. We also evaluated our experiments in an exclusive cluster in which no other applications were running in parallel and using any CPU, HDD, or network. Regarding our modeling approach, we specified the input of a subsequent Spark stage probabilistically in dependence on the output data of a previous stage. Therefore, our prediction error will increase, if the properties of the initial underlying data set change significantly. Furthermore, Heinrich et al. (2016) [33] discuss current problems such as modeling data structures and continuous data flows, but also potential solutions in modeling big data using Palladio.

VI. Conclusion and Future Work

In this work, we presented an approach to model and simulate the performance behavior of batch as well as stream processing systems by the example of Apache Spark. Therefore, we extended PCM to represent resource clusters and distributed and parallel operations. This included a M2T transformation to generate corresponding simulation code and an adaption of the simulation platform SimuCom. We evaluated the approach by using sample applications of the HiBench benchmark suite. We conducted upscaling scenarios for cluster sizes as well as data workload both by factor four. Afterwards, we compared simulation with measurements values. The results suggest accurate predictions for response times and CPU utilization. For batch applications, the relative prediction error was at most 8.76% for response time and 6.38% for CPU utilization, for stream applications 17.03% and 13.43%.

Currently, we are experimenting with applying our approach for stream processing systems that implement an operator-based model. We also intend to represent resource demands for HDD. Furthermore, we plan to automatically derive models and resource demands for big data applications based on measurements. This shall support performance engineers by omitting the manual creation of models and ease the usage of our approach.

References

[1] M. Schermann, H. Hemsen, C. Buchmüller, T. Bitter, H. Krcmar, V. Markl, and T. Hoeren, "Big data - an interdisciplinary opportunity for information systems research," *Business & Information Systems Engineering*, vol. 6, no. 5, pp. 261–266, 2014.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] R. Casado and M. Younas, "Emerging trends and technologies in big data processing," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 8, pp. 2078–2091, 2015.

[4] A. Brunnert, C. Vögele, A. Danciu, M. Pfaff, M. Mayer, and H. Krcmar, "Performance management work," *Business & Information Systems Engineering*, vol. 6, no. 3, pp. 177–179, 2014.

[5] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *Proceedings of the 17th International Conference on High Performance Computing and Communications*. IEEE, Aug 2015, pp. 166–173.

[6] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, J. Kroß, S. Spinner, C. Vögele, J. Walter, and A. Wert, "Performance-oriented DevOps: A research agenda," SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), Tech. Rep. SPEC-RG-2015-01, Aug. 2015.

[7] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev, "Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures," *IEEE Transactions on Software Engineering*, vol. 41, no. 2, pp. 157–175, 2015.

[8] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *The Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[9] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, and U. Dayal, "Analytical performance models for MapReduce workloads," *International Journal of Parallel Programming*, vol. 41, no. 4, pp. 495–525, 2013.

[10] A. Verma, L. Cherkasova, and R. H. Campbell, "Profiling and evaluating hardware choices for MapReduce environments: An application-aware approach," *Performance Evaluation*, vol. 79, pp. 328 – 344, 2014.

[11] Z. Zhang, L. Cherkasova, and B. T. Loo, "Benchmarking approach for designing a mapreduce performance model," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 2013, pp. 253–258.

[12] E. Barbierato, M. Gribaudo, and M. Iacono, "Performance evaluation of NoSQL big-data applications using multi-formalism models," *Future Generation Computer Systems*, vol. 37, no. 0, pp. 345 – 353, 2014.

[13] D. Ardagna, S. Bernardi, E. Gianniti, S. Karimian Aliabadi, D. Perez-Palacin, and J. I. Requeno, *Modeling Performance of Hadoop Applications: A Journey from Queueing Networks to Stochastic Well Formed Nets*. Springer International Publishing, 2016, pp. 599–613.

[14] S. Lehrig, "Applying architectural templates for design-time scalability and elasticity analyses of saas applications," in *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability*. ACM, 2014, pp. 2:1–2:8.

[15] A. Castiglione, M. Gribaudo, M. Iacono, and F. Palmieri, "Modeling performances of concurrent big data applications," *Software: Practice and Experience*, 2014.

[16] R. Niemann, "Towards the prediction of the performance and energy efficiency of distributed data management systems," in *Proceedings of ACM/SPEC International Conference on Performance Engineering*. ACM, 2016, pp. 23–28.

[17] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. D. Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. F. Pérez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladušič, "Dice: Quality-driven development of data-intensive cloud applications," in *Proceedings of the Seventh International Workshop on Modeling in Software Engineering*. IEEE, 2015, pp. 78–83.

[18] M. Guerriero, S. Tajfar, D. A. Tamburri, and E. Di Nitto, "Towards a model-driven design tool for big data architectures," in *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*. ACM, 2016, pp. 37–43.

[19] A. Gómez, J. Merseguer, E. Di Nitto, and D. A. Tamburri, "Towards a uml profile for data intensive applications," in *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*. ACM, 2016, pp. 18–23.

[20] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and B. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA: USENIX Association, 2016, pp. 363–378.

[21] R. Ginis and R. E. Strom, "Method for predicting performance of distributed stream processing systems," October 2010, uS Patent 7,818,417. [Online]. Available: https://www.google.com/patents/US7818417

[22] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[24] Apache Spark, "Lightning-fast cluster computing," 2017, accessed: 2017-04-01. [Online]. Available: https://spark.apache.org

[25] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *Proceedings of the 21st IEEE International Conference on Parallel and Distributed Systems*. IEEE, 2015, pp. 797–802.

[26] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: A fault-tolerant model for scalable stream processing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-259, 2012.

[27] J. Kroß, A. Brunnert, C. Prehofer, T. Runkler, and H. Krcmar, "Stream processing on demand for lambda architectures," in *Computer Performance Engineering*, ser. Lecture Notes in Computer Science, M. Beltrán, W. Knottenbelt, and J. Bradley, Eds. Springer International Publishing, 2015, vol. 9272, pp. 243–257.

[28] J. Kroß and H. Krcmar, "Modeling and simulating Apache Spark Streaming applications," *Softwaretechnik-Trends*, vol. 36, no. 4, 2016.

[29] J. Kroß, A. Brunnert, and H. Krcmar, "Modeling big data systems by extending the Palladio component model," *Softwaretechnik-Trends*, vol. 35, no. 3, 2015.

[30] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Proceedings of the 26th International Conference on Data Engineering Workshops*. IEEE, 2010, pp. 41–51.

[31] O. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández, "Spark versus Flink: Understanding performance in big data analytics frameworks," in *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 2016, pp. 433–442.

[32] D. A. Menascė and V. A. F. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*. Upper Saddle River, New Jersey: Prentice Hall, 2002.

[33] R. Heinrich, H. Eichelberger, and K. Schmid, "Performance modeling in the age of big data - some reflections on current limitations," in *Proceedings of the 3rd International Workshop on Interplay of Model-Driven and Component-Based Software Engineering*, October 2016, pp. 37–38.

big data and cognitive computing

Article

PerTract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop

Johannes Kroß [1,*] and Helmut Krcmar [2]

1 fortiss, Research Institute of the Free State of Bavaria, Guerickestr. 25, 80805 Munich, Germany
2 Chair for Information Systems, Technical University of Munich (TUM), Boltzmannstr. 3, 85748 Garching, Germany
* Correspondence: kross@fortiss.org; Tel.: +49-89-360-352-218

Received: 13 July 2019; Accepted: 6 August 2019 ; Published: 9 August 2019

Abstract: Evaluating and predicting the performance of big data applications are required to efficiently size capacities and manage operations. Gaining profound insights into the system architecture, dependencies of components, resource demands, and configurations cause difficulties to engineers. To address these challenges, this paper presents an approach to automatically extract and transform system specifications to predict the performance of applications. It consists of three components. First, a system- and tool-agnostic domain-specific language (DSL) allows the modeling of performance-relevant factors of big data applications, computing resources, and data workload. Second, DSL instances are automatically extracted from monitored measurements of Apache Spark and Apache Hadoop (i.e., YARN and HDFS) systems. Third, these instances are transformed to model- and simulation-based performance evaluation tools to allow predictions. By adapting DSL instances, our approach enables engineers to predict the performance of applications for different scenarios such as changing data input and resources. We evaluate our approach by predicting the performance of linear regression and random forest applications of the HiBench benchmark suite. Simulation results of adjusted DSL instances compared to measurement results show accurate predictions errors below 15% based upon averages for response times and resource utilization.

Keywords: peformance evaluation; performance modeling; model extraction; performance simulation; big data systems

1. Introduction

Big data frameworks are specialized to analyze data with high volume, variety, and velocity efficiently [1]. By distributing and parallelizing processing, they allow for horizontal scalability. Since the introduction of the MapReduce paradigm, there have been several frameworks released to support different types of applications, such as machine learning and stream processing. For all types, the performance of such software systems in terms of response time, throughput, and resource utilization is essential for a successful application [2]. It is a difficult and complex task to manage and evaluate the performance for different scenarios such as changing data input and hardware resources [3].

Practical evaluations such as load tests on test systems are expensive. They require multiple experiments and only test a subset of configuration parameters. Additionally, they usually run with a reduced amount of data and resources. Thus, it is not able to draw accurate conclusions about the performance behavior. Performance models, on the other hand, provide an established evaluation approach by depicting performance characteristics of software systems and simulating their behavior or analytically solving them [4]. However, there are several challenges: creating models by hand is expensive, error-prone and slow as software systems are complex and continuously evolve [5]. There is a lack of tool support for automatic model extraction. Regarding big data system, most related modeling approaches are also specific to a certain technology (i.e., Apache MapReduce) and only consider the response time of applications but not demands for resources (i.e., CPU).

In order to address these challenges, we propose a specification and model extraction approach for big data systems called PerTract to evaluate and predict the performance. We present a domain-specific language (DSL) to allow for modeling specifications on an architecture-level in a tool-agnostic way. To demonstrate our approach, we use Apache Spark for the application layer, in particular one random forest and one linear regression application that both use Spark's machine learning library. Additionally, we use Apache Hadoop for data provisioning and resource management. Figure 1 illustrates an overview of our approach. We extract execution components and inter-component interactions, resource landscape, and data workload in three separated specifications of a DSL instance using interfaces and logs of these technologies. In addition, we extract monitoring traces of applications (i.e., CPU times) and interrelate these with data workload information to identify parametric dependencies and estimate parametric resource demands of each execution component. On this basis, performance predictions are enabled. Therefore, we transform a DSL instance into a Palladio component model (PCM) [6]. Palladio is a model-based performance evaluation tool on the architecture-level that is supported by several analytical solvers and simulation engines.

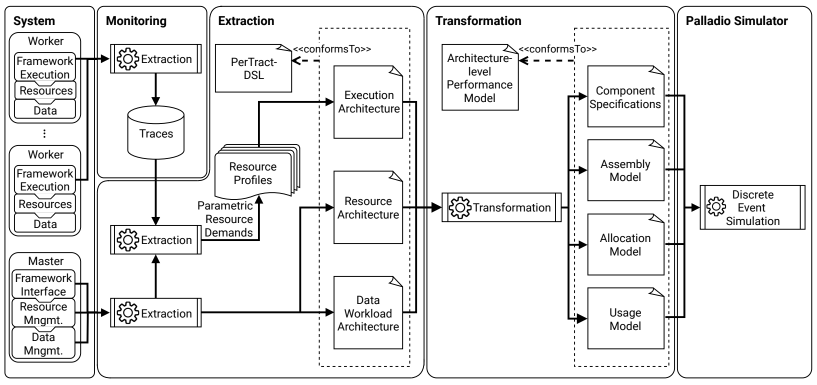


Figure 1. Overview of the extraction and transformation approach.

Our approach provides several benefits. It integrates model-based activities, which are performed during development, and measurement-based activities, which are carried out during operations (DevOps) [5]. The automated extraction process eliminates the effort to create models by hand. As applications are continuously updated, DSL instances can be extracted and tracked for each release as they evolve as well. This also enables engineers to continuously manage and plan required capacities and evaluate the performance for different scenarios (e.g., changing data workload) by adapting model parameters. Finally, it gives detailed insights about resource demands of execution components of an application and can be used to detect performance changes and regressions.

To sum up, the contributions of this paper are the following:

1. A DSL for modeling performance-relevant factors of big data systems,
2. An automatic extraction of system structure, behavior, resource demands, and data workload from Apache Spark and Apache Hadoop,
3. Transformations from DSL instances to model- and simulation-based performance evaluation tools,
4. Tool support for this approach.

To the best of our knowledge, our approach is the first white-box approach to extract performance-relevant metrics that allow for performance predictions of response times and resource usage. The developed tools are open source [7] and extendable for extracting DSL instances from other frameworks and for transforming them to other model-based performance evaluation tools.

This paper builds upon our previous work [8–11] on modeling and simulating the performance of big data applications and includes the following major improvements and extensions:

1. A formalism and DSL to model big data applications,
2. A lightweight Java agent to sample stack traces and CPU times from applications,
3. Automatic extraction of DSL instances,
4. Detailed evaluation against complex applications of the HiBench benchmark suite.

The remainder of this work is structured as follows: Section 2 describes related literature and approaches in the area of modeling and simulating big data applications. Section 3 introduces the model formalism as well as the DSL, which are required to understand this paper. Section 4 describes the extraction of DSL instances by the example of Apache Spark and Apache Hadoop. Section 5 presents the transformation to PCM models to allow for simulating the performance. Section 6 evaluates the prediction accuracy of our proposed approach for different upscaling scenarios and describes our assumptions and limitations. Finally, Section 7 outlines conclusions of our work and ideas for future activities.

2. Related Work

Since the Apache Hadoop family was the first widely-adopted big data framework, initial performance modeling approaches have been concentrating on this technology stack.

Vianna et al. [12] predict the response time of MapReduce applications by introducing an analytical model, which they validated against an event-driven queuing network simulator. Their approach primarily concentrated on synchronization delays between map and reduce tasks. Verma et al. [13] introduce another approach for MapReduce. They developed a framework to allow for predicting response times before moving applications to different target platforms. The framework applies multiple benchmarks on source platforms and a regression-based model to relate the performance of source and the target platforms. Zhang et al. [14–16] present multiple approaches where most of them are based on the analytical model by [13]. Therefore, they additionally take heterogeneous clusters and configuration optimizations into account.

For other applications of the Hadoop family, Barbierato et al. [17] developed a language for the description of performance models. As a main component, the model uses the SQL-like query language of Apache Hive, a data warehouse built on top of Apache Hadoop. Ardagna et al. [18] propose approaches to estimate response times of Hive requirements. Therefore, they presented multiple performance analysis models with increasing complexity and accuracy, such as queueing networks and stochastic well formed nets. They also considered unreliable resources in their experiments. Lehrig [19] proposes a scalability and elasticity analysis of Software-as-a-Service applications at design time using architectural templates for Palladio. They plan to enhance it for big data paradigms on the processing layer and data layer.

Wang and Khan [3] propose a prediction model for estimating response times of Apache Spark applications. In their approach, they consider demands for in-memory as well as demands for disk drives but not CPU processing. Another work by Ardagna et al. [20] explores three modeling approaches for execution time prediction of Spark applications: one queuing network with a fork-join model and one with a task precedence model. Third, they present a discrete event simulation engine dagSim. The evaluation was conducted for different applications such as logistic regression and K-Means running in a public cloud. Although the variance of the prediction accuracy is low for all approaches, the third approach delivers the most precise results.

Besides analytical and simulation-driven approaches, there are also approaches using machine learning for Apache Spark. Rekha and Praveen [21] evaluated different machine learning algorithms (i.e., multi linear regression and support vector machine) as well as an analytical model to predict execution times of Spark stages in development environments. They include multiple parameters from application logs into their models but only use execution times and do not consider resource demands. They also mention the drawback of machine learning approaches, which require intensive experiments and data collection. Furthermore, Venkataraman et al. [22] present Ernest, a performance prediction framework for large scale analytics using machine learning kernels. It involves an automatic process to collect training data and to build a non-negative least squared model taking only a few parameters. They evaluate their approach on Amazon EC2 and show accurate predictions of execution times for increasing machine numbers. It is a black-box approach and does not give any insight into components of an application. As Ernest is bound to the structure of machine learning jobs, Alipourfard et al. [23] present CherryPick, which intends to find best cloud configurations for various applications and use Bayesian optimization to create performance models. A configuration, for instance, contains parameters such as the number of virtual machines, CPU, and cores. In contrast to our work, they support additional types of applications (i.e., Spark SQL). Additionally, Witt et al. [24] provide an extensive survey on performance prediction of batch processing using black box monitoring and machine learning.

Castiglione et al. [25] propose a general approach to model the behavior of batch applications and concentrate on cloud infrastructures and evolution dynamics in terms of resource requirements and energy consumption. Therefore, they use an analytic modeling technique based Markovian agents and mean field analysis to describe the behavior of interactive cloud, batch, and time constrained applications. Niemann [26] also presents an approach in the area of energy consumption. He focuses on Apache Cassandra, a distributed data management system, and uses queueing Petri nets to predict the performance and energy consumption of different workloads and platforms. Casale et al. [27] propose a model-driven engineering for quality assurance of data-intensive software systems concentrating on Apache Hadoop and MapReduce, NoSQL databases, and stream processing (i.e., Apache Storm). Their approach aims at simulating, verifying, and optimizing architectures of big data applications. The models contain three different model layers including a platform-independent, a technology-specific and a deployment-specific model [28]. Gómez et al. [29] also shows an approach to transform these models into stochastic Petri nets, which is intended to allow for evaluating performance requirements. Lastly, Ginis and Strom [30] hold a patent in the area of stream processing. The patent describes a method to model performance characteristics of publish–subscribe systems using queueing theory. However, the method does not include resource demands such as CPU, memory, and disks.

To summarize, the mentioned approaches focus on predicting the metric response time and often only implicitly assume resource demands for service executions per resource but do not link them to software components and operations [5]. To the best of our knowledge, automatic model extraction in the area of big data are only supported by the mentioned machine learning approaches [22,23]. However, these are black-box approaches and the models serve as interpolation of the measurements [5]. Consequently, they do not model detailed information of the system architecture and dependencies and cannot be adapted for further evaluation scenarios. Finally, most of the mentioned models are technology-specific and, thus, are difficult to adapt and generalize them.

3. Modeling Approach

In this section, we describe the formalism for specifying big data systems. Afterwards, we present the PerTract-DSL based on the formalism.

3.1. Formalism

The specification consists of the following components:

- An *Execution Architecture* of the application, specifying nested directed graphs for execution components,
- A set of *Resource Profiles*, providing demands of different resources with parametric dependencies for the nodes of a graph,
- A *Data Workload Architecture*, specifying the underlying data model and type of data source
- A *Resource Architecture*, specifying a cluster of resource nodes, each with several resource units

3.1.1. Application Execution Architecture

The specification of the application Execution Architecture is a 2-tuple (c, n) where $c \in C$ is the application configuration and $n \in N$ specifies an initial node of the application.

A configuration $c \in C$ is represented by the 5-tuple $(p_d, e, ts_e, m_e, m_{ts})$ where p_d is the default parallelism for operations, more specifically tasks, of an application (e.g., join or reduce); e is the number of executors, which manage tasks; ts_e describes the number of tasks slots per executor that can be executed in parallel; m_e is the amount of main memory per executor that is available for tasks; and m_{ts} represents the amount of memory that each task slot requires to be allocated.

Nodes N are composite components. They can represent directed graphs $NG \subset N$ and execution nodes of a directed graph $NE \subset N$. In Figure 2, *ScalaWordCount* and *saveAsHadoopFile* represent a directed graph and *map* and *reduce* an execution node.

A directed graph $ng \in NG$ is a 2-tuple (N_{ng}, E_{ng}), in which N_{ng} is a set of nodes (or vertices) of the directed graph ng such that $ng \notin N_{ng}$; and E_{ng} is a set of directed edges. A directed edge $e \in E$ is represented by a 3-tuple (n_t, n_h, t_e), where $n_t \in N$ is the tail of e; $n_h \in N$ is the head of e; and $t_e \in \mathbb{R}_{\geq 0}$ specifies the factor of how many data are transmitted from n_t to n_h dependent on the amount of input data of n_t.

An execution node $ne \in NE$ is a 5-tuple (p_n, s, m, n_{ng}, rp) where p_n is the parallelism of node (e.g., some big data frameworks such as Apache Flink allow for specifying the parallelism for each operation individually); s indicates whether ne is a spout that is the node depending on partitioned data from an external source, such as a file system or messaging system; $m \in M$ is a reference to the dependent data model from the Data Workload Architecture; $n_{ng} \in NG$ references the parent directed node graph; and $rp \in RP$ describes the Resource Profile of ne.

3.1.2. Resource Profile

We use Resource Profiles to specify multiple resource demands. A Resource Profile $rp \in RP$ describes an ordered set of parametric resource demands RD. A parametric resource demand $rd \in RD$ is a 3-tuple (rt, f_{rt}, p) in which $rt \in RT$ represents the resource type and $f_{rt} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ is a function to specify the actual value of a resource demand in dependence on a parameter p (e.g., number of partitions of an input data source).

3.1.3. Data Workload Architecture

The model to represent the data workload is kept very simple. A Data Workload Architecture $d \in D$ is a singleton containing a set of data models M. A data model $m \in M$ contains one data source $ds \in DS$ element that consists of a parameter p_{ds} to specify the number of partitions.

3.1.4. Resource Architecture

A Resource Architecture $ra \in RA$ is a pair (nc, RN) in which $nc \in NC$ is a network channel and RN is a set of resource nodes. A network channel $nc \in NC$ is a 2-tuple (b, l) where b describes its bandwidth and l its latency. A resource node $rn \in RN$ describes a cluster node and is a 2-tuple (cs, RU) in which $cs \in CS$ is a cluster specification and RU is a set of resource units. A cluster specification

$cs \in CS$ is described by a 2-tuple (rr, sp) where $rr \in RR$ describes a resource role (i.e., master node or worker node) and $sp \in SP$ the scheduling policy for distributing task across resource nodes (i.e., round robin). A resource unit $ru \in RU$ represents CPU, drive, and memory units.

3.2. *PerTract*-DSL

The PerTract-DSL follows the system model formalism described in the previous subsection and constitutes a language for specifying such models. Figure 2 illustrates an exemplary PerTract-DSL instance for a big data application. The PerTract-DSL is implemented as an Ecore-based meta-model using the Eclipse Modeling Framework (EMF) [31]. We use the DSL as an intermediate language to extract model instances and adapt its parameters for different scenarios. Afterwards, we generate architecture-level performance models that we use to simulate and predict the performance.

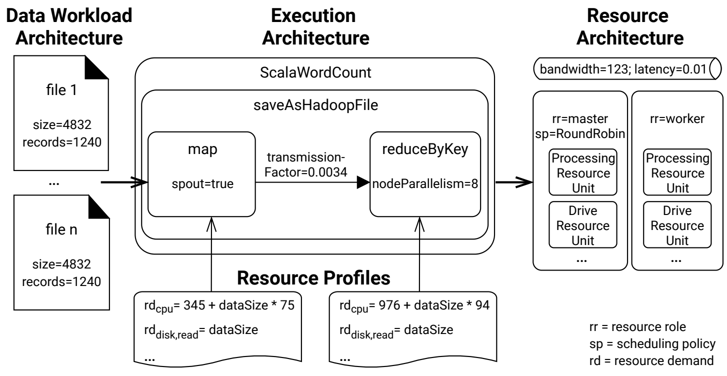


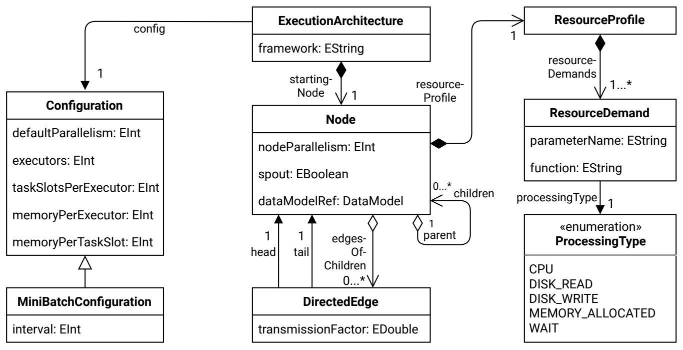
Figure 2. Exemplary PerTract-DSL instance.

Figure 3a shows the classes and relationships of the Execution Architecture and Resource Profile. The Execution Architecture includes execution flows and operations on data and a configuration of an application. The configuration includes multiple parameters to specify the application settings. Depending on the application type (i.e., batch, mini-batch, and stream), a corresponding configuration type can be instantiated and may include additional parameters. For instance, a *MiniBatchConfiguration* involves an interval variable to indicate the mini-batch intervals.

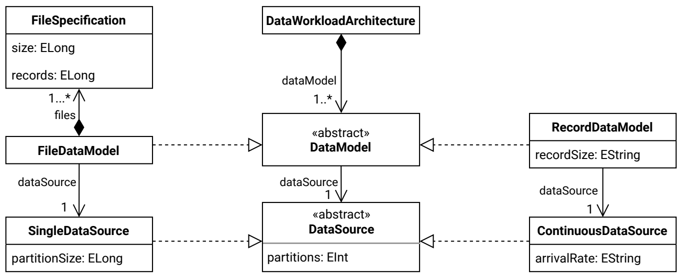
In order to specify operations on data and execution flows, we use nodes and directed edges (for instance, distributed acyclic graphs *DAGs* represent execution flows in Apache Spark, *topologies* in Apache Storm, and *job graphs* and *execution graphs* in Apache Flink). Therefore, a *Node* is a composite that can represent two roles—a directed graph that contains several nodes (*children*) and edges, and an execution node that executes tasks. In the latter case, a node contains a Resource Profile for its tasks.

The term Resource Profile describes a set of resource demands for transactions of an application [32–34]. This includes resource demands for CPU, disk, memory, and network usage. Resource Profiles have been used for transactions for a specific workload and specific servers [32,33] but also for component operations within the control flow of each transaction independent of their deployment topology [34]. Branches with probabilities for its occurrences represent operation control flows. As yet, the related approaches do not use parametric dependencies and use Resource Profiles in the area of enterprise applications, where the workload is mainly user-driven and the resource demands for operations may remain static for each user. In our case, operations highly depend on incoming data volume either dependent on the data size or number of records.

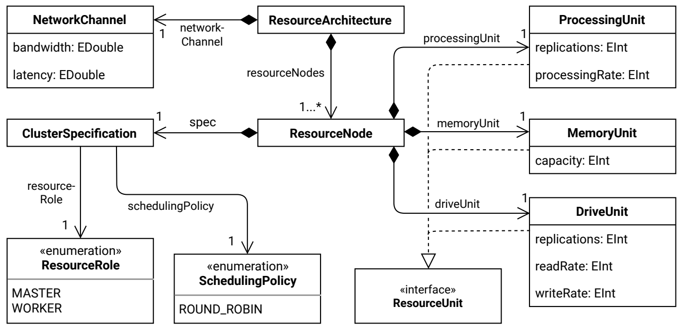
We change the notion of Resource Profiles for our purposes in three ways. First, we include parametric dependencies. Second, we do not model the control flow and probability as this information is contained in the directed graph. Third, we do not apply a Resource Profile on the same fine granularity level of operations except for a set of operations and tasks. Big data frameworks chain and group single operations together and transform each grouping into a set of tasks, which will eventually be executed multiple times in a distributed way. The number of executed tasks usually depends on the number of partitions. As we model data and hardware resources as first-class entities in dedicated specifications, the exact number and distribution of operations depends on them. Therefore, we apply a Resource Profile on a group of chained operations. It forms the basis to derive tasks with resource demands and predict the performance by combining them with data workload and Resource Architectures.



(**a**) Execution Architecture and Resource Profile



(**b**) Data Workload Architecture



(**c**) Resource Architecture

Figure 3. PerTract-DSL classes and relationships.

While considering data as first-class entities, we focus on specifying only performance-relevant factors of data as presented in Figure 3b. A Data Workload Architecture contains one or several data models, which are either file-based (e.g., for batch applications) or record-based (e.g., for stream applications). The former contains multiple file specifications and a single data source, which specifies the partition size of the files and the number of partitions. The latter contains a variable to indicate the mean record size and a continuous data source, which describes the number of partitions of a data stream as well as the arrival rate per second of one record.

Figure 3c illustrates an overview of the classes and relationships of the Resource Architecture. It is a simplified version based on the resource environment model of PCM including our extension [6,9]. It contains several resource nodes that, combined, represent a cluster. Each resource node contains a processing unit, memory unit, and drive unit with individual processing rates or capacities. The resource demands of one Resource Profile will be performed on the corresponding resource units of one resource node.

4. Extracting Model Instances by the Example of Apache Spark, Apache YARN and Apache HDFS

Since creating models for applications, data, and resources requires much effort, we propose an approach to automatically extract PerTract-DSL instances based on monitoring measurements and logs. The remainder of this section describes the approach to extract a DSL instance in detail, comprising the monitoring on application level (Section 4.1), the extraction of Execution Architectures from applications (Section 4.2), the estimation of Resource Profiles for stages of applications (Section 4.3), the derivation of Data Workload Architectures (Section 4.4), and the extraction of hardware resources (Section 4.5).

4.1. Extraction of Resource Demands

Collecting measurement data is necessary in order to extract Resource Profiles, estimate resource demands, and calculate parametric dependencies. Profilers provide a common way to extract fine-grained data such as stack traces and CPU times. We examined multiple Java profilers but found that the performance of big data applications is significantly increased by their overhead. Therefore, we chose a sampling approach and developed a lightweight Java agent for sampling CPU values for either stack traces or thread groups of long-running applications.

Algorithm 1 shows the main procedure of the agent. It collects samples in intervals of 100 milliseconds, which we found to cause only low overhead while still providing high accuracy in our experiments. Therefore, the agent fetches a dictionary of thread identifiers and corresponding stack traces by calling the *getAllStackTraces()* method provided by the Java *Thread* class. The dictionary contains only entries for threads that are in an active state at the point of time requested. The CPU time is collected for each thread by using the ThreadMXBean management interface (i.e., the *getThreadCpuTime(long id)* method) for monitoring of the Java Virtual Machine (JVM). The CPU times for thread groups with the same names will be summed up and sent as a batch to an Apache Cassandra repository. Additionally, the name of the JVM will be transmitted to the repository for each measurement.

Algorithm 1: Sampling thread groups and CPU values.

```
Output: $samples \leftarrow$ dictionary containing a timestamp as key and tuples of thread groups and
        CPU times as value
Schedule new thread every 100 milliseconds
    $threadGroups \leftarrow < k : String, v : long >$;
    $sampleTime \leftarrow$ current timestamp;
    /* procedure provided by Java                                  */
    $threads \leftarrow$ getAllStackTraces();
    for $thread$ to $threads$ do
        /* procedure provided by Java                              */
        $cpuTime \leftarrow$ getThreadCpuTime($thread.id$);
        $threadGroup \leftarrow thread.threadGroup$;
        $threadGroups[threadGroup] \leftarrow cpuTime + threadGroups[threadGroup]$);
    end
    $samples \leftarrow (sampleTime, threadGroups)$;
Until $application has terminated$;
```

4.2. Extraction of Execution Architectures

The Apache Spark framework introduces so-called resilient distributed datasets (RDDs). RDDs are parallel data structures to store intermediate results in memory and offer coarse-grained operations, which can be applied on them and work the same way on all data items [35]. Spark offers several operations and transformations such as *map* and *reduce*.

A Spark application is executed by forming a DAG based on associated operations and grouping them into stages of tasks. A stage chains operations with narrow dependencies, which means a shuffle operation is not required e.g., a *map* and a subsequent *filter* operation [35]. The number of tasks of one stage depends on the number of RDD partitions. Stages are executed successively and constitute one job. One or more jobs compose one Spark application. The application is managed by one context. It runs in the main process called the driver program. It allocates executors to worker nodes and schedules and assigns tasks of an application on to executors. An executor is a process that executes the tasks and operations in parallel [36].

In order to automatically extract execution components and inter-component interactions from Apache Spark, we access the interfaces of the embedded history server. We remind readers that we refer to the specification introduced in Section 3.1. We use the Spark environment properties to derive an Application Configuration. We set p_d to *spark.default.parallelism*, e to *spark.executor.instances*, ts_e to *spark.executor.cores*, and m_e to *spark.executor.memory*. While a DAG created by Apache Spark models RDDs as nodes and operations as edges, we create nodes on three levels—on application-, job- and stage-level—and data flows as edges (similar to the JobGraph of Apache Flink).

On the application-level, one initial node is created to represent the application itself (i.e., *ScalaWordCount* in Figure 2). It contains a set of child nodes and edges for the job-level.

On the job-level, we read the interface for job metrics of the corresponding application and create a set of nodes containing one element for each job entry. As jobs may be executed in parallel, we consider the chronological sequence of jobs by accessing start times and end times in order to create a set of directed edges and connect successive nodes. The data transmission factor of each edge is calculated by bringing the input data of the tail and head in dependence:

$$dt_e = \frac{input_{n_t}}{input_{n_h}}. \tag{1}$$

Each job node contains a set of child nodes and edges for the stage-level. On the stage-level, we access the interface for stage metrics of the corresponding application and create a set of nodes containing one element for each stage entry corresponding to one job. In order to derive the parallelism

p_n of each node and whether it represents a spout s_n, we obtain the read data metrics of each stage and distinguish between *input* and *shuffle* data:

$$s_n = \begin{cases} true, & \text{for } input > 0 \wedge shuffle = 0, \quad (2a) \\ false, & \text{otherwise,} \quad (2b) \end{cases}$$

$$p_n = \begin{cases} p_{ds}, & \text{for } input > 0 \wedge shuffle = 0, \quad (3a) \\ p_d, & \text{otherwise.} \quad (3b) \end{cases}$$

In case a stage has read input bytes, the initial RDD of the stage is created by an external data source and contains as many partitions as the data source. This usually applies to each initial stage of a job. For this case, we set s_n to true (Equation (2a)) and specify p_n according to the number of partitions of the data source p_{ds} (Equation (3a)). In case a stage has read shuffled data, the corresponding RDD of the stage is already transformed based on prior RDDs. Its partitions equal the default parallelism p_d. Therefore, we set s_n to false (Equation (2b)) and set p_n to p_a (Equation (3b)). The data transmission factor is calculated as in (Equation (1)). Finally, we extract one Resource Profile for each node element on the stage-level.

4.3. Extraction and Estimation of Resource Profiles

A Resource Profile consists of a set of resource demands where each element may involve a different resource type and a function to specify the value. Our main focus lies on the CPU resource. As Kay et al. [37] systematically identified by the example of Apache Spark, CPU is the bottleneck of data analytics applications in most cases contrary to the widely-accepted opinion that disk and network are weak points.

We define three different CPU demands for each stage $i \in EN$. The first one represents the actual time to process a task. We define a linear function dependent on the parameter p describing the data size for each task of a stage. The slope of the function is calculated by using aggregated CPU times originating from task-related thread groups across all Spark executors. This CPU time is divided by the total amount of read data for each stage:

$$f_{i,cpu,task}(p) = p \frac{cpuTime_{i,task}}{input_i + shuffle_i}. \quad (4)$$

The second CPU demand represents the overhead of coordinating with the driver program, preparing a task before it is actually executed, and postprocessing. These times are provided by the Spark task metrics interface (i.e., they are included in the variables *executorDeserializeTime* and *resultSerializationTime*). As the coordination grows with the number of Spark executors, we define the demand dependent on the configuration parameter e, the number of executors. We observed that this demand varies very strong from task to task, especially for the first tasks of a stage. As averaging the metric is not reasonable, we model this demand by converting the series of time values to a boxed probability density function (PDF) with variable interval sizes as specified by PCM [6]. In order to box the CPU values, we use the percentiles 5, 25, 50, 75 and 95 as intervals since they are provided by the Spark's interface.

The third CPU demand represents the overhead caused by providing infrastructure services for one task. As it is independent of data input, we define a static demand using aggregated CPU times of traces originating from worker-related thread groups across all Spark executors. We additionally divide the CPU times by the total number of tasks to receive the demand for one task:

$$f_{i,cpu,infra} = \frac{cpuTime_{i,worker}}{numComplTasks + numFailTasks}. \quad (5)$$

For the extraction of drive demands, we examined several approaches to estimate read and write demands. As we are not able to measure the drive demands on an appropriate level without adding instrumentation to HDFS (similar to [37]), we extract only a resource demand for reading data, which equals the dependent parameter p describing the data size for each stage.

Similarly, network demands on a low granularity level are only able to be retrieved by instrumenting Spark in a sophisticated way. In order to compensate and include the time delays caused by network traffic, we extract *wait* demands. We calculate the delays between stages by comparing their start and end times and model the demand accordingly.

Furthermore, we do not extract demands for allocating main memory at the moment. As simulation approaches for memory are still limited and neglect features such as garbage collection, the prediction accuracy of this resource is debatable [34].

4.4. Extraction of Data Workload Architectures

The Hadoop distributed file system (HDFS) is a distributed, scalable, and fault-tolerant storage system for big data [38]. Files are split into a sequence of blocks according to a specified block size, which are are replicated to different data nodes to support fault tolerance [38]. For instance, if Spark applications read a file from HDFS, it will be represented by one RDD with as many partitions as blocks.

In order to extract the Data Workload Architecture, we create a file-based data model and a single data source for a specified folder in HDFS and create a file specification for each file. To access the required information, we use the client library of Apache Hadoop. We access the size of each file as well as calculate the partition size and number of overall partitions p_{ds}.

4.5. Extraction of Resource Architectures

Cluster managers, such as Apache Hadoop YARN and Apache Mesos, arbitrate resources for batch and stream applications and provide support to distribute them on cluster nodes. YARN stands for Yet Another Resource Negotiator and follows a master–worker architecture [38]. This includes one resource manager and multiple node managers. A node manager runs on each worker node and is responsible for executing resource containers. A resource container is an abstract notion for resources such as CPU, memory, and HDD in which application tasks run [38]. If a new application is submitted, a responsible application master will be executed in a new resource container. It orchestrates application tasks and, therefore, requests resource containers from the resource manager and monitors their state [39]. Apache Spark is able to run in different modes on YARN. In the so-called client-mode, for instance, the driver program and Spark context runs at the client itself, the application master requests resources for executors, and each executor will run in its own resource container [36].

In order to extract Resource Architectures, we use the public interface provided by YARN to retrieve metrics of each cluster node. For each node manager, we create one resource node $rn \in RN$. Therefore, we assign a worker resource role and create a resource unit for each CPU, drive, and memory. The CPU cores and memory capacity are extracted via the interface. As drive information is not available, we set the read and write speed manually (e.g., by testing HDFS with the included DFSIO benchmark).

Besides the set of resource nodes, we create a network channel and also set the bandwidth and latency manually.

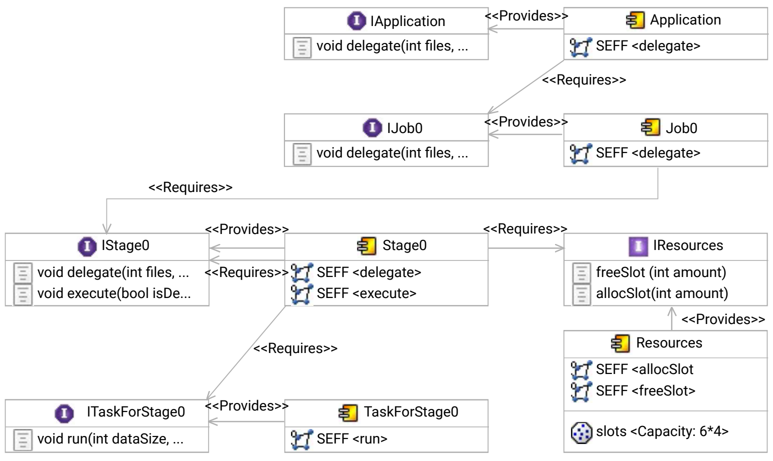
5. Transformation to Performance Models

This section describes the concepts of the architecture-level performance model PCM and how we transform DSL instances into PCM models.

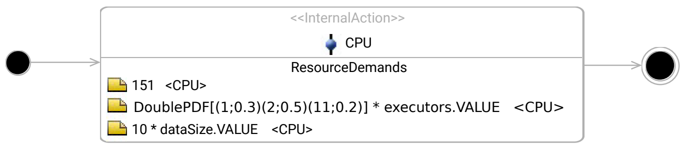
5.1. Palladio Component Model

We chose to use PCM [6] as a model-based performance evaluation tool as it enables engineers to specify software systems independent of technology, include resource demands for software components, consider resource contention, and predict not only response time, but also resource utilization. Furthermore, the tool support is mature, open source, and continuously maintained with a large community.

In particular, PCM is developed for component-based software systems and enables engineers to describe performance relevant factors of software architectures, resource environments, and usage behavior [4]. It is implemented in Ecore from the Eclipse Modeling Framework (EMF) and consists of multiple models [6]. Software interfaces and components are specified in the Repository Model (Figure 4a). Components provide the implementation for signatures of interfaces. Therefore, they contain a resource demanding service effect specification (RDSEFF) in which the activities such as parametric resource demands and external calls of signatures are modeled similar to activity diagrams (Figure 4b). Components are additionally assembled in a System Model. In the Resource Environment Model, network and hardware resources are specified such as processing resources (CPU, disk, and delay), processing rates, and scheduling policies. The Allocation Model allows for deploying assembled components from the System Model on resources from the Resource Environment Model. The usage and workload of software components are specified in the Usage Model. Finally, PCM provides a simulator for its models, which is based on a process-oriented discrete event simulation.



(**a**) PCM Repository model example


<<InternalAction>>
CPU
ResourceDemands
151 <CPU>
DoublePDF[(1;0.3)(2;0.5)(11;0.2)] * executors.VALUE <CPU>
10 * dataSize.VALUE <CPU>

(**b**) Resource demanding SEFF for a task (*PDF* probability density function)

Figure 4. Exemplary transformed PCM instances.

5.2. Transformation to PCM

We describe the transformation for each DSL component. Table 1 shows the mapping of DSL concepts to PCM elements. An Execution Architecture is transformed to a Repository Model (Figure 4a). In order to traverse the Edges and Nodes of an Execution Architecture, we use a recursive depth-first search. Upon visiting each Node, we check if it contains child Nodes and Edges. If this is the case, we again traverse this Node and the procedure repeats.

For each Node, we create one Interface with several signatures and a corresponding Basic Component that *provides* the signatures using an RDSEFF. If a Node contains child Nodes, we add a *delegate* signature to the corresponding Interface (i.e., *IJob0*). Additionally, the Basic Component *requires* the Interfaces of the child Nodes.

Parameters of the Configuration and parametric dependencies of the Execution Architecture are transformed into input parameters of each Signature. We consider parameters for the number of files, the data size of one file, the default partition size, the number of partitions, and the number of executors. In order to model and limit the maximum number of concurrent tasks, we separately specify an Infrastructure Component to represent a pool of available task slots. The component contains two SEFFs to acquire and to release a task slot. In order to finally execute a task, a slot must be acquired first. After task completion, the slot is released again. In the case of Apache Spark, the limiting number of task slots is the number of total cores.

Table 1. Mapping of PerTract-DSL to PCM elements.

***PerTract*-DSL**	***PCM Model Elements***
Execution Architecture	Repository Model
Nodes	Interface, Basic Component
Edges	RDSEFF
Configuration	Parameters, Infrastructure Component
Resource Profile	Distributed Call Action, RDSEFF
Resource Architecture	Resource Environment Model
Resource Node	Resource Container
Cluster Specification	Cluster Specification
Network Channel	Linking Resource
Data Workload Architecture	Usage Model
Data Model	Entry Level SystemCall, Parameters
Data Source	Workload

RDSEFF Resource Demanding Service Effect Specification; *Distributed Call Action, Cluster Specification* PCM extensions [9].

Edges are represented in the RDSEFF of a Basic Component. Each *delegate* RDSEFF models the flow by using External Call Actions to invoke signatures of required Interfaces in the specified order (i.e., *Job0* invokes the *prepare* signature of *IStage0*). In the course of this, the input parameters are forwarded and altered at specific points to model the data transmission factor t_e of an Edge.

If a Node contains a Resource Profile, we transform it by creating several model elements. In order to call a group of tasks in parallel, we add two signatures to the corresponding Interface of the Node (i.e., *Stage0*). The providing RDSEFF *prepare* is intended to create a set of parallel tasks. It uses a Distributed Call Action to invoke the *execute* signature of the same Interface several times in parallel. The parallelism is either defined by the number of partitions of a data source p_{ds} or the specified parallelism of the Node p_n. The *execute* RDSEFF acquires and releases a task slot before and after prompting a task.

We create an additional Interface and Basic Component (i.e., *TaskForStage*) to model a task. Its behavior *run* is responsible to execute the parametric resource demands of a task (Figure 4b). Only the *wait* demand of a Resource Profile will be executed in the prior *prepare* RDSEFF as the demand occurs once at the beginning of each stage and not for each task. We automatically assemble all Basic Components of the Repository Model in order to derive Palladio's System Model.

Since the Resource Architecture follows the concepts of Palladio's Resource Environment Model, the transformation is linear. We transform each Resource Node to a Resource Container and convert the Cluster Specification and Resource Role accordingly. Additionally, we transform each Resource Unit to an equivalent Processing Resource Unit including the specification of processing rates, number of replicas (e.g., the number of cores), and scheduling policies. Finally, all Resource Containers are connected to networks via a Linking Resource.

In order to create the Allocation Model, we deploy all assembled Basic Components from the System Model on the master Resource Container from the Resource Environment Model. Our previous extensions [9] enable Palladio's simulation framework SimuCom to distribute resource demands to Resource Containers that represent worker nodes with a round robin policy.

Finally, we transform the Data Workload Architecture to a Usage Model. We create one Entry Level System Call that invokes the *delegate* signature of the *Application* Interface. The required input parameters are transformed based on the Data Model and Data Source. We specify the number of files, the data size of one file, the default partition size, and the number of partitions. For the Single Data Source, we create a simple closed Workload with a population of one, which means the Entry Level System Call is triggered once.

All transformed models can be used by Palladio's simulator to predict performance metrics.

6. Evaluation

This section evaluates the model extraction and performance simulation approach introduced in this work.

6.1. Research Methodology

In order to validate our approach, we conduct three integrated controlled experiments by modeling and simulating the execution of two different exemplary machine learning applications [40]. Therefore, we formulate three claims by exemplary problems from a performance management perspective.

First, engineers are interested in the performance behavior of applications and resources in case data workload grows. This experiment evaluates the claim that data workloads can be changed independently of Execution Architectures and Resource Architectures. We initially extract one PerTract-DSL instance for each of the two applications based on monitoring data. Afterwards, we adapt data sizes in Data Workload Architectures and compare predictions for response times and CPU utilization with corresponding monitored measurements in several upscaling scenarios.

Second, engineers need to evaluate the scalability of applications if additional hardware resources are allocated. This experiment evaluates the claim that resources can be altered independently of Execution Architectures and Resource Architectures. We modify and add worker nodes in Resource Architectures without changing Execution Architectures and Data Workload Architectures. Afterwards, we compare predictions results with corresponding monitored measurements.

Third, engineers need to efficiently plan and manage capacities for given data workloads and performance requirements [5]. This experiment evaluates the claim that data workloads as well as resources can be changed independently of Execution Architectures. Similarly, we use the models extracted in the first experiment and conduct several upscaling scenarios regarding data workload and cluster size without modifying Execution Architectures. Afterwards, we compare the simulated prediction results with corresponding measurements.

6.2. HiBench Benchmark Suite

In our experiments, we apply the HiBench benchmark suite to run representative and reproducible applications and workloads for Apache Spark [41]. As the automatic extraction approach shall allow for modeling complex applications, we use two machine learning applications. We chose a random forest classification (RFC) since random forests represent frequently used machine learning models for classification and regression. HiBench implemented the application using Apache Spark's machine learning library MLlib and provides an RFC-specific data generator. Additionally, we chose a linear regression (LR) as it is a common approach for regression analysis and forecasting. Therefore, HiBench's implementation uses a model without regularization using a stochastic gradient descent to predict label values. Similarly, it implements Spark's MLlib and includes its own data generator.

6.3. Experiment Setup

Tables 2 and 3 illustrates our testbed and data configurations. The hardware environment includes five servers. Each server is connected to a storage area network (IBM System Storage EXP3512, New York, NY, USA) via fibre channel allowing for eight gigabits per second (GBit/s). The servers are also connected in a local area network (LAN) with one GBit/s.

We virtualized each server using the VMware ESXi hypervisor (VMware, Palo Alto, CA, USA) and configured eight cores and 36-gigabyte (GB) memory for each virtualized machine (VM). On each server, we allocated four VMs. On the first server, we use one VM as a master node for Apache HDFS and YARN, one VM for managing the cluster (i.e., Apache Ambari), one VM for storing monitoring data, and one VM for initiating the benchmark applications. On the remaining four servers, we use each VM as a worker node. We deployed the Hortonworks Data Platform to use Apache Spark, YARN, and HDFS. For HDFS, we kept the default configurations including a replication factor of three and a data block size of 128 megabytes (MB). For YARN, we configured 26 GB and six virtual cores (vCores) per container, for Spark executors 22 GB as well as six cores. Since we experienced that not all cores were utilized when running applications, we changed the resource calculator to be dominant and enabled CPU scheduling to address this issue. For evaluating the prediction accuracy, we compare the metrics response time and CPU utilization. For simulations, we captured the simulated mean response time (MRT) as well as the simulated mean CPU utilization (MCPU) across the cluster. For the benchmark measurements, applications were executed four times for each experiment to avoid any distortions. Similarly, monitored MRT and monitored MCPU on the user-level were calculated. Monitored response times are derived from the Spark monitoring API and monitored CPU measurements from the Ambari Metrics System (2.6.0).

Tables 4 and 5 list all simulated and monitored MRT and MCPU results, the root mean square errors (RMSE), and the relative prediction errors. They provide the basis for presenting and discussing our experiments in the following.

Table 2. Software and hardware configuration of the test system.

Software platform	5×	4×	Hortonworks Data Platform (2.6.3.0-235) - Apache Spark (2.2.0) - Apache Hadoop (2.7.3) - Apache Ambari (2.6.0)
Java virtual machine			Oracle JDK (1.8.0_60)
Operating system			CentOS Linux (7.2.1511)
Virtualization			VMware ESXi (5.1.0), 8 cores, 36 GB RAM
CPU cores		48 × 2.1 GHz	
CPU sockets		4 × AMD Opteron 6172	
Random access memory (RAM)		256 gigabyte (GB)	
Hardware system		IBM System X3755M3	

Table 3. Data workload scenarios and configurations.

Application	**Scenario**	**File Size**	**Files**	**Partitions**	**Total Size**
Random forest classification	Small	1.89 gigabyte	8	128	15.12 gigabyte
	Large	3.58 gigabyte	8	232	28.64 gigabyte
	Huge	5.52 gigabyte	8	360	44.16 gigabyte
Linear regression	Small	1.86 gigabyte	8	120	14.88 gigabyte
	Large	3.49 gigabyte	8	224	27.92 gigabyte
	Huge	5.59 gigabyte	8	360	44.72 gigabyte

Table 4. Monitored and simulated mean response times (seconds).

		Random Forest Classification Application				Linear Regression Application			
Worker Nodes	**Data Workload**	**Monitored MRT**	**Simulated MRT**	**RMSE**	**Prediction Error**	**Monitored MRT**	**Simulated MRT**	**RMSE**	**Prediction Error**
4	Small	264.79	262.71	4.47	0.78%	42.15	43.09	1.19	2.23%
	Large	502.09	462.41	40.26	7.90%	71.96	76.60	4.73	6.45%
	Huge	755.05	696.70	59.65	7.73%	124.21	116.38	13.39	6.30%
8	Small	222.46	199.04	24,92	10.53%	35.28	32.95	2.65	6.59%
	Large	378.31	322.54	56.62	14.74%	52.24	49.74	3.66	4.79%
	Huge	534.12	486.34	48.48	8.94%	76.73	73.54	4.60	4.15%
16	Small	196.62	196.46	4.34	0.08%	37.84	37.33	2.22	1.34%
	Large	287.38	285.20	11.56	0.76%	40.86	45.24	4.48	10.74%
	Huge	373.74	396.38	25.97	6.06%	53.27	56.96	4.05	6.93%

Table 5. Monitored and simulated mean CPU utilization.

		Random Forest Classification Application				Linear Regression Application			
Worker Nodes	**Data Workload**	**Monitored MCPU**	**Simulated MCPU**	**RMSE**	**Prediction Error**	**Monitored MCPU**	**Simulated MCPU**	**RMSE**	**Prediction Error**
4	Small	48.96%	45.69%	3.31%	6.69%	48.86%	47.43%	2.53%	2.94%
	Large	56.93%	48.70%	8.23%	14.45%	57.55%	52.06%	5.62%	9.53%
	Huge	56.06%	49.66%	6.43%	11.42%	56.32%	55.45%	4.02%	1.54%
8	Small	35.23%	34.83%	0.91%	1,13%	36.03%	32.48%	3.72%	9.86%
	Large	44.64%	39.60%	5.31%	11.29%	46.13%	42.51%	3.85%	7.85%
	Huge	47.27%	40.66%	6.61%	13.98%	52.93%	48.15%	4.81%	9.04%
16	Small	22.65%	22.12%	0.84%	2.32%	22.05%	19.34%	2.91%	12.26%
	Large	31.23%	27.61%	3.65%	11.57%	31.85%	28.99%	3.06%	8.97%
	Huge	34.00%	30.72%	3.39%	9.63%	38.22%	35.59%	3.13%	6.89%

6.4. Collecting Resource Demands and Extracting Execution Architectures

The extraction and transformation process follows the overview illustrated in Figure 1. In order to extract an Execution Architecture for one application, we monitor the application using our profiler presented in Section 4.1 to extract stack traces and corresponding CPU times. Additionally, the Spark framework itself monitors an application. As described in Section 4.2, execution components and inter-component interactions are extracted using Spark's interfaces. For each execution component, CPU resource demands are generated by processing corresponding CPU times and interrelating them with data input information of each component as explained in Section 4.3.

In order to evaluate the three proposed claims, we derive one initial PerTract-DSL instance for each of the two machine learning applications that we use throughout all experiments. According to each experiment and scenario, we adapt the PerTract-DSL instance and simulate it to derive predictions.

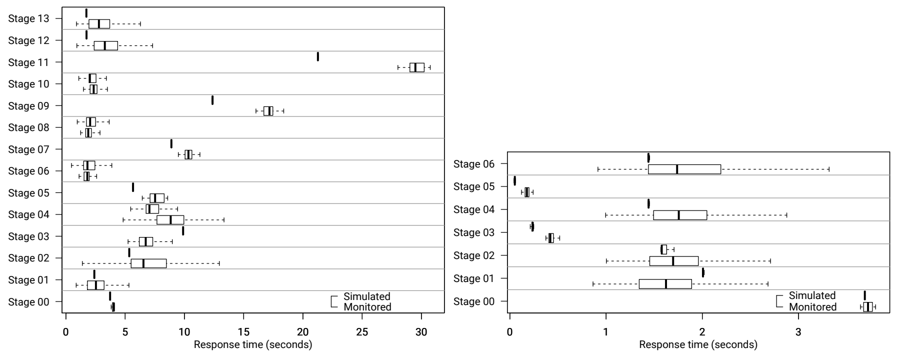
6.5. Evaluating Data Workload Changes

In order to evaluate our first claim that data workload changes can be modified independently, we specified three different scenarios *small*, *large*, and *huge* for both applications. Table 3 shows the corresponding number of files, file sizes, total partitions and total sizes for each scenario. The basis for evaluating workload changes of each application provides one initial PerTract-DSL instance each. We extracted this instance from a monitored experiment with a small data workload in a cluster of four worker nodes. Afterwards, we changed the Data Workload Architecture according to the scenarios large and huge and simulated the model instances. The simulation and monitoring results are part of Tables 4 and 5.

The starting experiment (i.e., four nodes and small workload) shows a response time prediction error of 0.78% for the RFC and 2.23% for the LR application. CPU prediction errors amount to 6.69% and 2.94%. Changing the data workload according to the large and huge scenarios leads to a response time prediction error of 7.90% and 7.73% for the RFC and 6.45% and 6.30% for the LR applications. Similar to the prediction errors, the RMSE increased in both scenarios. For the huge scenario, Figure 5 illustrates the response time statistics of simulated and monitored Spark tasks for each stage. For both applications, we predict the median of the tasks for 16 of 21 stages with errors below 30%. However, the monitored results show an increased deviation compared to the simulation results, especially, for the LR application. This is due to the monitored delays and task processing, which showed great variances. While we model delays with probability distributions, we only use the mean for estimating CPU demands and did not depict this behavior. For the RFC application, tasks for stages *05*, *07*, *09*, and *11* also differ significantly. These stages contain reduce operations for which the input data size does not exactly scale linearly with increasing data workload for this RFC application. However, the error only has a minor effect on the overall application response time as stages for reduce tasks consist of only eight tasks compared to 360 tasks for each of the other stages.

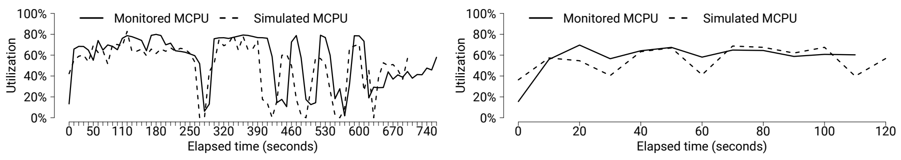
For the large and huge workload scenarios, the RMSE for CPU consistently remain below 9%. CPU prediction errors amount to 14.45% and 11.42% for the RFC and 9.53% and 1.54% for the LR application. Figure 6 illustrates the CPU utilization over time for one experiment run. In order to avoid illustrating too many lines, we calculated the mean across the worker nodes. Although underestimating the CPU utilization by 6.4% for the RFC application, the graphs of the simulated and monitored values map very closely.

The results for response time and resource utilization show accurate prediction results based upon averages for upscaling workload changes. Therefore, we validated the claim of being able to change data workloads independent of Execution Architectures and Resource Architectures.



(**a**) Random forest classification (**b**) Linear regression

Figure 5. Response time statistics of Spark tasks for each stage (four worker nodes, huge data workload).



(**a**) Random forest classification (**b**) Linear regression

Figure 6. Mean CPU utilization of four worker nodes (huge data workload).

6.6. Evaluating Resource Changes

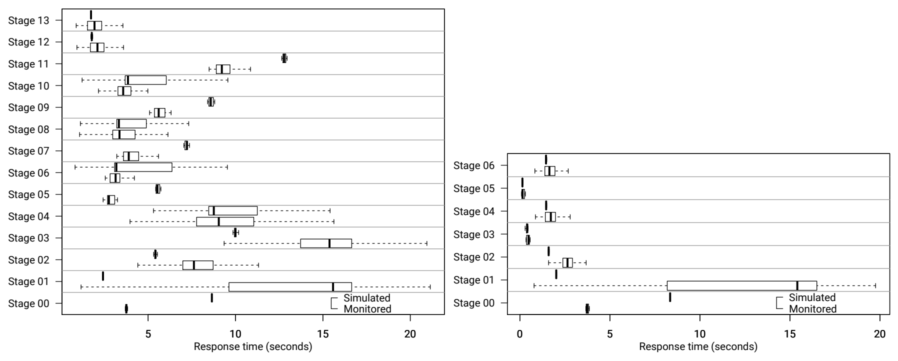
We increased the initial cluster size of four worker nodes by factors two and four in order to evaluate our second claim that hardware resources can be changed independently of Execution Architectures and Data Workload Architectures.

Similarly, the evaluation is based on one initial PerTract-DSL instance for each application, which is the same as for the data workload evaluation and was extracted from a monitored experiment with four worker nodes. Afterwards, we increased the worker nodes to eight and 16 nodes in the Resource Architecture. Additionally, we adapted the number of executors *e* in the application configuration of the Execution Architecture to match the number of worker nodes. The simulation and monitoring results are part of Tables 4 and 5.

In the previous subsection, we already discussed the same starting experiment, which does not include any changes. For eight worker and 16 worker nodes, response time prediction errors amount to 10.53% and 0.08% for the RFC application and 6.59% and 1.34% for the LR application, respectively. Compared to the data workload changes, the RMSE is lower throughout the resource changes for both applications. Figure 7 additionally shows the detailed response time statistics of Spark tasks for each stage of the applications. Compared to the data workload evaluation, the median values of simulated and monitored results lie closer together. The distance of the first and third quartiles are also predicted more accurately for most stages of both applications. For a few stages such as *Stage 01*, minimum, maximum, and quartiles differ significantly. Nonetheless, response time predictions errors on application-level remain below 15% in total.

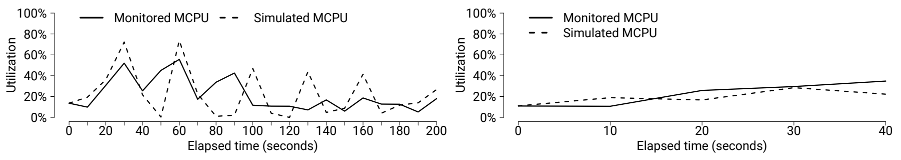
For eight worker and 16 worker nodes, CPU prediction errors come to 1.13% and 2.32% for the RFC application and to 9.86% and 12.26% for the LR application, respectively. Figure 8 illustrates the CPU utilization over time for one experiment run. For the RFC application, the simulated CPU usage overestimates several peaks and underestimates negative peaks. However, it depicts the progression of the monitored results overall. For the LR application, the predicted CPU utilization is very precise.

In total, the simulation results show accurate prediction results for upscaling hardware resource changes with mean prediction errors below 15% and validate the claim that hardware resource can be modified without changing Execution Architectures and Data Workload Architectures.



(**a**) Random forest classification (**b**) Linear regression

Figure 7. Response time statistics of Spark tasks for each stage (16 worker nodes, small data workload).



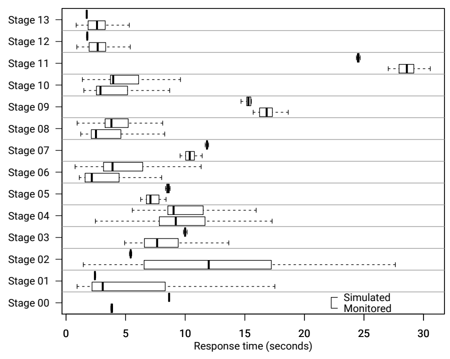
(**a**) Random forest classification (**b**) Linear regression

Figure 8. Mean CPU utilization of 16 worker nodes (small data workload).

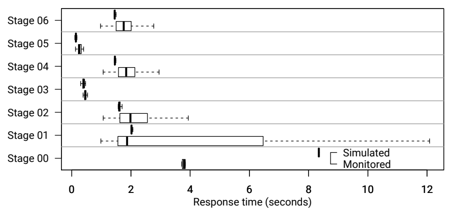
6.7. Evaluating Data Workload and Resource Changes

In order to evaluate our claim that data workload and hardware resources can be modified without changing application Execution Architectures, we applied both upscaling scenarios together, regarding data workload as well as worker nodes. The simulation and monitoring results are part of Tables 4 and 5. Again, the evaluation is based on the same initially extracted PerTract-DSL instance for each application.

For eight worker nodes and a large data workload, response time prediction errors amount to 14.74% for the RFC and 4.79% for the LR application. For huge data workload, the errors are 8.94% and 4.15%, respectively. For 16 worker nodes and a large data workload, response time prediction errors come to 0.76% for the RFC and 10.74% for the LR application. With huge data workload, the errors are 6.06% and 6.93%, respectively. The RMSE results consistently behave similarly to prediction errors. The highest RMSE amounts to 56.62 s, which equals 14.97% of the corresponding monitored response times. For all scenarios, prediction errors constantly remain below 15%. Figure 9 additionally shows the response time statistics of results with 16 worker nodes and huge workload. Compared to the two previous evaluations, the simulation results depict monitoring results as the closest for both applications.



(**a**) Random forest classification

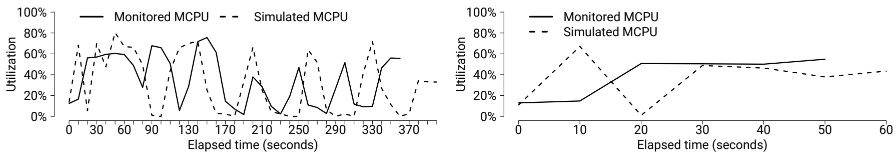


(**b**) Linear regression

Figure 9. Response time statistics of Spark tasks for each stage (16 worker nodes, huge data workload).

Looking at the CPU results for eight worker nodes and a large data workload, prediction errors amount to 11.29% for the RFC application and 7.85% for the LR application. For a huge workload, the errors remain similarly with 13.98% and 9.04%. For 16 worker nodes and a large data workload, the errors also remain 11.57% and 8.97%. With a huge data workload, they decrease a little to 9.63% and 6.89%, similar to the response time prediction.

Figure 10 shows the CPU utilization over time of one run with 16 worker nodes and a huge data workload. In case of the RFC application, the simulation graph depicts the progression of the monitored measurements. However, it shifts as the response time differs. In case of the LR application, the simulated CPU utilization is also slightly shifted due to the different response times. Otherwise, it depicts the monitored utilization except for one peak at the beginning. This is due to overestimating the CPU demand for *Stage 00*. Similarly, the task response time also significantly differs for *Stage 00* for both applications throughout all experiments. The reason for the overestimation is that this stage consists of only one task, which does not scale linearly with the dependent data size. This is a case that we intentionally did not consider and could not cover as it requires metaknowledge of the application that we do not expect in an automatic extraction process.



(**a**) Random forest classification (**b**) Linear regression

Figure 10. Mean CPU utilization of 16 worker nodes (huge data workload).

Overall, the simulated results for response times on an application-level as well as CPU utilization show accurate predictions for both data workload changes and hardware resources. The mean prediction errors remained below 15% as well as the RMSE compared to the monitored results. In performance evaluation literature, prediction errors of 30% across cluster sizes are expected [20]. Therefore, we validated the claim of being able to change data workloads and resources' architectures independent of Execution Architectures. Our approach enriches related work by predicting CPU utilization across clusters and over time.

6.8. Threats to Validity

Although we applied some sophisticated machine learning applications, we generated data and used only a set of sample applications from one benchmark suite. As they are far more complex applications and have deviating data in praxis, this represents a threat to external validity [42].

Furthermore, we evaluated our approach only for one technology (i.e., Apache Spark) and one type of application (i.e., batch). In previous work, we showed that our approach is also applicable for Spark Streaming applications [11]. However, we claim that the DSL builds a foundation to specify other technologies as well, such as Apache Flink and Apache Storm. Extensions might be required (e.g., additional parameters) to support modeling and accurate predictions. We plan to evaluate this in our future work.

We used several visualizations and statistical measures such as mean, standard deviation, and relative error to ensure statistical conclusion validity. While the results of one measure can be close to each other (e.g., mean), another measure can differ significantly (e.g., minimum value).

6.9. Assumptions and Limitations

We allocated one Spark executor to each node during our experiments. It is also possible to size less cores and memory for Spark executors, which would enable Spark to allocate multiple executors to one node. Although we are also able to model and simulate these scenarios, we did not evaluate such a case. We evaluated our experiments in a virtualized, but exclusive cluster in which no other applications were running in parallel and using any CPU, disk drives, or networks. For data analytics applications, CPU is usually the bottleneck [37]. As HiBench and other industry benchmarks mainly consist of only compute-intensive applications, we did not evaluate our approach for a wider variety of applications.

Regarding our modeling approach, we specified the input of a subsequent Spark stage probabilistically depending on the output data of a previous stage. Therefore, our prediction error will increase, if the properties of the initial underlying data set change significantly (e.g., the number of distinct words in case of a word count application). Another limitation is that we only include network delays in our models and simulations, but did not simulate network throughput and bandwidth yet. The same applies to disk drives. In addition, we also did not consider rack awareness in our specification. Regarding big data features and PCM, Heinrich et al. [43] discuss current challenges and potential solutions, for instance, for modeling data structures and continuous data flows.

7. Conclusions and Future Work

Modeling and predicting the performance of big data applications are essential for planning capacities and evaluating configurations. Automatically deriving models, specifying applications tool-agnostic, and gaining insights into performance-relevant factors of system architectures and dependencies are complex challenges. We present PerTract, an approach to automatically extract model specifications and transform them to the model-based performance evaluation tool Palladio. A PerTract-DSL allows the specification of (i) application execution architectures including components, parametric dependencies, and resource demands, (ii) computing resources, and (iii) data workloads. It is specifically designed for big data systems, decreases the complexity compared to full performance models, and simplifies the changeability to users. We demonstrated the extraction of DSL instances by the example of Apache Spark applications, Apache YARN resources, and Apache HDFS data. This is the first white-box approach to present an automated way to integrate measurements and estimate resource demands to produce performance models that can be simulated. We used two machine learning applications of the HiBench benchmark suite in the evaluation and upscaled data sizes as well as cluster sizes in different scenarios. We are able to predict mean response times on application-level and CPU usage with accurate predictions errors below 15%.

In our future work, we plan to extract DSL instances from more technologies. We already provide a way to extract the execution architecture of Apache Flink applications, but need further investigations to estimate accurate resource demands. Additional technologies include Apache Mesos for modeling computing resources and Apache Kafka for characterizing data workload. We also plan to implement direct transformations from the DSL to a scalable event-oriented discrete-event simulation as we are reaching the limit for simulating continuous sources (data streams). Finally, we will extend the specification of continuous data sources to include load intensity profiles that model variations in arrival rates [44].

Author Contributions: Conceptualization: J.K. and H.K.; Resources: H.K.; Software: J.K.; Validation: J.K.; Writing—original draft: J.K.; Writing—review and editing: J.K. and H.K.; Supervision: H.K.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CPU	Central processing unit
DSL	Domain-specific language
EMF	Eclipse modeling framework
GB	Gigabyte
HDFS	Hadoop distributed file system
LAN	Local area network
LR	Linear regression
MB	Megabytes
MCPU	Mean CPU utilization
MRT	Mean response time
PCM	Palladio component model
PDF	Probability density function
RDD	Resilient distributed dataset
RDSEFF	Resource demanding service effect specification
RFC	Random forest classification
RMSE	Root mean square error
VM	Virtualized machine

References

1. Schermann, M.; Hemsen, H.; Buchmüller, C.; Bitter, T.; Krcmar, H.; Markl, V.; Hoeren, T. Big Data—An interdisciplinary opportunity for information systems research. *Bus. Inf. Syst. Eng.* **2014**, *6*, 261–266. [CrossRef]
2. Brunnert, A.; Vögele, C.; Danciu, A.; Pfaff, M.; Mayer, M.; Krcmar, H. Performance management work. *Bus. Inf. Syst. Eng.* **2014**, *6*, 177–179. [CrossRef]
3. Wang, K.; Khan, M.M.H. Performance Prediction for Apache Spark Platform. In Proceedings of the 17th International Conference on High Performance Computing and Communications, New York, NY, USA, 24–26 August 2015; pp. 166–173.
4. Brosig, F.; Meier, P.; Becker, S.; Koziolek, A.; Koziolek, H.; Kounev, S. Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-Based Architectures. *IEEE Trans. Softw. Eng.* **2015**, *41*, 157–175. [CrossRef]
5. Brunnert, A.; van Hoorn, A.; Willnecker, F.; Danciu, A.; Hasselbring, W.; Heger, C.; Herbst, N.; Jamshidi, P.; Jung, R.; von Kistowski, J.; et al. *Performance-Oriented DevOps: A Research Agenda*; Technical Report SPEC-RG-2015-01; SPEC Research Group—DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC): Gainesville, FL, USA, 2015. Available online: http://research.spec.org/fileadmin/user_upload/documents/wg_devops/endorsed_publications/SPEC-RG-2015-001-DevOpsPerformanceResearchAgenda.pdf (accessed on 8 August 2019).
6. Becker, S.; Koziolek, H.; Reussner, R. The Palladio component model for model-driven performance prediction. *J. Syst. Softw.* **2009**, *82*, 3–22. [CrossRef]
7. Kroß, J. PerTract. Available online: https://github.com/johanneskross/pertract (accessed on 7 August 2019).
8. Kroß, J.; Brunnert, A.; Prehofer, C.; Runkler, T.; Krcmar, H. Stream Processing on Demand for Lambda Architectures. In *Computer Performance Engineering*; Beltrán, M., Knottenbelt, W., Bradley, J., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2015; Volume 9272, pp. 243–257.
9. Kroß, J.; Brunnert, A.; Krcmar, H. Modeling Big Data Systems by Extending the Palladio Component Model. In Proceedings of the 2015 Symposium on Software Performance, Munich, Germany, 4–6 November 2015.
10. Kroß, J.; Krcmar, H. Modeling and Simulating Apache Spark Streaming Applications. In Proceedings of the 2016 Symposium on Software Performance, Kiel, Germany, 8–9 November 2016.
11. Kroß, J.; Krcmar, H. Model-based Performance Evaluation of Batch and Stream Applications for Big Data. In Proceedings of the IEEE 25th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), Banff, AB, Canada, 20–22 September 2017; pp. 80–86.
12. Vianna, E.; Comarela, G.; Pontes, T.; Almeida, J.; Almeida, V.; Wilkinson, K.; Kuno, H.; Dayal, U. Analytical Performance Models for MapReduce Workloads. *Int. J. Parallel Program.* **2013**, *41*, 495–525. [CrossRef]
13. Verma, A.; Cherkasova, L.; Campbell, R.H. Profiling and evaluating hardware choices for MapReduce environments: An application-aware approach. *Perform. Eval.* **2014**, *79*, 328–344. [CrossRef]
14. Zhang, Z.; Cherkasova, L.; Loo, B.T. Benchmarking Approach for Designing a Mapreduce Performance Model. In Proceedings of the ACM/SPEC International Conference on Performance Engineering, Prague, Czech Republic, 21–24 April 2013; ACM Press: New York, NY, USA, 2013; pp. 253–258.
15. Zhang, Z.; Cherkasova, L.; Loo, B.T. Performance Modeling of MapReduce Jobs in Heterogeneous Cloud Environments. In Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, 28 June–3 July 2013; IEEE: Washington, DC, USA, 2013; pp. 839–846.
16. Zhang, Z.; Cherkasova, L.; Loo, B.T. Exploiting Cloud Heterogeneity to Optimize Performance and Cost of MapReduce Processing. *SIGMETRICS Perform. Eval. Rev.* **2015**, *42*, 38–50. [CrossRef]
17. Barbierato, E.; Gribaudo, M.; Iacono, M. Performance evaluation of NoSQL big-data applications using multi-formalism models. *Future Gener. Comput. Syst.* **2014**, *37*, 345–353. [CrossRef]
18. Ardagna, D.; Bernardi, S.; Gianniti, E.; Karimian Aliabadi, S.; Perez-Palacin, D.; Requeno, J.I. Modeling Performance of Hadoop Applications: A Journey from Queueing Networks to Stochastic Well Formed Nets. In *Algorithms and Architectures for Parallel Processing*; Carretero, J., Garcia-Blas, J., Ko, R.K., Mueller, P., Nakano, K., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2016; pp. 599–613.

19. Lehrig, S. Applying Architectural Templates for Design-Time Scalability and Elasticity Analyses of SaaS Applications. In Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability, Dublin, Ireland, 22 March 2014; pp. 2:1–2:8.
20. Ardagna, D.; Barbierato, E.; Evangelinou, A.; Gianniti, E.; Gribaudo, M.; Pinto, T.B.M.; Guimarães, A.; da Silva, A.P.C.; Almeida, J.M. Performance Prediction of Cloud-Based Big Data Applications. In Proceedings of the ACM/SPEC International Conference on Performance Engineering, Berlin, Germany, 9–13 April 2018; pp. 192–199.
21. Singhal, R.; Singh, P. Performance Assurance Model for Applications on SPARK Platform. In *Performance Evaluation and Benchmarking for the Analytics Era*; Nambiar, R., Poess, M., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2018; pp. 131–146.
22. Venkataraman, S.; Yang, Z.; Franklin, M.; Recht, B.; Stoica, I. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), Santa Clara, CA, USA, 13–17 March 2016; USENIX Association: Santa Clara, CA, USA, 2016; pp. 363–378.
23. Alipourfard, O.; Liu, H.H.; Chen, J.; Venkataraman, S.; Yum, M.; Zhang, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, 27–29 March 2017; USENIX Association: Boston, MA, USA, 2017; pp. 469–482.
24. Witt, C.; Bux, M.; Gusew, W.; Leser, U. Predictive performance modeling for distributed batch processing using black box monitoring and machine learning. *Inf. Syst.* **2019**, *82*, 33–52. [CrossRef]
25. Castiglione, A.; Gribaudo, M.; Iacono, M.; Palmieri, F. Modeling performances of concurrent big data applications. *Softw. Pract. Exp.* **2014**, *45*, 1127–1144. [CrossRef]
26. Niemann, R. Towards the Prediction of the Performance and Energy Efficiency of Distributed Data Management Systems. In Proceedings of the ACM/SPEC International Conference on Performance Engineering, Delft, The Netherlands, 12–16 March 2016; pp. 23–28.
27. Casale, G.; Ardagna, D.; Artac, M.; Barbier, F.; Nitto, E.D.; Henry, A.; Iuhasz, G.; Joubert, C.; Merseguer, J.; Munteanu, V.I.; et al. DICE: Quality-driven Development of Data-intensive Cloud Applications. In Proceedings of the Seventh International Workshop on Modeling in Software Engineering, Florence, Italy, 16–24 May 2015; pp. 78–83.
28. Guerriero, M.; Tajfar, S.; Tamburri, D.A.; Di Nitto, E. Towards a Model-driven Design Tool for Big Data Architectures. In Proceedings of the 2nd International Workshop on BIG Data Software Engineering, Austin, TX, USA, 2016; pp. 37–43.
29. Gómez, A.; Merseguer, J.; Di Nitto, E.; Tamburri, D.A. Towards a UML Profile for Data Intensive Applications. In Proceedings of the 2Nd International Workshop on Quality-Aware DevOps, Saarbrücken, Germany, 21 July 2016; pp. 18–23.
30. Ginis, R.; Strom, R.E. Method for Predicting Performance of Distributed Stream Processing Systems. U.S. Patent 7,818,417, 19 October 2010.
31. Steinberg, D.; Budinsky, F.; Paternostro, M.; Merks, E. *EMF: Eclipse Modeling Framework*, 2nd ed.; Addison-Wesley: Boston, MA, USA, 2009.
32. King, B. *Performance Assurance for IT Systems*; Auerbach Publications: Boston, MA, USA, 2004.
33. Brandl, R.; Bichler, M.; Ströbel, M. Cost accounting for shared IT infrastructures. *Wirtschaftsinformatik* **2007**, *49*, 83–94. [CrossRef]
34. Brunnert, A.; Krcmar, H. Continuous Performance Evaluation and Capacity Planning Using Resource Profiles for Enterprise Applications. *J. Syst. Softw.* **2017**, *123*, 239–262. [CrossRef]
35. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, San Jose, CA, USA, 25–27 April 2012; USENIX Association: Berkeley, CA, USA, 2012; p. 2.
36. Apache Spark. Lightning-Fast Cluster Computing. Available online: https://spark.apache.org (accessed on 19 February 2018).

37. Ousterhout, K.; Rasti, R.; Ratnasamy, S.; Shenker, S.; Chun, B.G. Making Sense of Performance in Data Analytics Frameworks. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, Oakland, CA, USA, 4–6 May 2015; USENIX Association: Oakland, CA, USA, 2015; pp. 293–307.
38. Apache Hadoop. Welcome to Apache Hadoop! Available online: https://hadoop.apache.org/ (accessed on 1 January 2017).
39. Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
40. Hevner, A.R.; March, S.T.; Park, J.; Ram, S. Design Science in Information Systems Research. *MIS Q.* **2004**, *28*, 75–105. [CrossRef]
41. Huang, S.; Huang, J.; Dai, J.; Xie, T.; Huang, B. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In Proceedings of the 26th International Conference on Data Engineering Workshops, Long Beach, CA, USA, 1–6 March 2010; pp. 41–51.
42. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering*; Springer: Berlin/Heidelberg, Germany, 2012.
43. Heinrich, R.; Eichelberger, H.; Schmid, K. Performance Modeling in the Age of Big Data—Some Reflections on Current Limitations. In Proceedings of the 3rd International Workshop on Interplay of Model-Driven and Component-Based Software Engineering, Saint-Malo, France, 2 October 2016; pp. 37–38.
44. Kistowski, J.V.; Herbst, N.; Kounev, S.; Groenda, H.; Stier, C.; Lehrig, S. Modeling and Extracting Load Intensity Profiles. *ACM Trans. Auton. Adapt. Syst.* **2017**, *11*, 23:1–23:28. [CrossRef]