Energy-Conscious Deployment Optimization for Component-Based Cyber-Foraging Systems

Master’s Thesis of

Sebastian Dieter Krach

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)
and the
Carnegie Mellon Software Engineering Institute (SEI)

Reviewer: Prof. Dr. Ralf H. Reussner
Second reviewer: Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor: M.Sc. Grace Lewis (SEI)
Second advisor: M.Sc. Christian Stier (KIT)

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Pittsburgh, April 30, 2015

(Sebastian Dieter Krach)
Abstract

Cyber-foraging systems circumvent resource restrictions of mobile devices by offloading parts of the application to more powerful infrastructure in close proximity. A non-negligible constraint is put on the devices by their limited battery capacity. Being able to assess energy consumption already at design time allows to identify viable candidate components for offloading early in the development process.

In this thesis I analyze the applicability of model-based performance analyses to generate energy consumption estimations for mobile devices to be able to decide between multiple architecture candidates for the most energy efficient option. In particular, I focus on component-based architectures for cyber-foraging applications as components encapsulate distinct application functionality and therefore already partition applications into separate entities.

Using an external power monitor I develop an automated power model calibration approach calculating regression analyses over power measurements determined by a controlled execution of micro-benchmarks. Architecture-level energy consumption predictions are generated leveraging Palladio performance analyses and existing energy consumption prediction extensions. By extending a multi-objective software architecture optimization approach I provide means of automatically analyzing the impact of component allocation decisions on the mobile device energy consumption.

Conducting a case study on an existing speech recognition application I found energy consumption estimates to be highly sensitive to response time changes. The presented approach is either able to correctly identify the optimal deployment scheme or results in candidates not worse than 1% of the optimal solution’s measured consumption.

In this thesis I show that model-based performance analysis can be leveraged to generate design-time information on energy consumption as long as sufficiently exact performance models can be created. Further work is necessary to model execution environments more detailedy and, in particular, refine the power models employed for network transfers.
Zusammenfassung


Die Auswertung einer realen Spracherkennungsanwendung zeigte eine hohe Sensitivität der Energieverbrauchsvorhersage auf Veränderungen der Antwortzeit des Systems. Der vorgestellte Ansatz ist in der Lage, die Partitionierungsschemen zu identifizieren, die einen Energieverbrauch mit einer Abweichung von 1% oder weniger der optimalen Lösung vorweisen.

Contents

Abstract i
Zusammenfassung iii

1. Introduction 1
   1.1. Goal of this Thesis ........................................... 1
   1.2. Structure of this Thesis ..................................... 2
   1.3. Style Guide ...................................................... 2

2. Foundations 3
   2.1. Cloud-Based Mobile Augmentation ................................. 3
       2.1.1. Cloud Offloading ......................................... 3
       2.1.2. Cyber-Foraging ......................................... 4
   2.2. Models and Meta-Models ......................................... 5
       2.2.1. Eclipse Modeling Framework (EMF) ......................... 6
       2.2.2. Model Transformation ................................... 6
   2.3. The Palladio Approach to Component-Based Software Engineering 7
       2.3.1. Palladio Component Model ................................ 8
       2.3.2. Model-Based Performance Analysis ......................... 13
   2.4. Automated Architecture Optimization ............................. 13
       2.4.1. Design Decision Models ................................ 14
       2.4.2. Quality of Service Modeling Language .................... 14
   2.5. Energy Consumption Analysis .................................... 15
       2.5.1. Basics ...................................................... 15
       2.5.2. Power Models .............................................. 15
   2.6. Energy Consumption Prediction Using Palladio .................... 17

3. Concept and Approach 19
   3.1. General Approach ............................................... 19
   3.2. Automated Power Model Generation ............................... 20
       3.2.1. Benchmark Workload ...................................... 21
       3.2.2. System Metric Targets .................................... 21
       3.2.3. Power Consumption Measurements ......................... 22
       3.2.4. Raw Power Consumption Analysis .......................... 23
   3.3. Application Resource Demand Simulation ........................ 24
       3.3.1. Modeling Network Demand .................................. 25
       3.3.2. Determining Network Demand ............................... 26
### Contents

3.4. Energy Consumption Prediction ........................................... 28  
3.5. Automated Architecture Optimization ................................. 30  
  3.5.1. Quality Criteria Specification .................................. 30  
  3.5.2. Optimization Mechanism .......................................... 31  

4. Automated Device Power Profiling ................................. 33  
  4.1. Experiment Specification ........................................... 33  
    4.1.1. The Experiment Specification Meta-Model .................. 34  
  4.2. Experiment Execution ............................................. 37  
    4.2.1. Experiment Execution Scenario ............................. 37  
    4.2.2. Mobile Device Application ................................. 39  
    4.2.3. Profiling Server ........................................... 45  
  4.3. Measurement Data Preprocessing ................................ 47  
    4.3.1. Experiment Data Management ............................... 48  
    4.3.2. Aggregate & Filter Architecture .......................... 51  
  4.4. Model Parameter Extraction .................................. 56  
    4.4.1. The Power Model Generator Approach .................... 58  
    4.4.2. Automated Regression Analysis ........................... 59  
  4.5. Limitations and Future Work ................................... 67  

5. Simulation ................................................................. 71  
  5.1. Device-Specific Network Performance Completions ............ 72  
    5.1.1. The Concept ................................................. 72  
    5.1.2. Network Demand Proxy .................................... 73  
    5.1.3. Network Demand Middleware ............................... 75  
    5.1.4. Example: CMU Sphinx ..................................... 75  
    5.1.5. QVTo Model Transformation ............................... 77  
  5.2. Energy Consumption Prediction ................................ 82  
    5.2.1. Extension of Power Prediction Capabilities ............. 84  
    5.2.2. Expression-based Power Consumption Calculation ........ 89  
    5.2.3. Power Consumption Integration ............................ 89  

  6.1. Quality Dimensions Extension ................................ 91  
  6.2. Energy Consumption Analysis ................................ 92  
  6.3. Limitations ....................................................... 94  

7. Evaluation ............................................................... 95  
  7.1. Automated Profiling Approach ................................ 95  
    7.1.1. Linear CPU Power Model .................................. 95  
    7.1.2. WiFi Power Model Approximation ........................ 97  
  7.2. Speech Preparation .............................................. 98  
    7.2.1. The Sphinx Architecture ................................ 99  
    7.2.2. Model Creation for Speech ............................... 101  
    7.2.3. Model Calibration ....................................... 102
## List of Figures

2.2. Source code and Resource Demanding Service Effect Specification (RDSEFF) example (Source: [7]) ................................................. 10  
2.3. Approximating a pdf by specifying discrete intervals (Source: [33]) ... 11  
2.4. The ResourceEnvironment for the cyber-foraging scenario ............... 11  
2.5. PerOpteryx process model (Source: [30]) ................................... 14  
2.6. WiFi interface power states [77] ............................................... 16  
2.7. Excerpt from Infrastructure, Binding and Specification meta-models [68] 17  

3.1. Overview of the profiling experiment scenario ............................. 23  
3.2. The proposed network interface ............................................... 26  
3.3. Example: Network demand depends on component allocations .......... 27  
3.4. Example: Direct specification of network demand ....................... 27  
3.5. Relationship between response time and energy consumption trace .... 32  

4.1. Excerpt from the BindingModelGeneration meta-model ................ 34  
4.2. ExperimentDimension meta-model ........................................... 36  
4.3. AndroidProfiler workflow states and transitions ....................... 40  
4.4. Resource profiling experiment architecture ............................. 42  
4.5. ProfilingServer modules ...................................................... 46  
4.6. EDP2: ExperimentData class structure (simplified) .................... 49  
4.7. Power consumption trace: supposedly idle CPU ......................... 52  
4.8. Aggregate and filter architecture: the principle (CPU) ............... 53  
4.9. EDP2 architecture extension: annotations .............................. 54  
4.10. MeasuringValue aggregation ................................................ 55  
4.11. Regression model structure ................................................ 59  
4.12. Linear power model as ExpressionOasis expression ................... 61  
4.14. Linear power model specification using ExpressionOasis ............ 62  
4.15. Export of measurement data to R ....................................... 63  
4.16. Regression Model Calculator class hierarchy ......................... 64  
4.17. Assembling the NLS regression model string ......................... 65  
4.18. Transforming ExpressionOasis model to R compatible representation .. 66  

5.1. Exemplary situation of two components interconnected pre-transformation 73  
5.2. Exemplary situation of two components interconnected post-transformation 73  
5.3. Repository model of two components taken from CMU Sphinx including generated proxy ................... 75
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>System model corresponding to the Repository model in Figure 5.3</td>
<td>76</td>
</tr>
<tr>
<td>5.5</td>
<td>Allocation model corresponding to the System model in Figure 5.4</td>
<td>76</td>
</tr>
<tr>
<td>5.6</td>
<td>RDSEFF corresponding to proxy component in Figure 5.3</td>
<td>77</td>
</tr>
<tr>
<td>5.7</td>
<td>Infrastructure model of Galaxy Nexus device</td>
<td>82</td>
</tr>
<tr>
<td>5.8</td>
<td>Excerpt from PCA architecture [68]</td>
<td>83</td>
</tr>
<tr>
<td>5.9</td>
<td>Iteration behavior of EvaluationContext</td>
<td>85</td>
</tr>
<tr>
<td>5.10</td>
<td>EvaluationScope and PowerModelCalculator interface excerpt</td>
<td>86</td>
</tr>
<tr>
<td>5.11</td>
<td>Extended PCA architecture (for CPU and WiFi)</td>
<td>88</td>
</tr>
<tr>
<td>5.12</td>
<td>Extended PCA instance at run-time (for CPU and WiFi)</td>
<td>90</td>
</tr>
<tr>
<td>6.1</td>
<td>Aggregation of energy consumption values</td>
<td>93</td>
</tr>
<tr>
<td>7.1</td>
<td>Power traces for utilization targets in CPU profiling experiment</td>
<td>96</td>
</tr>
<tr>
<td>7.2</td>
<td>Plot of linear power model and raw measurements</td>
<td>97</td>
</tr>
<tr>
<td>7.3</td>
<td>Plot of exponential power model and raw measurements</td>
<td>97</td>
</tr>
<tr>
<td>7.4</td>
<td>Power traces for selected targets during WiFi profiling experiment</td>
<td>98</td>
</tr>
<tr>
<td>7.5</td>
<td>Plot of approximated power model and raw measurements</td>
<td>99</td>
</tr>
<tr>
<td>7.6</td>
<td>Architecture overview of CMU Sphinx (Source: [70])</td>
<td>100</td>
</tr>
<tr>
<td>7.7</td>
<td>CMU Sphinx FrontEnd data flow (Source: [70])</td>
<td>101</td>
</tr>
<tr>
<td>7.8</td>
<td>Simulated and Measured Response Time at Different Transmission Speeds</td>
<td>106</td>
</tr>
<tr>
<td>7.9</td>
<td>Power profiles of unchanged speech for different speeds and file sizes</td>
<td>109</td>
</tr>
<tr>
<td>7.10</td>
<td>Simulated and measured energy consumption at different transmission speeds</td>
<td>110</td>
</tr>
<tr>
<td>7.11</td>
<td>Error between measured and simulated energy consumption before and after linear correction by the response time deviation</td>
<td>114</td>
</tr>
</tbody>
</table>
# List of Tables

1.1. Overview of styles used in this thesis ............................................. 2

3.1. WiFi-related metric extensions ......................................................... 30

4.1. Overview of Protocol Buffers result file parser ................................. 50

7.1. Order of optimal deployment schemes (simulated and measured) at unlimited WiFi transmission speed ($\approx 2.2\text{MByte/s}$) ........................................... 111

7.2. Order of optimal deployment schemes (simulated and measured) at 128KByte/s 112

7.3. Order of optimal deployment schemes (simulated and measured) at 64KByte/s 113

A.1. Summary of Palladio Component Model (PCM) elements relevant to this thesis (I) .......................................................... 131

A.2. Summary of PCM elements relevant to this thesis (II) ........................ 132

B.1. Overview of different CMU Sphinx data objects that pass the FrontEnd pipeline ................................................................. 133
Acronyms

**API**  Application Programming Interface  
**CMU**  Carnegie Mellon University  
**CPU**  Central Processing Unit  
**DI**  Dependency Injection  
**DRY**  Don’t Repeat Yourself  
**DUT**  Device-under-Test  
**DVFS**  Dynamic Voltage Frequency Scaling  
**EDP**  Experiment Data Presentation and Persistence  
**EMF**  Eclipse Modeling Framework  
**EMOF**  Essential MOF  
**GPU**  Graphics Processing Unit  
**HMM**  Hidden Markov Model  
**IPD**  Institute of Program Structures and Data Organization  
**IoC**  Inversion of Control  
**JRI**  Java/R Interface  
**KIT**  Karlsruhe Institute of Technology  
**LTI**  Language Technology Institute  
**M2M**  Model-2-Model  
**M2T**  Model-2-Text  
**MDA**  Model-Driven-Architecture  
**MOF**  Meta Object Facility  
**MTTF**  Mean Time To Failure  
**MTTR**  Mean Time To Repair
List of Tables

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTU</td>
<td>Maximum Transmission Unit</td>
</tr>
<tr>
<td>oAW</td>
<td>openArchitectureWare</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PCA</td>
<td>Power Consumption Analyzer</td>
</tr>
<tr>
<td>PCM</td>
<td>Palladio Component Model</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain Old Java Object</td>
</tr>
<tr>
<td>PSU</td>
<td>Power Supply Unit</td>
</tr>
<tr>
<td>QML</td>
<td>Quality of Service Modeling Language</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>QVT</td>
<td>Query/View/Transformation</td>
</tr>
<tr>
<td>QVTo</td>
<td>Query/View/Transformation Operational Mappings</td>
</tr>
<tr>
<td>RDSEFF</td>
<td>Resource Demanding Service Effect Specification</td>
</tr>
<tr>
<td>RUP</td>
<td>Rational Unified Process</td>
</tr>
<tr>
<td>RTT</td>
<td>round-trip time</td>
</tr>
<tr>
<td>SDQ</td>
<td>Software Design and Quality</td>
</tr>
<tr>
<td>SEI</td>
<td>Carnegie Mellon Software Engineering Institute</td>
</tr>
<tr>
<td>SoC</td>
<td>Separation of Concerns</td>
</tr>
<tr>
<td>StoEx</td>
<td>Stochastic Expression</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
1. Introduction

While CPU performance of modern handheld devices has continued to grow significantly, an increase of the battery capacity in the same order of magnitude is indiscernible. Cloud-based mobile augmentation describes a concept, where cloud computing infrastructure is leveraged to "increase, enhance and optimize computing capabilities" [1]. Apple’s Siri [2] is one of the most commonly used examples for cloud-offloading applications. The phone records your voice and transmits the record to a more powerful cloud server, which in turn executes language processing. Recent work [16, 14, 34, 35, 20] analyzes offloading of resource-intensive tasks into the cloud in order to increase battery run-time.

Satyanarayanan et al. [60] propose cyber-foraging as trade-off between increased latency and dependence on the stability of the internet connection on the one hand, and increased performance through more sophisticated hardware on the other. The authors introduce surrogate hosts or cloudlets, micro data centers located in one-hop proximity to the mobile device. A significant increase in performance and decrease in latency can be achieved through offloading application components to cloudlet instead of a distant cloud. In particular for crisis situations poor connectivity and limited resource availability present serious obstacles, e.g. for first responders. Cloudlet-based cyber-foraging can increase the efficiency of tactical edge applications, e.g. face recognition, helping to overcome the aforementioned obstacles [39].

Application execution efficiency on mobile devices can benefit from cyber-foraging in various ways: increased performance through more sophisticated CPUs, decreased dependence on large-scale mobile networks through relying on local one-hop connections, and increased battery run-time through offloading of computations with high resource demand. Nevertheless, offloading application components from mobile device to a cloudlet does not necessarily lead to the desired decrease in energy consumption. Generally speaking, the execution of a component on a cloudlet can be advantageous if the energy that would be consumed by local execution exceeds the energy necessary for data transmission and waiting for the cloudlet to finish processing.

1.1. Goal of this Thesis

The hardware and software platforms of mobile devices and cloudlets usually differ significantly. While cloudlets mostly employ powerful hardware built for desktop or server use, mobile devices rely on energy-efficient variants. Although the platforms are usually incompatible, there exist frameworks which support the offloading of application logic to the cloud(let). CloudClone [14] and MAUI [16] are two representatives which rely on application virtualization (DalvikVM and .Net-CLR, respectively). Nevertheless, both approaches are dependent on the underlying virtualization platform. Furthermore, using
1. **Introduction**

Platform-specific services often allows a more efficient implementation of the application logic.

The goal of this thesis is to enable application developers to determine which components are suitable to offloading, already during design time. Using a model-based simulation I want to derive the optimal allocation of components with regard to the energy consumption. With PerOpteryx there exists already an approach which determines configurations for a PCM instance which are Pareto-optimal with respect to certain quality criteria. Currently, PerOpteryx does not support energy consumption as a quality measure.

The main contribution of this thesis is an automatic partitioning optimization for component-based applications. The optimization is determined depending on a model-based simulation of the application’s resource consumption using energy consumption characteristics for the simulated mobile device.

### 1.2. Structure of this Thesis

Chapter 2 introduces the foundations for this thesis. At first, I outline cyber-foraging concepts, basics of model-driven and component-based development as well as the Palladio approach. Furthermore, I introduce the automated architecture optimization approach PerOpteryx. Finally, I present how modeling techniques can be used to express energy and power consumption and their integration into the Palladio framework. Thereafter, Chapter 3 discusses the concept of this thesis and identifies the distinct work packages which are then examined in Chapters 4 to 6. The realized approach is validated using a real-world speech recognition application in Chapter 7. Related work in the domain of modeling the energy consumption of mobile devices and energy-conscious application partitioning are presented in Chapter 8. Chapter 9 concludes the thesis and summarizes the major findings.

### 1.3. Style Guide

Throughout the thesis I use different font styles to indicate meaning and give contextual information. Table 1.1 presents an overview of the styles used throughout this document.

<table>
<thead>
<tr>
<th>Font style</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model Transformation</strong></td>
<td>Emphasizing new terminology, together with the explanation</td>
</tr>
<tr>
<td>public void foo()</td>
<td>Reference to element in source code artifact (e.g. class, object, method, attribute)</td>
</tr>
<tr>
<td>RepositoryComponent</td>
<td>Model element type, a.k.a meta-model element</td>
</tr>
<tr>
<td>ComponentA</td>
<td>Concrete model element (usually element identifier), to distinguish from element type</td>
</tr>
</tbody>
</table>

Table 1.1.: Overview of styles used in this thesis
2. Foundations

The following chapter provides an overview of scientific work which form the foundations for the approach of this thesis. I want to explore the applicability of architecture quality analysis using model-based predictions with respect to energy consumption in the context of cyber-foraging applications. Therefore, this chapter consists of three major parts. At first the concepts of cyber-foraging is presented and the important aspects from the perspective of this thesis are highlighted. Thereafter, the term component-based software is defined and the concepts of model-based quality analysis and optimization introduced. This second part focuses on the PCM as means of describing software architectures, which forms an essential basis of this thesis. The third major part of the foundations chapter focuses energy and power consumption analyses for mobile devices as well as concepts for predictive modeling of the consumption.

2.1. Cloud-Based Mobile Augmentation

Mobile devices present significant restrictions, particularly with respect to battery capacity. Cloud-based mobile augmentation schemes help to overcome mobile device resource restrictions [4]. Mobile applications can leverage more powerful infrastructures accessible using a (wireless) network connection.

2.1.1. Cloud Offloading

Offloading of application parts is a very common augmentation scheme. It refers to the execution of certain application parts on a server instead of the mobile device. The reduced computational work on the mobile device is payed for by increased network transmissions and additional coordination effort. Kumar et al. [35] identified computation and communication costs as the two determining factors for offloading decisions. Furthermore, different restrictions for offloading schemes exist, e.g. application parts which are dependent on mobile-device specific hardware (e.g. camera, sensors) can only be executed locally.

Application partitioning or offloading mechanisms can be categorized as either static or dynamic. Static approaches answer the question which application parts are supposed to be offloaded before the application is executed and once for the entire run-time. Dynamic approaches rely on run-time profiling techniques to determine the optimal partitioning while the application is executing. While dynamic approaches are able to react on changes to the environment (e.g. network congestions), they increase the application complexity and often limit application developers design possibilities.

Offloading mechanisms differ with respect to their granularity. Lewis et al. [40] identify five levels of granularity: Process, Function, Component, Service and Application.
2. Foundations

Application Level Entire application logic is offloaded. The mobile device client is kept as small as possible.

Service Level Offloading of self-contained artefacts, accessible through specified interfaces, which encapsulate coarse-grained application capabilities.

Component Level Software artefacts, mostly determined to execute in specific container, are offloaded. The artefacts are most-likely self-contained but often show tight dependencies to the execution platform on the mobile device.

Function Level Functions, methods or operations are offloaded. Global states have to be synchronized. Offloading candidates are often manually annotated.

Process Level A clone of the mobile device runs virtualized on the offloading target. Process states are exchanged. Process level granularity allows the most fine-grained offloading schemes.

In this thesis the focus will lie on component-based software architectures. A more restrictive definition of components will be applied in comparison with aforementioned component-level offloading granularity (see Section 2.3 for details).

2.1.2. Cyber-Foraging

Cloud-based mobile augmentation schemes can significantly increase mobile application capabilities. However, offloading of application parts to a distant cloud infrastructure depends on a good network quality. The necessary availability and accessibility of cloud resources can not be guaranteed in certain hostile environments. Ha et al. [20] define these hostile environments as dominantly characterized by a "short-term large-magnitude uncertainty". The expression refers to disturbances that influence in particular network connections, potentially culminating in a complete breakdown of the network infrastructure.

Therefore, mobile application design for personnel operating at the tactical edge, e.g. first responders in crisis areas, is constrained by certain environmental characteristics. The characteristics include resource-restrictions (e.g. limited battery capacity), unreliable connection to traditional infrastructure, potentially large amounts of data and unpredictable environmental changes. Furthermore, the personnel has to operate under stress and high cognitive loads [39].

Ha et al. [20] employ cyber-foraging mechanisms to improve mobile device usage in hostile environments. Cyber-foraging refers to enhancing performance of mobile clients through the usage of opportunistically discovered servers in the environment [3]. Ha et al. propose the introduction of cloudlets as surrogate hosts in single-hop proximity of the mobile device. Cloudlets are stateless servers located in direct proximity of mobile devices which are discovered and afterward initialized dynamically upon usage [60]. Simanta et al. [62] present a reference architecture (see Figure 2.1) for hostile environments leveraging the concept of Ha et al. [20].

Simanta et al. [62] use VM synthesis as core technique to provision discovered cloudlets fast. VM synthesis requires a base VM image containing the underlying operating system
Models are formal representations of real world entities and their relationships. According to Stachowiak every model is characterized by the three properties: 1. Homomorphism, 2. Abstraction and 3. Pragmatism [63].

Homomorphism refers to elements of the model representing real-world entities and there exists a valid bi-directional mapping between them. Abstraction describes model elements as simplified representations of the corresponding real-world entity. Aspects that are not important in the model context are removed. Pragmatism refers to models being created with a particular purpose in mind.

The valid elements of a model are specified by a meta-model. Meta-models define the necessary parts and rules to create valid models. Meta-models themselves are models specified by a meta-meta-model. The Object Management Group (OMG) describes the four hierarchies M0 to M3: M0 being model instances, M3 meta-meta models.

The Unified Modeling Language (UML) is a modeling language used to describe software architectures during the development process. UML’s objective is "to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems" [49].

Valid UML models are specified by the Meta Object Facility (MOF), a simplified version of the class modeling capabilities provided by UML [47]. In the four-layer meta-model hierarchy of the OMG the models would be placed as follows [49, 64].
2. Foundations

**M3** *Meta-Meta-Model:* MOF, specifying valid elements for M2 models (e.g. Class)

**M2** *Meta-Model:* UML meta-model, (e.g. Class, Attribute, Instance, all three instances of MOF element Class)

**M1** *Model:* UML model instance, (arbitrary user model)

**M0** *Instances:* Instances of model representation on M1 (run-time instances)

Meta-models specify the abstract syntax for models, that is, a representation-independent specification of the valid model elements and their relations (e.g. UML: class, attribute). A model’s concrete syntax defines the way model instances are represented (e.g. UML: rectangles containing text) [64].

A particular concrete syntax for all models compliant with the MOF is the XML Metadata Interchange (XMI) format. XMI describes the usage of eXtensible Markup Language (XML) and XML schemas to describe MOF compliant models and ultimately store them in a textual representation [50].

**2.2.1. Eclipse Modeling Framework (EMF)**

The Eclipse Modeling Framework (EMF) builds upon the Eclipse framework to provide practical means for modeling and model-driven software engineering. It allows to unify UML, XML and program code representations of the same model entities [66].

EMF specifies its own meta-model Ecore, basically a subset of the UML meta-model. The Ecore meta-model in itself is a valid EMF model. EMF provides graphical editors to create and work with Ecore models leveraging the XMI format for persistence [66].

A major feature of the EMF is the capability to specify own meta-models using Ecore and based on them generate Java code for the handling of conforming model instances. Furthermore, EMF provides Eclipse-based editors for creating and manipulating models instances based on the meta-model specification [66].

**2.2.2. Model Transformation**

Model transformations are defined with respect to a set of input meta-models and desired output artifacts. For instances of the input models the transformation generates the corresponding output. Model transformations approaches can be distinguished into Model-2-Model (M2M) and Model-2-Text (M2T) transformations.

The first type refers to approaches that produce as output also a model instance. Dependent on the concrete transformation the output model is compliant to the same or to a different meta-model than the input. For *in-place* transformation the changes are applied to the actual input model instance directly. M2T transformations create text-based output artifacts based on the input models and are used, e.g. by EMF to convert Ecore models into executable Java code.

Model transformations play a crucial role in the OMG’s Model-Driven-Architecture (MDA) concept. The MDA standard describes an approach to model-driven software development leveraging amongst others the UML, the MOF and the XMI standards. MDAs
generally describe software systems using three different layers of model abstractions (domain models, logical system models and implementation models) [48].

Query/View/Transformation (QVT) is the standardized model transformation approach as part of the MDA concept [45]. The three parts query, view and transformation refer to the principle of how QVT describes the relations between models.

**Query** Describes selection mechanisms to extract relevant parts from the source model.

**View** Describes schemes for the selected data in the target meta-model.

**Transformation** Describes the actual changes to generate the target representation from the source.

Three different languages are defined as part of the QVT specification: Core Language (QVTc), Relations Language (QVTr) and Operational Mappings (QVTto). While QVTr and QVTc take a descriptive approach to specify relationships between source and target models, QVTto provides constructs of imperative programming languages. Although the upcoming release of the Eclipse Model to Model Transformation Project (MMT) will bring support for QVTr the QVTto implementation is more mature and established.

**Query/View/Transformation Operational Mappings (QVTto)** The signature of a QVTto transformation specifies the meta-models for input, output or input/output parameters. Instantiate and executing the transformation requires model instances for input and input/output parameters to be specified. Operational Mappings supports an imperative model transformation approach. Therefore, the entry to a QVTto transformation is its main routine. The main routine is an ordered sequence of expressions which are evaluated in linear order.

QVTto distinguishes three types of top-level operations: mappings, queries and helpers. Mappings specify relations between elements in the source and in the target model. The mapping operation defines input and output meta-model elements and is executed only once per input element. Subsequent mapping operations for the same model element always return the same created target model element. Mapping operations are usually called on a collection of compatible source model elements and result in a collection of target model elements not necessarily of the same size. It is possible to specify guards in order to conduct the mapping only for elements that fulfill pre- (when-clause) or postconditions (where-clause). Every mapping operation can be transformed into a semantically identical declarative relation (QVTr).

Helpers are ordered sequences of expressions that are able to manipulate parameters and cause side-effects. They resemble very much normal methods in imperative programming languages. Queries are helper operations limited to side-effect free behavior.

### 2.3. The Palladio Approach to Component-Based Software Engineering

According to Cheesman and Daniels component-based software engineering takes a divide and conquer approach to cope with system complexity [12]. The goal is to structure a
problem and decompose it into smaller units that can be solved more easily. Although they refrain from defining the term component explicitly, Szyperski provides a good summary of the main properties [69]. The term component throughout this thesis is used in accordance with his definition.

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Cheesman and Daniels state that the main advantage of using component-based architectures is a facilitated handling of changes due to components being able to be substituted by other ones. In order to be able to substitute components, they are required to have properly specified interfaces [12, p. xiv]. The exact same requirement is formulated by Szyperski as "contractually specified interfaces and explicit context dependencies" [69].

Furthermore, formally specified contracts for components on non-functional properties, e.g., response time, allows reasoning about the properties of the composed system [6].

An advantage of component-based architectures is the possibility of implementing components independently from each other after formally specifying their interfaces. Koziolek and Happe present a Quality of Service (QoS)-driven component-based development process [32]. The process is based on Cheesman and Daniels’ component-based development process [12]. Taking QoS-criteria into account early in the development process allows to base architectural design decisions on QoS-related requirements.

Koziolek and Happe identify four roles in the development process: component developer, system architect, system deployer, and domain expert.

Component developers are responsible for specifying components and their implementation. System architects assemble complete systems from single components and ensure that each component is provided with its required dependencies. System deployers determine the execution context for the components of a system. The execution context essentially specifies the infrastructure the system will run on. Domain experts provide statistical data to describe the behavior of the target user group in order to assess whether the architecture design fulfills the required QoS-criteria.

### 2.3.1. Palladio Component Model

The Palladio Component Model (PCM) is a domain-specific language to describe component-based software architectures. The perspectives of the four roles identified by Koziolek and Happe are reflected in the model structure of the PCM. Component and interface specifications as determined by the component developer are described through RepositoryModels. SystemModels encapsulate the system architect’s decisions regarding the composition of components to systems. The system deployer is responsible for specifying the execution environment as ResourceEnvironmentModel and the components deployment to it as AllocationModel. Domain experts use their knowledge to formalize usage patterns as UsageModel.

The following description of the individual PCM models is based on the technical report by Reussner et al. [53].
2.3. The Palladio Approach to Component-Based Software Engineering

2.3.1. Repository Model

Repository models describe interfaces and components of the architecture. Every component has to have at least one *Provides Interface*, specifying services offered by this component. In turn, a component can rely on an arbitrary number of *Requires Interfaces*, specifying services which have to be provided by other components in order for this component to work.

Interfaces in the context of the PCM specify a list of operation signatures. They are used as contracts between components insofar as that components which *provide* the interface are required to provide all the specified operations. Therefore, components that explicitly declare an interface as required can rely on all its operations to be offered by the providing entity. The PCM uses Roles to describe requiring and providing relationships between components and interfaces. Components having a *ProvidedRole* represents a contractual commitment to provide the operations specified by the referenced interface. Similar, a *RequiredRole* explicitly specifies the component’s dependency to operations specified in the referenced interface. *RequiredRoles* fulfill Szyperski’s requirement for components to specify dependencies explicitly.

The PCM distinguishes a three level hierarchy for components. As interaction with components is only allowed through explicitly specified interfaces every component has to provide at least one. The most abstract type of a component only specifies the component’s *ProvidedRoles*. For every so-called *Provided Component Type* there exists an arbitrary number of possible *Complete Component Types* which additionally describe the components’ *RequiredRoles*. Complete types still only provide black-box information about the component.

Details on realization of provided functionality is given by *ImplementationComponentTypes* which are further distinguished into *BasicComponents* and *CompositeComponents*. While the latter specify a composition of multiple *CompleteComponents*, *BasicComponents* describe component functionality on the lowest level of composition. As it is the goal of the PCM to enable predictions on non-functional software characteristics, *BasicComponents* provide measures to describe the encapsulated behavior, that is, provide information about its internal complexity and how the component interacts with its dependencies.

**RDSEFF** Resource Demanding Service Effect Specifications (RDSEFFs) describe the inner behavior of a component by specifying the order of calls to required components *ExternalCall* and the internally generated resource demand. Resource demand is an abstract representation of the load the component would generate on the hardware environment during execution.

Leveraging a syntax similar to UML Activity Diagrams RDSEFFs specify the execution flow for a distinct service provided by the component. Figure 2.2 presents a small piece of Java code and its representation as RDSEFF. The execution of the *innerMethod()* in the example is assumed to consume 1000 CPU units.

RDSEFFs allow component developers to model the abstract resource demand of the distinct components without having to incorporate the demand of its dependencies. Explicitly specifying the invocations of external services allows to estimate the demands of a component’s dependencies based on their corresponding RDSEFFs. Therefore, the
2. Foundations

Figure 2.2.: Source code and RDSEFF example (Source: [7])

QoS characteristics for the entire system can be analyzed based on the composition of the system from distinct BasicComponents.

**StoEx** PCM models allow to specify selected model parameters, e.g. resource demand specifications, loop iterations and parameters of operation calls, using probabilistic values. The so called Stochastic Expression (StoEx) framework [33] provides an expression language allowing to describe constant values, probabilistic distributions and almost arbitrary combinations of them.

StoEx distinguishes the data types integer, real, boolean and enum. Expressions are dynamically typed and use a type inference mechanism. Literals represent constant values and are specified by the value only (e.g. 5 is a valid IntLiteral, 1.7 a valid DoubleLiteral).

Two kinds of probabilistic distributions are supported, discrete and continuous random variables. Probability Mass Functions (PMF) specify for a finite set of values the probability for each value to be picked whenever a random sample is taken. The following StoEx evaluates to 5 statistically in half of all cases and with equal probabilities to 10 or 20 otherwise.

\[
\text{IntPMF}[(5;0.5)(10;0.25)(20;0.25)]
\]

Probability Density Functions (PDF) describe statistical distributions for continuous dimensions by specifying intervals together with the probability of a random sample to be within the boundaries of this distinct interval. Figure 2.3 presents an example depicting the concept of approximating the probability distribution using discrete intervals.

2.3.1.2. System Model

Systems represent enclosed entities composed from distinct components. AssemblyContexts specify instantiations of components taken from a repository. AssemblyConnectors connect every RequiredRole of an instantiated component to a different component instance that
2.3. The Palladio Approach to Component-Based Software Engineering

Figure 2.3.: Approximating a pdf by specifying discrete intervals (Source: [33])

has an according **ProvidedRole** or delegates the responsibility to another **System** (specified as **SystemRequiredRole**). Furthermore, **Systems** declare provided interfaces for its users to interact with.

The instantiated components (**AssemblyContexts**) are specified in order to be allocated to a suitable execution environment. Still, the **System** does not incorporate information about the deployment scheme.

**2.3.1.3. Resource Environment and Allocation Model**

Reasoning about non-functional characteristics of represented systems requires taking the execution environments into account. The **PCM ResourceEnvironment** model specifies the infrastructure the system will run on. It describes executing nodes, their resources (e.g. CPU, HDD), execution containers (e.g. application servers), and network connections. Figure 2.4 presents the **ResourceEnvironment** model representing the cyber-foraging scenario.

Figure 2.4.: The **ResourceEnvironment** for the cyber-foraging scenario

Resources are abstract specifications of capabilities that are required by components during their execution and have to be provided by the executing environment. The **PCM** distinguishes two kinds of **Resources**: **ProcessingResources** and **PassiveResources**. **ProcessingResources** represent entities of the execution environment that actively process resource demands. **ProcessingResourceTypes** represent distinct abstractions from a concrete processing capability provided by the execution environment (e.g. the **ProcessingResourceType**
2. Foundations

CPU represents an environment’s capability of handling CPU demand requests. Similar to OperationInterfaces specifying a Component’s capabilities, ResourceInterfaces specify the signatures through which demand can be issued to an instance of the appropriate ResourceTypes.

PassiveResources act as semaphores and can be acquired or released in order to provide synchronization capabilities and model limited resource access. ResourceRepositories capture available ResourceInterfaces and provide a common ground for component developers and system deployers in a sense that both operate on the same set of hardware capability specifications.

The ResourceEnvironment specifies a set of ResourceContainers representing infrastructure entities to which instantiated components can be deployed. ResourceContainers are equipped with a set of ProcessingResourceSpecifications describing the entities capabilities of handling resource demand. Every ProcessingResourceSpecification related to a corresponding ProcessingResourceType and quantifies the resource’s processing capabilities as Processing Rate. The manner in which a concrete resource processes issued demands is determined by a SchedulingPolicy (e.g. First-Come-First-Served).

CommunicationLinkResourceTypes and their instantiations LinkingResource represent entire networks. In the same way a ProcessingResourceSpecification represents a parameterized resource of a distinct ProcessingResourceType, LinkingResources represent parameterized networks of a CommunicationLinkResourceType. While ProcessingResourceSpecifications are ResourceContainer-local LinkingResources are specified for the entire ResourceEnvironment and reference the ResourceContainers that are interconnected by the represented network.

The Allocation of a System specifies for each AssemblyContext the ResourceContainer to which the component instances is deployed to.

2.3.1.4. Usage Model

Domain experts describe in UsageScenarios how users interact with the system. The Usage Models consists of one or more UsageScenarios, specifying which system services are requested by the user and the order in which they are called. Every usage scenario is characterized by a specification of the user behavior and the workload.

User behavior is represented as ScenarioBehavior which similar to the RDSEFF notation uses a UML Activity Diagramm-like syntax. While there is no resource demand specification at the scenario-level it allows to model calls to the System’s provided services, describe loops and branches, and leverage the capabilities of the StoEx framework.

The PCM distinguishes with respect to UsageScenario workload specifications OpenWorkloads and ClosedWorkloads. OpenWorkloads are characterized by an unbound amount of users and parameterized with an inter-arrival time. The inter-arrival time specifies the duration between two users arriving at the system using as a StoEx. Whenever a user arrives at the system an independent execution of the corresponding ScenarioBehavior is started. Closed Workloads specify a constant number of users and a think time. After arriving at the system and executing the ScenarioBehavior each user waits for the duration of the think time before starting from the beginning. Similar to the inter-arrival time the think time is specified as StoEx.
2.4. Automated Architecture Optimization

2.3.2. Model-Based Performance Analysis

Palladio Bench\(^1\) is a set of Eclipse\(^2\) plug-ins that support creating PCM models and conducting various analyses in order to assess different architecture quality aspects. Leveraging the extensible Eclipse platform new analyses can be implemented and dynamically attached. Currently, there are 12 model solvers, analyses and simulation mechanisms available providing support to evaluate the four quality dimensions: Performance, Reliability, Maintainability and Costs \(^52\).

SimuCom \(^6\) was the first analysis method and constitutes the default choice in assessing performance characteristics of component-based architectures. Using model-to-text transformations executable simulation code is generated from the PCM instance. The code is bundled and loaded as a separate Eclipse plug-in into the Palladio Bench instance.

Leveraging the DESMO-J framework\(^3\) a discrete-event simulation is executed. During the execution parameters of the simulated system are recorded, in particular, response time for service calls and information on the simulated resource activity. After the simulation finishes, the recorded measurements are available and can be visualized by various graphic analyses.

2.4. Automated Architecture Optimization

Palladio can be used to decide among multiple architecture candidates for the best one with respect to one of the supported quality dimension. Every candidate has to be expressed as separate PCM instance and the desired analysis executed. With an increasing number of design decisions and potentially multiple quality dimension to be evaluated the process becomes from very time consuming to practically infeasible.

PerOpteryx \(^42\) constitutes an approach to automatically conduct multi-criteria optimizations based on PCM architecture descriptions. Instead of requiring the user to manually provide models for each candidate PerOpteryx relies on a formal specification of the available degrees of freedom. Based on this specification and an initial architecture candidate PerOpteryx conducts an evolutionary optimization. Figure 2.5 depicts the high-level optimization process.

Evolutionary optimization as realized by PerOpteryx iteratively determines a set (generation) of candidates, evaluates each one and generates a new set for the next round. PerOpteryx evaluates candidates by conducting a series of analyses confirming to a optimization goal specification. For each quality dimension the analyses produce a ranking among all evaluated candidates.

Based on the initially specified architecture candidate and a set of degrees of freedom new candidates are created by selecting a possible choice for each of the degrees of freedom and adapting the initial candidate accordingly. Therefore, every candidate among the set of evaluated ones can be uniquely identified by the combination of particular choices taken for the different degrees of freedom.

\(^1\)http://www.palladio-simulator.com/
\(^2\)http://www.eclipse.org/
\(^3\)http://desmoj.sourceforge.net/
2. Foundations

2.1. Search problem instantiation

2.2. Evolutionary Optimisation

2.3. Present results

Figure 2.5.: PerOpteryx process model (Source: [30])

2.4.1. Design Decision Models

The Design Decision model specifies a set of degrees of freedom for a distinct PCM instance. Martens et al. [42] list the three degrees of freedom Allocation, Alternative Component and Hardware as their initially realized set.

Allocation degrees of freedom allow to specify for a component assembly the set of resource containers where it can be allocated to. For the deployment optimization presented in this thesis allocation degrees of freedom are essential as they specify for each component the choice between running locally or offloading to more powerful infrastructure. Alternative component degrees allow to replace the assembly of a deployed component with a different, compatible one. Hardware degrees allow modifications to the ResourceEnvironment (e.g. adapt processing rate of CPU).

2.4.2. Quality of Service Modeling Language

PerOpteryx’ multi-criteria optimization of software architectures is controlled by a specification of the optimization goals. Noorshams et al. [46] developed a model-based representation of the Quality of Service Modeling Language (QML) as formal way of expressing constraints and goals for the multi-criteria optimization. QML was presented in 1998 by Frølund and Koistinen as a language to describe Quality of Service (QoS)-criteria and characteristics for distributed object-oriented systems [19].

The QML is designed to integrate with object-oriented concepts. It allows interface-providing entities to specify QoS guarantees and interface-requiring ones to declare minimal QoS-characteristics that have to be provided.

QML distinguishes the three top-level entities Contract Type, Contract and Profile. Contract Types provide a description of the dimensions that can be used in instantiating Contracts. Dimensions in the context of PerOpteryx specify a measure for a particular quality characteristic (e.g. the response time for the characteristic performance) and how
the measure is supposed to be evaluated during optimization (e.g. lower response times are better).

Contracts instantiate Contract Types by specifying constraints and optimization goals for the dimensions. Profiles bind Contracts to particular interfaces and UsageScenario or System-level objectives.

2.5. Energy Consumption Analysis

In times of increasing prices for electricity the energy consumption becomes an important design criterion for modern computer systems. The life-time of battery powered devices in particular is directly influenced by its energy consumption.

2.5.1. Basics

The energy consumption of a device is determined by its power consumption over time:

$$E(t) = \int_0^t P(t) dt$$  \hspace{1cm} (2.1)

The power consumption can be measured based on the voltage and the electric current momentarily drawn from a battery:

$$P(t) = I(t) \times U(t)$$  \hspace{1cm} (2.2)

2.5.2. Power Models

Energy consumption models as well as power models are abstractions of real world energy or power consumption behavior respectively. The difference between energy models and power models is the relevance of the time parameter. Energy consumption models usually assign certain events (e.g. a CPU cycle, a systemcall or the network transmission of 1 Byte) with a amount of energy. Therefore, energy consumption models usually do not take the time perspective into account. In contrast, power models predict the momentary draw of electricity based on other metrics describing the current state of the system. Energy consumption values can be derived from power models using integration over the time perspective.

Power models describe relations between resource consumption and the amount of power drawn by the system. According to Rivoire et al.[57] an ideal power model has to have several characteristics. The most notable are:

**Accuracy** The power consumption prediction of the model has to reflect the real world consumption well.

**Portability** The power model should be applicable to different hardware designs and not depend on characteristics only offered by a certain design.
Applicability for different workloads  The prediction of the power model should be valid without limitations to the tasks that are performed on the modeled hardware.

Accuracy and portability of a power model is highly dependent on its degree of abstraction. The more hardware details are taken into account, the more accurately a prediction can become. On the other hand, every hardware detail additionally taken into account reduces the portability, since only platforms which comply to the model can be analyzed.

Power model parameters (e.g. the following equations) are device-specific abstractions of the concrete device hardware characteristics. The parameters can be determined using detailed information of the hardware platform. Therefore, many scientific approaches consider it the responsibility of device manufacturers to provide the models [43, 26].

2.5.2.1. Linear CPU Power Model

Many power models proposed for mobile devices rely on Linear Regression (LR), which assumes a linear relationship between system metrics and the energy consumption of a single component [61, 10, 77, 17, 43, 26]. In particular, for the CPU most models use utilization as predictor for energy consumption. Modern mobile processors employ Dynamic Voltage Frequency Scaling (DVFS) to reduce energy consumption, when not executing under full load. Therefore, most models account for different states ($freq_i$ in Equation 2.3).

$$P_{CPU}(t) = \sum_{i=1}^{n} \theta_{freq_i}(t) \cdot \beta_{freq_i} \cdot u(t), \quad \theta_{freq_i} = \begin{cases} 1, & freq_i \text{ active} \nonumber \vspace{1em} \\ 0, & \text{else} \end{cases} \tag{2.3}$$

2.5.2.2. WiFi Power Model

Zhang et al. [77] identify four power states for the WiFi interface. The state machine is shown in Figure 2.6. The energy consumption of the interface is determined entirely by the consumption of the active state if the state does not support sending of data (low, high). Outgoing traffic is additionally accounted for on a linear basis [77, 26].

$$P_{WiFi}(t) = \begin{cases} \beta_{LT} \cdot p(t) + \beta_{LT base}, & \text{if } p \leq \text{Threshold} \nonumber \vspace{1em} \\ \beta_{HT} \cdot p(t) + \beta_{HT base}, & \text{if } p > \text{Threshold} \end{cases} \tag{2.4}$$

Figure 2.6.: WiFi interface power states [77]
2.6. Energy Consumption Prediction Using Palladio

Stier et al. [68] unify the concept of metric based power models with model-based architecture quality analysis. Leveraging the meta-model capabilities of EMF they present a model-based approach to describe power consumption behavior for computer systems and their infrastructure. Their Power Consumption Analyzer (PCA) supports generating predictions on the system’s power consumption using the power model descriptions.

Figure 2.7 shows an excerpt of the three meta-models PowerModelSpecification, PowerBinding and Infrastructure Stier et al [68] present to extend PCM model instances with details on the power consumption behavior of the executing infrastructure.

The Infrastructure model describes power consuming and power providing entities of a technical infrastructure. The model acts as bridge between the ResourceEnvironment and the power-conscious modeling. Power consumption is accounted for on a per-resource level.

PowerModelSpecifications represent concrete power models and specifies their ConsumptionFactors. These factors describe input parameters to the model. The PowerModelSpecifications distinguishes MeasuredFactors and FixedFactors. MeasuredFactors are determined during analyses of the PCM model (e.g. the utilization of a particular resource). FixedFactors describe static characteristics of the particularly represented system (e.g. the idle consumption of the CPU). While the PowerModelSpecifications only specifies the existence of the input parameters concrete FixedFactorValues are bound to corresponding FixedFactors as part of the PowerBinding.

PowerBindings allow to fit a generally defined power model to the device- or resource-specific characteristics. PowerConsumingResources reference the represented ProcessingResourceSpecification in the PCM ResourceEnvironment and the corresponding ResourcePowerBinding.

Differences in hardware design are abstracted by different instances in the InfrastructureModel. If two instances of the same ProcessingResourceType (e.g. CPUs) present the same power consumption characteristics they share a PowerBinding instance. Different
characteristics but similar underlying mechanism to determine the consumption is represented by distinct PowerBinding instance referring to the same power model. Entirely different characteristics can be represented by two distinct power models.
3. Concept and Approach

First responders working in edge environments have to rely on their equipment. Increasing capabilities make modern smart phones a viable alternative to former distinct devices (e.g. camera, navigation unit and radio). Still, mobile devices have to deal with serious resource constraints, limited battery capacity being an important one. Therefore, energy consumption is a non-negligible quality characteristic in application development.

Cyber-foraging applications try to circumvent resource restrictions by leveraging powerful infrastructure available in close proximity. The concept of cloud-offloading is applied, that is, certain parts of the application are run on infrastructure that does not suffer from the same resource restrictions. Energy consumption can be reduced as the device is relieved from executing the offloaded application portions. Instead, the device is required to transmit the necessary input data to the offloaded application elements and receive the results back. Cyber-foraging essentially replaces local computation effort with network effort. Consequently, the offloading decision is reflects a trade-off between reducing local effort through generating a different one.

Taking energy consumption into account when designing applications becomes a logical consequence. Energy consumption of mobile devices is influenced by a wide range of factors (e.g. hardware components in use, connectivity to networks, network conditions). Optimizing applications to reduce the energy consumption requires time-consuming analyses. Prototypes have to be built and resulting consumption measured. A process that is not suitable to scale with the increasing number of devices and their heterogeneous hardware platforms. The ability to reason about offloading decisions based on their energy consumption impact could provide significant benefits for mobile application development.

Model-based approaches allow to decouple determining device energy consumption behavior from the implementation of the application. Architecture level performance analyses have been established to generate predictions on run-time application behavior [6, 31]. Furthermore, power models abstracting the significant device characteristics have been shown to provide usable predictions on power and energy consumption [57, 77, 51, 26, 76, 75]. Combining the two concepts allows to evaluate the energy consumption impact of architectural design decisions at design-time. Still, there is currently no approach that closely integrates the power model determination with energy-consumption-conscious QoS-driven model-based software development processes.

3.1. General Approach

A central design decision for component-based cyber-foraging applications is identifying which components are suitable offloading candidates and which are best run locally. In this thesis I evaluate the applicability of using model-based energy consumption predic-
tions to automatically determine offloading decisions for component-based applications. The approach focuses on automating the entire process from determining power model parameters until generating optimized architecture deployment schemes in order to cope with a wide range of heterogeneous hardware devices.

The entire approach can be structured as a sequence of four consecutive parts. First, the device specific power model parameters are determined (Section 3.2). Second, based on a model-based architecture description of the application a simulative analysis generates predictions on run-time behavior, particularly on computation effort and network activity (Section 3.3). Third, energy consumption predictions are generated using the simulated performance characteristics and the power models (Section 3.4). Fourth, the energy consumption results are evaluated and the architecture description adapted automatically to reflect other deployment schemes (Section 3.5). Step two through four are conducted iteratively to assess each of the resulting architectures and identify the optimal candidates.

This chapter presents concept of this thesis to use model-based energy consumption predictions to automatically optimize software architectures. Single work packages are identified, illustrated and relations between them clarified. The implementation of the concepts is discussed in detail in Chapters 4 through 6.

### 3.2. Automated Power Model Generation

Model-based prediction of software characteristics relies on simplifying abstractions from real-world matters. Determining the right abstraction has a deciding impact on the prediction accuracy. Making the right assumptions and simplifications in a model is the key to reliable prediction results. Particularly, complex hardware devices require to focus on the parameters which have the most impact on energy consumption. Furthermore, in the context of model-based architecture quality analysis the parameter selection has to take into account what information is available or can be determined at design-time.

A common approach to extract power consumption characteristics from real-world infrastructure is to monitor its behavior while being in use. Artificial workload resembling different usage scenarios can be issued to the system in order to speed up the model generation process as distinct usage scenarios can be induced. For power model generation this benchmarking and profiling approach, that is, executing workload while recording the power consumption, is well established [77, 26, 29].

A power model generation approach based on the benchmarking and profiling concept requires the automated execution of artificial workloads while monitoring the consumption of the Device-under-Test (DUT). According to Kistkowski et al. [28] one of the most important characteristics for a benchmark is the degree to which its behavior resembles the one of real-world applications. The more fine-grained artificial workloads reflect application characteristics, the more similar observed power consumption traces should become. Fine-grained traces on the other hand result in benchmark behavior becoming application specific. Power models have to provide predictions using a strong abstraction level in order to be employable in architecture-level analyses. Consequently, there is a significant trade off between accuracy and applicability.
3.2. Automated Power Model Generation

3.2.1. Benchmark Workload

Developing an automated profiling approach for model-based power prediction approaches in heterogeneous hardware environments requires making trade-offs. Existing experiment specification concepts [54, 24] focus on a detailed description of the benchmark execution using a device-independent abstraction level. The concept I am proposing decomposes application workloads by identifying isolated characteristics (e.g. memory-bound workload, CPU-bound workload). The isolated characteristics are then captured as similar micro-benchmarks. When generating artificial system load during experiment execution application characteristics can be represented by a set of such micro-benchmarks. The micro-benchmarks are implemented for each platform using native techniques. Thereby, the micro-benchmark developer is able leverage typical platform-specific features which are would be used in real-world application development as well.

The major advantages compared to including a fine-grained specification of the benchmark operations are:

1. Lightweight experiment specification, as only workload characteristics are specified
2. Micro-benchmarks can be implemented by platform experts, leveraging platform-specific features
3. Facilitate extensibility, as the support of new hardware devices could require significant changes to the micro-benchmark specification.

Honoring the Relevance requirement of benchmarks [28] the overall experiment workload during the profiling experiments is composed from different micro-benchmarks. Therefore, the presented approach makes the implicit assumption that application workload can be represented sufficiently as a probability distribution of selected micro-benchmarks.

3.2.2. System Metric Targets

Power consumption of mobile devices or their hardware components is influenced by a wide range of factors. Power models describe the relationship between measurable metrics of the real-world entities and their power consumption. Nevertheless, power models have to make a trade-off between accuracy (modeling as many metrics as possible) and portability (modeling influences more general) [57].

System metric values represent the state of the monitored device or one of its resources at a certain point in time. Quantifiable relationships between different metrics allow to use metrics which are more easily determined as proxies for other metrics. Rivoire et al. [56] observe, e.g., that there is a linear relationship of the CPU utilization and the power consumption of the system. Consequently, CPU utilization values can be used as proxy from which to derive power consumption predictions.

System metric based power models, as employed by the approach of Stier et al [68], describe relationships between measured system metrics and the resulting power consumption. Usually, the relation can be expressed as mathematic specification on how to calculate power consumption values given the input metrics. In order to identify and
3. Concept and Approach

parameterize suitable relationships experiments have to be executed targeting different metric values while evaluating the power consumption.

Profiling refers to monitoring the device behavior during predefined situations and collecting values for significant metrics. The profiling approach proposed in this thesis separates the description of experiment workload from the system metric targets that are evaluated to increase re-usability. For example, two CPU-bound applications, one generating load the entire time, the other issuing load only during 50% of the time, could be represented by the same CPU-bound micro-benchmark. Therefore, the micro-benchmark execution has to be schedulable in a fine-granular manner. In the example of the two applications, 50% of the time could be represented by defining a time-slice length and executing the micro-benchmark for the first half of it. The SPECpower benchmark [65] describes a formal concept of how to achieve different utilization levels using this approach.

Assessing different metric targets requires the executions of multiple experiment runs. The SPECpower benchmark for example varies the load put on the system-under-test controlled by the throughput metric. After calibrating the maximum throughput the benchmark starts to lower the load from 100% decreasing in steps of 10%. Similarly, I propose specifying the system metric targets and executing an experiment for each target (e.g. system metric: CPU utilization; targets: 10%, 50%, 100%). In order to capture influences of multiple (not necessarily independent) metrics an experiment has to be conducted for every possible combination. Hence, the overall set of experiments is determined by a cross product of all the specified target value sets. Example: The evaluation of Metric A (target values $A_1$ and $A_2$) and Metric B (target values $B_1$ and $B_2$) results in four experiments being executed, one for each of the configurations: $\langle A_1, B_1 \rangle$, $\langle A_1, B_2 \rangle$, $\langle A_2, B_1 \rangle$, $\langle A_2, B_2 \rangle$.

3.2.3. Power Consumption Measurements

The analysis of a device’s power consumption during experiment execution requires taking measurements. There are generally two ways of measuring the power consumption for mobile devices: 1) using device integrated battery sensors and 2) using external power meters.

Device inbuilt sensors do not require additional measuring equipment and can be read using APIs offered by the operating system. There are two major drawbacks to using integrated sensors: limited accuracy and increased effort. Integrated sensors infer power consumption based on changes in battery voltage using a manufacturer-supplied battery-model or determine it by measuring the discharge current [17]. The measurements can be read using Application Programming Interfaces (APIs) provided by the mobile operating system. Querying the API and storing the power measurements in itself creates additional effort on the mobile device and relies on the accuracy of the implemented battery model.

External power meters allow for taking fine-grained measurements without impact on the experiment execution. Furthermore, external power meters provide a higher measuring sample rate (5000Hz [44] vs. 0.5-4Hz [17]). On the other hand, using an external power meter requires probes to be attached to the battery connectors of the Device-under-Test (DUT). The latter is not always easy with many mobile phones today not providing exchangeable batteries.
3.2. Automated Power Model Generation

For this thesis I decided to use an external power meter, as I value an increased measurement accuracy, higher than the additional manual effort necessary to disassemble the device. The resulting experiment scenario is depicted in Figure 3.1.

3.2.4. Raw Power Consumption Analysis

From an architectural perspective the offloading decision is a trade-off between increased network activity and reduced CPU demand. Therefore, the two major power consumer relevant to evaluate in this scenario are CPU and Network device. The purpose of executing experiments while taking measurements of the DUT’s power consumption is to be able to extract power model parameters, in particular for CPU and network device.

CPU Power Consumption

The CPU power model presented in Chapter 2.5.2.1 describes a linear relationship between an input (CPU utilization) and an output parameter (power consumption). Relationships of that kind can be determined using linear regression. The analysis can either be conducted based on the raw power measurement values or using values that underwent a preprocessing step. Conducting regression analyses on aggregated measurements can reduce the processing time. Furthermore, preprocessing measurements beforehand allows to apply pre-conditioning mechanisms to increase the measurement quality.

Although the CPUs of modern mobile devices make intensive use of Dynamic Voltage Frequency Scaling (DVFS) techniques I refrain from taking different frequencies into account, primarily due to time constraints. Thereby, power traces generated using different utilization levels already include DVFS activity as the CPU switches back to lower power states during idle phases.

Hence, the power model that is used for the consumption of CPU and the base system is determined using linear regression based on Equation 3.1.

\[
P_{CPU} = P_{CPU_{\text{min}}} + u_{CPU} \times (P_{CPU_{\text{max}}} - P_{CPU_{\text{min}}}) \tag{3.1}
\]
3. Concept and Approach

WiFi Power Consumption

The WiFi power model proposed by Zhang et al. [77] uses a linear model on top of a state machine. The states are reflecting the current activity state of the resource, transitions in between states are initiated by changes in the activity. They identified the network packet rate as significant parameter, primarily determining the activity state. In this thesis I will approach the WiFi consumption by trying to approximate the power model mathematically. The approximation allows to determine the model parameters using non-linear regression. Thereby, it allows to re-use features necessary for parameter extraction of CPU power models. In section 4.5 I describe an approach to identify the model parameters for a state machine based model as described in the model proposed by Zhang et al. [77].

**Approximation**  In their model Zhang et al. [77] distinguish between a high power and a low power state. The transition is initiated once the packet rate exceeds a certain limit. In the high power state there is an additional linear component scaling the power consumption with increasing transmission rate.

I propose using the two parameters packet rate and packet size as model input and replacing the two distinct power states and the transition in between with the model function in Equation 3.2. While Zhang et al. [77] take uplink channel rate and uplink data rate into account for the architecture-level analysis I assume network conditions to be constant. Therefore, predictions generated with a calibrated model are only valid for network conditions comparable to the calibration scenario.

\[
P_{\text{WiFi}} = C_1 \exp(1/\text{packet size}) + C_2 \exp(1/\text{packet rate}) + C_3 \cdot \text{rate}^C_4 \cdot \text{size}^C_5 \quad (3.2)
\]

The function resulted from taking measurement values and manually conducting different regression analyses using the statistics environment R. The function produces the least squared error among all candidates that were evaluated. It consists of three terms and four model parameters (\(C_1\) to \(C_5\)). The third term allows to be fit into a root function (\(C_4\) and \(C_5 < 1\)) in order to approximate a fast increase, due to the switch from low power to high power state. The curve flattens for larger values, representing a saturation. A saturation naturally occurs as at some point the device reaches a limit of how much data it could transmit. The first two terms are exponential functions of the reciprocal value of packet size and packet rate. The terms are significant for small values of size and rate thereby are able to act as correction factors for small input metric values.

An evaluation of the approximated model is discussed in Chapter 7.1.2.

3.3. Application Resource Demand Simulation

The energy consumption assessment of a distinct deployment scheme based on an architecture-level model requires runtime behavior to be determined at design time. The required system metric values have to be generated in order to employ the power models determined in the previous section. In particular, the linear CPU power model and the non-linear WiFi
approximation model require CPU utilization values and network activity metrics (packet rate and packet size) respectively.

The Palladio approach (see Section 2.3) provides mechanisms to conduct performance analyses based on a PCM specification of the software architecture. The PCM is a domain-specific language which allows to describe component-based software architectures, their execution environment and usage profiles. The performance analysis conducts event-based simulations of the architecture based on the specification and generates predictions for response time and resource utilization.

Components explicitly specify resource demand to ProcessingResource instances in their RDSEFFs. The overall demand to a ProcessingResource instance is determined by all the components allocated to the same ResourceContainer as the ProcessingResource. A resource-specific scheduling policy controls the order and the time for executing the demand. The State of Active Resource metric reflects for a distinct point in time whether the ProcessingResource is actively processing or idle.

Utilization metrics describe for resources the fraction of a time-period the resource has been busy. Consequently, utilization measurements lie between 0.0 (idle the entire time) and 1.0 (busy the entire time). The Power Consumption Analyzer (PCA) presented by Stier et al. [68] provides measures to determine utilization metrics from measurements on resource state. The existing capabilities of Palladio [6] and its extensions for power and energy consumption prediction [68] suffice to generate the system metrics required to evaluate CPU energy consumption.

3.3.1. Modeling Network Demand

PCM instances model resource demand requests as calls to a ResourceInterface. The interface describes the services offered by that resource. It is up to the scheduler of a resource to distinguish between services and to decide how the requests are scheduled. Per-default Palladio Bench comes with support for the ProcessingResourceTypes CPU, HDD and Delay.

The PCM models networks as LinkingResource instances. Each instance references the connected ResourceContainers and represents the entirety of a network. There is currently no ResourceContainer-specific model element representing the connection to a network. Analyses of the demand on a LinkingResource include every network transmission between any two connected ResourceContainers.

Network activity metrics required by the power models have to be determined at a ResourceContainer-level. There are two options of introducing a ResourceContainer-local notion of network demand: 1) add a new ProcessingResourceType instance NetworkController or 2) add NetworkController but also additionally extend the meta-model of the PCM with a modified ProcessingResourceSpecification.

Resource Type: Network Controller Resource

As neither the ResourceInterface of CPU, HDD and Delay resource appear entirely suitable for specifying network demand I introduce the new NetworkInterface similar to the existing HDDInterface (see Figure 3.2). In contrast to the signatures read and write of the HDDInterface
3. Concept and Approach

the `NetworkInterface` provides the resource signatures `send` and `receive`. Network demand can be determined from the PCM model characteristics describing the data size of parameters and return values that have to be transmitted between components. Therefore, network related resource demand is specified as an amount of bytes to be either sent or received.

Similar to the `ProcessingResourceType CPU` and `HDD` which provide the ResourceInterfaces `CPUInterface` and `HDDInterface` respectively, the new `ProcessingResourceType NetworkController` provides the `NetworkInterface`.

![NetworkInterface](image)

**Figure 3.2.: The proposed network interface**

Describing a new `ProcessingResourceType` instance is as easy as creating a new `ResourceRepository` model, adding the desired type and referencing the model in the `ResourceEnvironment` model. The changes can be completed with the EMF model editor only, as no changes to the meta-model or the simulation logic are needed.

**Meta-Model Extension: Network Demand Processing Resource (chosen alternative)**

Adding support for `ResourceContainer`-local network resources only as outlined by the previous alternative has a distinct disadvantage: the network resource is limited to specifying exactly the same properties as `CPU` and `HDD` resource. Therefore, additionally to the realization of the previous alternative, extending the `ProcessingResourceSpecification` offers the capabilities of specifying additional parameters for instances. The extended `NetworkDemandProcessingResourceSpecification` particularly allows to create a relationship between the network resource and the network (LinkingResource).

Evaluating PCM models containing `ResourceContainer`s with multiple `ProcessingResource`s providing the same interface is currently not supported, as resource calls could not be unambiguously assigned. Still, `ResourceContainer`s can be connected to multiple `LinkingResource`s. Modeling a relation between `NetworkDemandProcessingResourceSpecifications` and `LinkingResource` allows to properly simulate the local demand of one network connection.

### 3.3.2. Determining Network Demand

Processing resource demand of a component is assumed to be independent from the current deployment context. On the other hand, the time it takes to process the demand depends on the processing rate and the scheduling behavior of the `ProcessingResource` on the `ResourceContainer` the component is allocated to. Therefore, allocating more components to a `ResourceContainer` leads to an increased resource demand which in turn leads is reflected by a higher utilization.

Network demand cannot be specified independently from component allocations. Figure 3.3 provides an example scenario. Parameters and results of a service invocation are passed between two components (A and B, B and C). Network demand only occurs in case
3.3. Application Resource Demand Simulation

![Figure 3.3.: Example: Network demand depends on component allocations](image)

two directly connected components are allocated to different resource containers (B and C).

**Direct Specification of Network Resource Demand**

The easiest way to include network demands into architectural specifications is having the demand issued by the components where it occurs. Their RDSEFFs would then contain an explicit demand specification similar to Figure 3.4.

![Figure 3.4.: Example: Direct specification of network demand](image)

**Advantage** No extension to the resource demand handling of the simulation or changes to the meta-model necessary. Easy and intuitive specification of the demands.

**Disadvantage** Implicit knowledge of the allocation scheme is necessary for determining the set of components responsible for generating network demand. Additional to changing components allocations PerOpteryx’ automatic generation of candidates would need to change the RDSEFFs in-depth to explicitly specify network demand.

**Adapt Code-Generation Templates**

The SimuCom analysis uses a model-to-text transformation to create executable simulation code based on the PCM instance. Whenever a component calls a different one the simulation code checks whether the two components are allocated to the same ResourceContainer. If that is not the case and a latency has been specified for the LinkingResource the call is delayed for that amount of time. The delay is internally achieved by issuing demand to a ProcessingResourceSpecification of an “invisible” LinkingResourceContainer.

The template generating the simulation code above could be adapted to also determine the size of parameters and return type whenever a cross-network call is issued. The amount of data could then be issued to the NetworkDemandProcessingResource of the container.

**Advantage** Network activity gets determined and recorded transparently. Modification of a single code-generation template file required. No further meta-model extensions necessary.
3. Concept and Approach

**Disadvantage** Solution limited to SimuCom analyses, since other approaches [5, 33] do not use code-generation templates. Furthermore, adapting the templates impacts the validated simulation logic. Thereafter, the validity of the simulatively results cannot be assumed anymore.

**Network Demand Model Transformation (chosen alternative)**

*Component Completions* refers to a concept of extending functional specifications based on additional not necessarily required information [74]. Happe et al. extended the concept to enable support for message-oriented middlewares for Palladio performance analyses [22], later refined by Kapová and Becker [27].

Leveraging a completions approach allows to keep PCM instances free from explicit network demand annotations while having them available during the analysis. The general idea is to transform the model instance during the analysis workflow shortly before the actual analysis is conducted. The transformation has to identify connectors between two components that are deployed to different resource containers. Selected connectors are replaced with connectors to and from a generated proxy component. The proxy component issues network demand to the `NetworkDemandProcessingResource` whenever a service is called and delegates the call to the actual target.

**Advantage** Network activity gets determined and recorded transparently. The completed model is a valid PCM instance that has no particular requirements to the analysis mechanism. No further meta-model extensions necessary.

**Disadvantage** Integrating the transformation requires changes to the simulation workflow.

The first alternative does not seem viable in the context of PerOpteryx’ automatic model adaption as it would require substantial changes to the allocation adaption mechanism. Furthermore, it would contradict the concept of separated roles in the component-based software development process (see Section 2.3), as deployment information has to be available for the specification of component behavior.

Both the second and the third alternative determine network demand transparently. I opted for realizing the completions approach as it does not require simulation changes and keeps modifications on the abstraction level of PCM instances. Furthermore, the model-based representation does not dictate the use of SimuCom as analysis framework.

3.4. Energy Consumption Prediction

The purpose of specifying power models and determining system metric values is the prediction of power measurements which in turn can be aggregated to energy consumption predictions. The Power Consumption Analyzer (PCA) presented by Stier et al. [68] relies on *calculators* to determine power measurements based on input metric values. So far, the PCA requires the developer to programmatically implement a calculator for every `PowerModelSpecification` instance.
Conducting regression analyses depends on a formalized representation of the power model’s underlying calculation description. A common principle of software engineering (Don’t Repeat Yourself (DRY) [25]) discourages the maintenance of the same logic in distinct locations or representations. Therefore, I employ a unified representation (Section 4.4.2.1) allowing to specify a mathematical expression of the power model as part of the PowerModelSpecification instance. The unified representation allows its use as regression analysis input (Section 4.4.2.5) as well as to be evaluated in the PCA (Section 5.2.2). In order to be fully backwards compatible, lightweight and as non-invasive as possible the mathematical expression is specified as part of the PowerModelSpecification instance’s name.

The PCA relies on an EvaluationContext for post-simulation analysis that determines the utilization metric values for ProcessingResources. The EvaluationContext acts as iterator over the utilization values of all resources to evaluate and is used to determine the parameters to feed into the calculators. Currently, the EvaluationContext is limited to handling one utilization metric per resource. Although providing utilization measurements is fine for predicting CPU power consumption the proposed WiFi model requires different metrics (packet size and packet rate). Since there is currently no support for using metrics other than utilization the EvaluationContext has to be adapted.

Similar to the utilization measurements the network packet rate also has to be determined over time. Therefore, the UtilizationFilter can be reused instead of implementing similar aggregation functionality. Utilization metric values for the network device can be converted to packet rate measurements using Equation 3.3. \( \text{rate}_{\text{nominal}} \) refers to the processing rate specified in Bytes/s for the network resource.

\[
\text{rate}_{\text{packet}} = u_{\text{WiFi}} \times \text{rate}_{\text{nominal}} \times \frac{1B}{\text{size}_{\text{packet}} + \text{overhead}_{\text{packet}}}
\]  

(3.3)

Deriving packet rate from utilization requires additional metrics describing the WiFi controllers processing rate and packet characteristics size and overhead. The metrics that have to be supported additionally are specified more general and summarized in Table 3.1.

Leveraging the support for the new metrics the following changes are necessary in order to provide the required capabilities. The changes will be discussed in Chapter 5.2.1.

1. Decouple utilization measurement generation from EvaluationContext to support different metrics (Section 5.2.1.1)
2. Provide extension mechanism for other metrics than utilization (Section 5.2.1.2)
3. Allow for chaining of multiple providers (e.g. State Of Active Resource \( \Rightarrow \) Utilization \( \Rightarrow \) Processing Rate)

While the required adaptions extended the power prediction capabilities of the PCA with respect to the acceptable input metric, there is no change to the output metrics. Therefore, the provided implementation integrating the power consumption measurements over time does not need to be modified and can be applied.
3. Concept and Approach

<table>
<thead>
<tr>
<th>Metric</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized Processing Rate of Active Resource</td>
<td>The processing rate, initially in the ResourceEnvironment model. Varying during simulation if support for DVFS is included.</td>
</tr>
<tr>
<td>Normalized Processing Rate of Active Resource Tuple</td>
<td>Processing rate measure including a point in time measurement of the measurement time</td>
</tr>
<tr>
<td>Processing Rate of Active Resource</td>
<td>Processing rate with respect to processing unit characterization</td>
</tr>
<tr>
<td>Processing Unit Size</td>
<td>Size of unit that is processed at once by ProcessingResource</td>
</tr>
<tr>
<td>Processing Unit Characterization</td>
<td>Overhead necessary to process an unit</td>
</tr>
</tbody>
</table>

Table 3.1.: WiFi-related metric extensions

3.5. Automated Architecture Optimization

The goal of this thesis is to automatically determine deployment-schemes that optimize the energy consumption of mobile applications in a cyber-foraging context. Leveraging the concepts presented in the last section allows us to generate energy consumption predictions based on component-based architecture specifications. Given a set of architecture design decisions, in particular a set of deployment options, it would be possible to create a model instance for each of the decisions, conduct a simulation and calculate the energy consumption. Ranking all alternatives then identifies the best choice.

This approach carries an inherent problem: the number of model instances to create manually and conduct simulations on increases exponentially with every degree of freedom. For example, give 4 components each having 2 deployment options already results in $2^4 = 16$ different simulations to conduct.

The logical consequence is to apply an automated approach. PerOpteryx provides automation capabilities for the Palladio analyses, including the required optimization logic (see Section 2.4). It’s Degrees Of Freedom model already provides the necessary capabilities to describe for a set of components allocation options to different ResourceContainers. PerOpteryx also is able to alter model candidates in the required way (change the ResourceContainer reference of components’ AllocationContexts). No further model changes are necessary as changes in network demand are resolved by the completion approach as outlined in Section 3.3.2.

3.5.1. Quality Criteria Specification

Energy-conscious architecture optimization using an automated optimization framework requires at first being able to specify the optimization goal. PerOpteryx employs a EMF-model representation of the QML language. In order to specify Energy Consumption as optimization goal it is defined as additional (Quality-)Dimension. The architecture optimization is conducted with the following quality specifications in textual QML representation.
Measurements of the energy consumption dimension come as floating point (numeric) values specified in Watt-hours. Smaller values are better (decreasing).

```
type EnergyConsumption = contract {
  energy consumption : decreasing numeric W*h;
};
```

The optimization is conducted based on the mean energy consumption determined over multiple executions. There is no constraint to the mean consumption (< INFINITY).

```
EnergyOptimizingContract = EnergyConsumption contract{
  energy consumption { mean < INFINITY };
};
```

The architecture quality is assessed based on the energy consumption determined for the UsageScenario DefaultScenario specified in the UsageModel DefaultUsageModel.

```
EnergyOptimizingProfile for DefaultUsageModel = profile {
  from DefaultScenario require EnergyOptimizingContract;
};
```

### 3.5.2. Optimization Mechanism

PerOpteryx’ optimization mechanism revolves around the Opt4J optimization framework\(^1\). By-default it uses an evolutionary algorithm that generates new architecture candidates either randomly based on the Degrees of Freedom model instance or by combining two existing ones.

The EnergyOptimizingContract requirement for the DefaultScenario leads PerOpteryx to expect the existence of an analysis mechanism for all described quality dimensions. Therefore, providing PerOpteryx with energy consumption evaluation capabilities requires the implementation of an additional adapter to the energy consumption analysis. Analyses integrate with PerOpteryx using the Extension Point mechanism for Eclipse plug-ins.

Figure 3.5 shows two simulated metrics an arbitrary architecture specification. There are two ways of generating an evaluation result with respect to energy consumption:

1) Take the last cumulative energy consumption measurement for the entire experiment. This approach requires only energy consumption metric values (Figure 3.5b).

\[
\text{Result}_{\text{Analysis}} = E(t_{\text{end}}) - E(t_{\text{start}})
\]

2) Take response time into account and determine a statistical aggregation of the amounts that were consumed during each UsageScenario execution.

\[
\text{Result}_{\text{Analysis}} = \text{Mean}_{rt\in \text{ResponseTimes}} (E(t_{rt_{end}}) - E(t_{rt_{start}}))
\]

\(^1\)\url{http://opt4j.sourceforge.net/}
3. Concept and Approach

There are arguments to both alternatives. It depends on the actual application scenario to identify a preferable one. For example, a long running cyber-foraging application which regularly sends updates (e.g. a first responder’s position to coordinate with others) to the cloudlet would benefit from the first alternative. In this case the actual amount necessary for transmitting data once is not as significant as the amount of energy consumed over a long period of time. The long running process can be well reflected in a time-bound experiment execution.

On the other hand, with an application that e.g. captures a photo, sends it to the cloudlet to be analyzed and transmits back the results the amount of energy per service request carries more weight. Even more, the first alternative is likely to lead to false positives. Given two design options, one leading to very little local resource demand but taking a long execution time, the other generating higher local demand but shortening the execution period drastically, the first analysis alternative would probably favor the first option, the second analysis alternative the other. The problem at hand is that for the first analysis alternative there is no difference between the time the device is idle during a service request (while waiting for results) and the idle time in between requests (a.k.a. think time).

As various cyber-foraging scenarios pursue result-oriented tasks (speech/face/object recognition) I opted for realizing the second alternative. With little effort the first alternative can be included additionally.
4. Automated Device Power Profiling

Heterogeneous hardware landscapes and frequent hardware platform changes present difficulties for energy consumption analysis as every new device generation potentially presents different consumption behavior. The goal of an automated device profiling approach is to facilitate the generation of device specific power models.

Power consumption profiles can be determined using a benchmarking approach. Nevertheless, benchmark executions require time-consuming experiments on the device while system parameters, particularly the energy consumption, are monitored. In this chapter I present an automated profile generation approach assisting in the creation of power models suitable for model-based energy consumption prediction.

The general idea of the profiling approach is to execute experiments on the mobile device while monitoring its power consumption. Every experiment is characterized by its workload, that is a set of micro-benchmarks stressing certain hardware components. The workload execution is controlled through a set of target system metrics for every experiment. Analyzing target metrics and power consumption measurements allows to derive power model parameters.

The presented profiling approach can be structured into three phases. During the Experiment Description Phase the relevant hardware components of the mobile device are identified for which power model parameters should be determined. The device’s composition is reflected in the PCM ResourceEnvironment and the PCA’s Infrastructure model. For each hardware component settings and goals for the profiling experiments are formally specified in an instance of the BindingModelGeneration model. The specification is further discussed in Section 4.1.

During the Experiment Execution Phase the specification is transferred to the mobile device and executed by a profiling application. The power consumption of the mobile device is recorded and a report containing the power measurements is generated after the execution finishes. The profiling application concept as well as the concept of monitoring the device is described Section 4.2.

In the Evaluation Phase the measured values are analyzed and the device specific parameters are determined based on a generic power model specification. Section 4.3 presents how preprocessing using an architecture of pipelines and filters can facilitate the power consumption analysis. The extraction of power model parameters is topic of Section 4.4.

4.1. Experiment Specification

The approach for specifying profiling experiments presented in this chapter is build upon the Eclipse Modeling Framework (EMF). EMF was selected a viable basis since all models in the context of the PCM including the ones of the PCA [68] are built upon it. Furthermore,
4. Automated Device Power Profiling

EMF allows to generate Java code and Eclipse-based editors for handling model instances based on an Ecore meta-model specification. The model-based specification approach allows to integrate the experiment target specification with the existing models. In particular, it allows to reference Infrastructure model entities, notably PowerProvidingResources and PowerConsumingResources.

Leveraging EMF takes care of generating a platform-independent experiment description since meta-models as well as models are per-default persisted as XMI files. Therefore, experiment specifications generated with the Eclipse-based editor are readable on any platform providing support for handling XML documents. The EMF library manages XMI file parsing, inter-model reference resolving and the Java object instantiation on supported platforms. The core library and generated code (except for GUI elements and Eclipse editors) is compatible with the Android platform.

4.1.1. The Experiment Specification Meta-Model

![BindingModelGeneration meta-model diagram]

Figure 4.1.: Excerpt from the BindingModelGeneration meta-model

Figure 4.1 shows an excerpt from the meta-model of the profiling experiment specification. The name BindingModelGeneration refers to the BindingModel as the model which specifies device specific values for abstract power models.

The root element of every BindingModelGeneration model instance is a BindingModelGenerationContext. The purpose of the root element is to allow the specification of multiple experiments for different devices as part of one model instance to facilitate the deployment across multiple devices.

Device-specific experiments are specified by DeviceProfilingExperimentContext instances. Every instance references a MountedPowerDistributionUnit in the Infrastructure model. The MountedPowerDistributionUnit instances represent power-conscious PCM ResourceContainer instances in the Infrastructure model.
4.1. Experiment Specification

The PCA approach [68] determines power consumption at the ProcessingResource level. Consequently, in order to provide power consumption predictions the profiling approach has to determine power model parameters for each resource specific power model. Hence, DeviceProfilingExperimentContexts contain experiment descriptions for each PowerConsumingResource which should be profiled.

The ResourceProfilingExperimentContext represents a set of experiments that control the device behavior to analyze particular hardware component’s behavior. Each ResourceProfilingExperimentContext is characterized by three independent configuration dimensions:

WorkloadProfile   Specification of experiment workload characteristics as a set of micro-benchmarks reflecting basic elements of real-world application workloads.

ExperimentDimension Specification of target system metrics that are controlled during the experiment execution and the declaration of target values for the distinct experiments.

EnvironmentSettings Device specific configuration parameters which are not supposed to be taken into consideration for power model generation. The configuration parameters are specified as textual key-value pairs.

4.1.1.1. Workload Profile

Workload instances represent micro-benchmarks which are characteristic for certain aspects of application workloads. Each workload instance is a model representation of the task that is to be executed. Three different workloads are currently supported by the proof-of-concept I developed for the Android platform: CalculatePI, SieveOfEratosthenes and SendRandomData.

The first two represent micro-benchmarks for associated with ProcessingResourceType CPU. Calculating \( \pi \) to a given precision can be implemented entirely CPU-bound with little to no accesses to the memory. It represents CPU load resulting from intensive calculations. The Sieve of Eratosthenes is a memory-bound algorithm that efficiently determines all prime numbers up to a specified number. The algorithm allows to represent programs prone to memory accesses.

Network demand is simulated by sending an array of a fixed size via TCP to a server. The amount of data is controlled by the system metric Processing Unit Size. Due to time constraints other network protocols or receiving of data has not been realized.

As tasks are most-likely neither purely CPU-bound nor entirely memory-bound WorkloadProfiles describe the synthetic experiment workload as composition of multiple micro-benchmarks.

ProbabilityDistributionWorkloadProfile instances describe a set of Workload instances as a probability mass function. Every Workload is associated with its probability to be issued for execution. Consequently, the ProbabilityDistributionWorkloadProfile contains an arbitrary number (>1) of WorkloadProfileElements, each specifying a probability and referencing a Workload instance.
4. Automated Device Power Profiling

4.1.1.2. Experiment Dimensions

Creating device specific power model parameters requires analyzing the observed power consumption for different sets of system metric values. Therefore, an essential part in the experiment description is defining the system metrics that are varied throughout the execution.

The excerpt from the ExperimentSpecification model in Figure 4.2 shows the relevant model elements to specify system metric targets. The ExperimentDimension element represent a dimension which offers variation possibilities during experiment execution. SystemMetricExperimentDimension instances represent experiment variations specified by a distinct MetricDescription instance. For system metrics representing a continuous spectrum of values the ContinuousValuesSMED type allows to specify an upper and a lower bound (e.g. utilization values lie between 0.0 and 1.0). DiscreteValuesSMED refer to metrics which are defined only for a set of discrete values (e.g. CPU frequencies for DVFS).

The distinction between continuous and discrete dimensions is motivated from the power model generation perspective. Continuous values require a power prediction function that is continuously defined between the lower and the upper bound. Discrete values on the other hand translate to a distinction of cases (cf. CPU frequency in Chapter 2.5.2.1).

The abstract DiscreteValueSeries type represents elements that can be treated as a series of metric target values. The distinct metric values are encapsulated in DiscreteValueSetter instances. In order to simplify the specification of experiment targets for continuous metric dimensions the DiscreteApproximatedContinuousValuesSMED is introduced. It allows to specify explicit target values while keeping the notion of a continuous values dimension.

![ExperimentDimension meta-model](image-url)

Figure 4.2.: ExperimentDimension meta-model
4.2. Experiment Execution

The nomenclature in this section will be aligned to the one used by Experiment Data Presentation and Persistence (EDP2) of the Quality Analysis Lab [38]: ExperimentGroups contain ExperimentSettings, which in turn contain ExperimentRuns. In the context of the mobile device profiling application their respective meaning will be as follows.

**Experiment Run** Exactly one experiment execution during which a set of system metric target values, specifying exactly one value per metric, were kept constant.

**Experiment Setting** All profiling experiments that are conducted with the same target values for all system metrics under variation are subsumed under the term *Experiment Setting*.

**Experiment Group** All profiling experiments and measurements regarding one device are subsumed under the term *ExperimentGroup*.

4.2.1. Experiment Execution Scenario

The experiment scenario is depicted in Figure 3.1 in Chapter 3.2.3. The experiment specification as presented in previous section controls the experiment execution behavior. The DUT runs the *AndroidProfiler* application while being attached to an external power monitor. The power monitor is controlled and measurements are stored by the *ProfilingServer*.

The power measurements are taken with a power monitor manufactured by Monsoon Solutions Inc.¹

¹https://www.msoon.com/LabEquipment/PowerMonitor/

The power monitor allows to manually control the measurement procedure using the provided *PowerTool*. In order to automate the process *PowerTool* can be controlled via an automation API. Monsoon Solutions Inc. recommend building application which use the API on top of the .net-Framework, like the *PowerTool* itself. Therefore, I implemented the *ProfilingServer* as standalone C#-application.

There are different ways of assigning responsibilities between *ProfilingServer* and *AndroidProfiler*. In particular, the experiment specification can be evaluated by both applications, *Profiler* and *Controller*. There are two conceptually different execution scenarios: server-controller and device-controlled.

**Server-Controlled Execution** The *ProfilingServer* parses the ExperimentSpecification and instructs the AndroidProfiler to execute the experiment runs appropriately. System metric targets and workload specifications have to be transmitted explicitly requiring the server to keep track of the currently running experiment settings and future ones to come. Furthermore, devices are required to keep a connection to the ProfilingServer to be able to receive commands on how to proceed execution.

A server-controlled scenario allows to easily identify power measurements and assign them to the corresponding experiment settings, as the server is aware of the time each experiment run starts and ends.
Device-Controlled Execution (chosen alternative) A device-controlled execution allows the DUT to perform multiple experiment runs without the need to be connected to the server the entire time. Independent execution on one device while generating measurements on another requires a synchronization of the clocks on both devices. After synchronizing the clock and starting the power measurement process the device can disconnect from the server and act completely independent.

The ProfilingServer can be kept very lightweight as it does not have to be agnostic of different workloads or the target metrics to generate power traces of the experiments.

The ability to conduct experiments independent from being connected offers extended benchmark capabilities, e.g. deactivating the WiFi controller entirely while running CPU benchmarks. Furthermore, the increase of application complexity due to handling the experiment specification is smaller for the AndroidProfiler than the ProfilingServer as the AndroidProfiler already has to be aware of the experiment specification details (e.g. different micro-benchmarks and system metrics).

The self-controlled experiment execution on the mobile device requires the AndroidProfiler to notify the server of the device experiment’s begin in order to start taking measurements. Thereafter, the AndroidProfiler is required to keep record of changes to profiled resources and metric targets including their point in time as the server is unaware of them. After the entire device profiling experiment is concluded the power measurements corresponding to the distinct experiment runs can be extracted using the record kept by the AndroidProfiler. The clocks between ProfilingServer and AndroidProfiler have to be synchronized in order for the recorded points in time to be significant for the ProfilingServer (details are presented in Section 4.2.2.4).

Using a request/response-based protocol for the communication between AndroidProfiler and ProfilingServer allows a lightweight service-oriented server reducing the effort of state keeping in the server. Therefore, the ProfilingServer is designed to offer a set of services which are requested explicitly and generate a response. As the experiment execution is controlled by the mobile device there is no need for a bidirectional request initiation.

The protocol has to take the following aspects into account and provide concepts and/or messages accordingly:

1. Time synchronization
2. Signal start and end of power measuring
3. Request for a distinct network channel for network related benchmarks
4. Transmit timings for experiment runs

Encapsulating the ProfilingServer tasks into independent services allows to simplify the protocol structure. AndroidProfiler and ProfilingServer exchange Payload messages. This message type provides means to identify the desired service, specify additional parameters and/or transmit further data.

Power traces are recorded by the ProfilingServer. The task of generating the final report that contains power measurements is best allocated to the server. Generating the report
on the mobile device would involve transmitting the power measurements to the phone and thereafter, sending the entire report to the server as the analysis is conducted on a standard computer.

Keeping the server lightweight and unaware of experiment specifics is achieved by using a template approach. After all experiment runs are executed and the AndroidProfiler requested the server to stop taking power measurements a report of all experiments runs is transmitted (DeviceProfilingMeasurementResults). The AndroidProfiler creates the report and fills it with all analysis relevant information available on the mobile device. Based on the timestamps for start and end of experiment phases included in the report the server inserts its measurements accordingly.

The messages are specified using Google Protocol Buffers\(^2\) and transmitted using the low latency messaging framework zeroMQ\(^3\). Transmitting the serialized Protocol Buffers messages via a raw TCP stream proved unstable, as it did not handle network connection loss very well. zeroMQ abstracts from the underlying TCP layer and provides lightweight means of a longer running connection. During periods when the ProfilingServer is not available (e.g. due to the WiFi device being deactivated) zeroMQ buffers the messages.

4.2.2. Mobile Device Application

Heterogeneous hardware platforms and a high frequency of changes are characteristic for mobile platforms. Therefore, any application which focuses on assessing hardware properties has to provide extensibility measures to prevent itself from becoming obsolete very fast. In order to leverage the flexible, extensible and model-based experiment description concept presented in the previous section the profiling application is structured into loosely coupled modules which can be extended with new capabilities. The architecture strictly follows the Separation of Concerns (SoC) concept. SoC requires every architectural entity to focus on one task and separate non-related tasks into distinct entities.

In order to achieve the desired flexibility and to facilitate extendability and exchangeability the AndroidProfiler relies on the two architectural patterns Event-driven architecture and Inversion of Control.

4.2.2.1. Architectural Principles

A general approach to achieve lose coupling in a system is encapsulate application logic into distinct functional units which interact with each other only through interfaces. Specifying interfaces independently allows to exchange one implementation with another. Although, in object-oriented development, interfaces decouple interacting classes from depending on a particular implementation they still need to instantiate a concrete type.

The Inversion of Control (IoC) principle aims to relieve classes from the duty to resolve their dependencies, as only then the class is completely decoupled from concrete types. The Dependency Injection pattern is very common way of achieving IoC, particularly in the Java world [18]. Every class get provided with concrete instances of its dependencies while the implementation specific code is accumulated at few central locations. Generally,
the task of instantiating dependencies lies within the responsibility of a Dependency Injection (DI) container which is usually provided by a DI framework. The popularity of the pattern is reflected by a wide range of available frameworks providing Dependency Injection capabilities.

The AndroidProfiler additionally relies on the Event-Based Architecture pattern to further reduce coupling among participating application parts and to facilitate interaction. Events can be characterized as "significant change in the state of the universe" [11]. Events are generated by a producer, routed through channels and consumed by an arbitrary amount of listeners.

Leveraging the event-based pattern allows to modularize the experiment execution logic and properly separate cross-cutting concerns, e.g. the recording of system metric target changes or the start and end of experiment runs. The AndroidProfiler relies on the EventBus framework\(^4\), a lightweight implementation of a publisher-subscriber architecture designed for the Android platform.

EventBus applies a concept that Lazar et al. [36] identified as Convention over Configuration. In order for a class to be able to be notified upon distinct events it needs to provide a handler method. The name of the event handler method controls the thread on which the handler is called (e.g. blocking the publisher thread or asynchronous delivery in the background).

When determining appropriate handler method EventBus takes inheritance hierarchies into consideration. If a class specifies multiple suitable event handlers the most specific one is called. Therefore, every entity can attach to exactly the relevant events. For example, a logger for debugging purposes can trace every event by specifying an event handler for events of the type `Object`.

### 4.2.2.2. Device Profiling Experiment Concept

The behavior of the AndroidProfiler application is characterized by the abstract workflow depicted in Figure 4.3. The workflow is influenced by the concept of processing the ExperimentSpecification and executing the profiling experiments for the declared resources. The distinct workflow states encapsulate the experiment execution behavior. Transitions from one state to another are initiated event-based.

![AndroidProfiler workflow states and transitions](https://github.com/greenrobot/EventBus)

---

\(^4\)https://github.com/greenrobot/EventBus
The Workflow  Until initialized the application remains in the UninitializedState. Initialization is currently conducted by the user loading a valid ExperimentSpecification file and selecting the appropriate DeviceProfilingExperimentContext. Employing further automation this step could be initiated by a central management application in order to speed up the process for multiple devices.

During the DeviceProfilingExperimentInitialization state a queue of the resource-specific sets of experiments is built up. The ResourceProfilingExperimentInitialization state instantiates the profiling experiment infrastructure. In particular, the initialization parameterizes the ProfilingExperimentRunner with the queue of experiment settings that is determined from all possible combinations of system metric targets (ExperimentDimensions).

The ResourceProfilingExperimentExecution state behavior consists entirely of letting the ProfilingExperimentRunner, which is detailed further in the next section. During the ResourceProfilingExperimentFinished state the application decides whether another resource is supposed to be profiled or if the device profiling experiment is finished.

The workflow represents the coarse application behavior and is designed to facilitate flexible extension. The transitions between WorkflowState instances act as extension points for additional functionality to hook in. The feature is particularly used to attach notification to the ProfilingServer, time synchronization and logging initialization to the workflow. The extensible workflow concept is discussed in further detail in Section 4.2.2.4.

4.2.2.3. Resource Profiling Experiment Run

Analysing the power consumption behavior of separate hardware components of the DUT requires monitoring them isolated from other influences. With the exception of completely modular devices [10] there is no way to measure the consumption of an embedded system’s parts. Therefore, resource profiling experiments are required to measure the consumption of the entire device and infer the influences stemming from the particular resource.

The execution of a ResourceProfilingExperiment leads to conducting an experiment run for every possible combination of SystemMetricExperimentDimension values (as described in Section 3.2.2). During each experiment run the corresponding set of metrics is kept constant over the entire time.

The excerpt of the AndroidProfiler architecture depicted in Figure 4.4 is responsible for generating the benchmark workload while regulating the device behavior.

A central entity of the architecture is the WorkloadDriver. The driver creates resource demand in the form of issuing work to concrete micro-benchmark instances while monitoring and regulating selected system metrics. The concept of a similar WorkloadDriver is presented by the SPECpower benchmark [65] in order to generate different levels of server system utilization.

The separation of system metric targets and workload characteristics is reflected in the presented architecture similar to the separation of the two aspects in the experiment specification (see Section 4.1.1).

Workload Characteristics  Characteristic aspects of application workloads reflecting a particular significant hardware demand are represented by micro-benchmarks. The bench-
marks strive to concisely recreate the same hardware demand patterns. References to all available workload types are stored in a WorkloadRepository which resolves concrete instances based on the experiment specification.

The micro-benchmark logic is separated from the generation of input parameters and the verification of results. WorkloadParameterGenerator instances can initialize workload parameters before the actual experiment execution to avoid influencing the hardware behavior (e.g. generation of an array of random data to be sent via WiFi). Results determined during the execution of the benchmark (e.g. an approximation of \( \pi \)) can be validated by a distinct entity to ensure a properly functioning benchmark and prevent compile-time or run-time optimization from removing unused operations (e.g. calculation without using the result).

Workload instances are bundled together with suitable a WorkloadParameterGenerator and a WorkloadResultVerifier to a WorkloadTriple. A triple provides the necessary entities to execute the contained micro-benchmark in order to generate artificial load.

Three different micro-benchmarks have been implemented as part of this thesis: 1. Calculation of \( \pi \) approximation using the Leibniz formula, 2. Determining of all prime numbers to an upper limit and 3. the transmission of a fixed amount of random data.

Calculating \( \pi \) using the Leibniz formula
\[
\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \cdots = \frac{\pi}{4}
\]
is not necessarily efficient in the sense of convergence but effective in keeping the CPU busy. The duration of the workload execution can be varied by the amount of iterations conducted.

The SieveOfEratosthenes creates an array of prime number candidates and starts removing non-prime numbers. Similar the controlling the \( \pi \)-benchmark by the maximum number of iterations the execution time of the Sieve of Eratosthenes depends on the highest number to be evaluated.

The SendNetworkData benchmark establishes a separate TCP connection to the ProfilingServer. The channel is already created during the benchmark initialization in order to prevent from influencing the observed power trace during the experiment run. Upon execution the benchmark transmits an array of random binary data (generated beforehand) to the server. The execution time can be influenced by specifying different amounts of data to transmit.
4.2. Experiment Execution

**Workload Profiles** Application behavior is approximated by composing the benchmark workload from multiple micro-benchmarks. The WorkloadProfile specification is represented in the AndroidProfiler architecture by concrete implementations of the WorkloadSelector interface. The Selector is initialized based on the experiment specification and thereafter, can be queried for the next WorkloadTriple to be executed.

ProbabilityDistributionWorkloadProfiles represent a set of Workloads together with a fixed probability of selection. The concrete WorkloadSelector implementation provided for the probability distributions initializes a probability mass function based on the specified probabilities. At runtime a sample of the function is taken every time a new workload is requested and the associated WorkloadTriple returned.

**System Metric Targets** System metrics are numerical values observed during execution that describe the current state of a system. Manually inducing metric targets in order to analyze the impact on the power consumption requires influencing the benchmark execution. There is no general approach to handling all metric targets as dependent on the metric type different execution aspects or system parameters have to be changed.

The WorkloadProfiles concept allows to describe the kind of demand issued by applications, system metrics can be observed during execution. The metrics often depend on different factors in the system, e.g. other applications running in parallel. Different system metric levels can be induced by controlling the behavior of how micro-benchmarks are executed.

WorkloadDriver instances encapsulate the behavior of determining the next micro-benchmark (using the WorkloadSelector) and executing it in a particular manner. Depending on the system metrics that are supposed to be varied a different behavior is necessary. Resources for which the describing power models rely on utilization metrics have to be controlled in a manner that the resource activity over a preset time frame reflects the target utilization levels. It is not relevant which micro-benchmark is running at which point in time as long as an appropriate overall ratio between active and idle times is achieved. The WiFi power model relies on a packet rate metric. Distinct rates can be induced having the micro-benchmark send out one packet and control the rate with which micro-benchmark is invoked. The AndroidProfiler provides two concrete implementations to reflect the two WorkloadDriver behaviors:

**UtilizationControllingWorkloadDriver** The driver implements the behavior similar to the one specified by the SPECpower benchmark [65]. Target utilization values are achieved with respect to a configurable time-slice length. Micro-benchmarks are executed for the fraction of the time-slice that corresponds to the utilization value. The driver retrieves the next WorkloadTriple, queries the parameter generator, invokes the micro-benchmark and verifies the results while not more than the desired time-slice fraction has passed.

**ProcessingRateControllingWorkloadDriver** The driver implements a single execution behavior and allows to control the rate with which micro-benchmarks are invoked. The length of a time-slice is determined based on Processing Rate metric targets. Each time-slice execution consists of one iteration of selecting next workload, retrieving...
4. Automated Device Power Profiling

parameters, executing workload and verifying results. This driver is implemented in particular to control the rate with which data packets are sent to the server in the SendRandomData benchmark.

**Resource Profiling Experiment Runner** The WorkloadDriver schedules micro-benchmark execution in order to achieve certain metric targets (*Utilization*, *Processing Rate*) to create a distinct system state synthetically. The generation of artificial system load is decoupled from the experiment control logic. While the WorkloadDriver continuously executes the benchmarks in the same pattern the ResourceProfilingExperimentRunner encapsulates the notion of time-bound experiment runs.

The Runner is initialized with the ResourceProfilingExperimentContext during the ResourceProfilingExperimentInitialization workflow state. It stores a queue of system metric settings that have to be executed as distinct experiment run. The queue is built based on the SystemMetricExperimentDimension instances specified for the resource to profile.

The ResourceProfilingExperimentRunner is responsible for enforcing the experiment structure, that is, parameterizing the WorkloadDriver to run for the duration of an experiment run with constant metric targets. Depending on the type of the profiled resource or the analysis that is conducted on the measurements every experiment run has to consist of different phases.

Instantiating experiment executions, preparing the micro-benchmarks and in some cases notifying the ProfilingServer can generate significant load on the system. The *Idle Phase* is a period of time between the initialization is finished and the WorkloadDriver is started, to allow the system to calm down.

Measuring the power consumption for the entire device during benchmarking different resources results in the device’s idle consumption being measured multiple times. In accordance with Rivoire et al. [55] who relate entire server’s consumption to the CPU utilization I regard the idle consumption as caused by the CPU resource. Therefore, for any resource profiled separate from the CPU the measured power consumption has to be reduced by the consumption generated not by the resource activity itself. During the *Calibration Phase* the WorkloadDriver works exactly as normal, except for the missing invocation of the micro-benchmark.

The *Load Phase* captures the normal execution of the WorkloadDriver.

A fourth phase is designed for analyzing the time a resource stays in an elevated power state before returning to the idle consumption. This *Cool-down* phase is particularly designed for long running data transfers.

4.2.2.4. Extensible Workflow

**Extension: Timing Provider** Power measurements recorded by the power meter are stored on the computer running the Profiling Controller application. While the power consumption is continuously measured, the mobile device executes separate experiments for different system metric targets. Matching observed power consumption behavior to a certain set of system metric values is a fundamental to the model parameter extraction. In order to be able to match timestamps generated by different devices it is inevitable to employ a time synchronization mechanism.
Cristian’s algorithm [15] is an established mechanism for low latency networks. The underlying assumption of the algorithm is that the round-trip time is split half, ergo sending the request to the server takes the same amount of time as sending the response to the mobile device. The assumption does not necessarily hold for wireless networks. Nevertheless, round-trip times measured in an experimental set-up were in the vicinity of $< 10\text{ms}$. Therefore, the resulting worst-case error seems a tolerable error which can even be counteracted by trimming the measurement profiles accordingly (discussed in Section 4.3.1).

In situations with very long running experiments and strongly diverging clocks the plugin can instead be attached to the ResourceProfilingExperimentInitialization state to re-synchronize the clocks between resource profiling experiments.

**Extension: Result Generation** The purpose of employing a central event bus is the ability of objects to register for and be notified of relevant events without having to manage a queue of observers for each potential event creator. In particular keeping a protocol of experiment-execution-determining events is as easy as attaching a ResultEventLogger object to the event bus.

The ResultEventLogger is configured with a high priority and gets the events delivered on the same thread where it is posted (onEvent-event handler). The event handler is called in a blocking manner, that is, the event publishing operation does not return before the event handler is executed to accurately record the points in time. In order to reduce latency the delivered events are just stored in a buffer providing enough capacity to be processed after the experiment execution is finished.

The ResultEventLogger is a workflow plugin itself and hooks to the DeviceProfiling-ExperimentInitialization and the DeviceProfilingExperimentFinished state. Upon the entry to former it registers itself to the EventBus, thereafter being notified of any Result-GeneratingEvent. When entering the latter, it de-registers itself again and processes all events that have been buffered in the mean time.

ResultBuilder instance abstracting from the result message build-up process is handed to each event, in chronological order. Every event that is relevant to generating the resulting report implements the interface ResultGeneratingEvent. The described acceptResultBuilder(ResultBuilder builder) method represents a mixture of the Builder pattern and a lightweight Visitor pattern, that is, the behavior of how to handle a specific subclass is delegated to the class. The resulting message, built using Protocol Buffers, is serialized and sent to the ProfilingServer. It contains points in time experiment runs started and finished, and for each experiment run the configuration of system metric values. At the time of serialization the ResultTimestamps referencing both, local and remote time, are reduced to storing the server time only.

**4.2.3. Profiling Server**

Although the experiment execution as presented in the previous chapter is able to execute independently, some tasks cannot be performed on the mobile device. In particular, reading measurements from the external power monitor has to be performed by a computer running Microsoft Windows.
4. Automated Device Power Profiling

The extensible concept of the AndroidProfiler is reflected in the architecture of the server as well. Therefore, the same architectural principles, event-based architectures and dependency injection, are employed. While the AndroidProfiler controls the experiment execution workflow the ProfilingServer is designed to react upon requests.

Similar to the EventBus in the AndroidProfiler, the ProfilingServer employs an EventAggregator to realize the publisher/subscriber principles. The basic architecture consists of the EventAggregator, a thread handling incoming zeroMQ connection requests and a module (de)serializing the Protocol Buffers messages.

The ProfilingServer realizes an extensible feature set by allowing to provide additional services. Every service is uniquely identified by a ServiceIdentifier and provides a handler for incoming service requests. Upon start of the ProfilingServer the available set of ServiceRequestHandlers is determined and the distinct services are initialized. During initialization every handler registers itself with the EventAggregator to be notified of incoming requests corresponding to its ServiceIdentifier.

Given the scope of the power models used in this thesis and the realization of the AndroidProfiler the ProfilingServer has to provide the following capabilities:

- Experiment time synchronization with the mobile device
- Act as receiving entity for the SendDataBenchmark
- Control the power monitor, particularly start and stop measurements
- Store measurement results and provide them for further processing

The capabilities are provided by distinct ServiceRequestHandlers which are called during different phases of the AndroidProfiler experiment execution respectively. Figure 4.5 provides an abstract overview of the server architecture including the four handlers required for the presented benchmarking and profiling features.

Figure 4.5.: ProfilingServer modules
4.3. Measurement Data Preprocessing

The Profiling Server is designed to be extendable with support for additional functionality. For example, micro-benchmarks on the mobile device representing incoming WiFi transmissions require the server to actively send data to the device. Taking additional parameters into account can also require extended server features, e.g. the utilization of the WiFi channel by other devices can be artificially induced using traffic generators.

4.2.3.1. Power Meter Control

The AndroidProfiler sends a start notification to the PowerProfilingController on entering the DeviceProfilingExperimentInitialization workflow state and a stop notification on entering the DeviceProfilingExperimentFinished state. The notification mechanism relies on registering a hook on the application workflow.

The PowerProfilingController instantiates a new SamplingExperiment and starts the power measurements upon receiving the start notification. The interface IPowerMonitorControl and ISamplingExperiment abstract from the concrete dependence to the API of the Monsoon Power Monitor.

The MonsoonPowerMonitorAdapter implements the IPowerMonitorControl interface and encapsulates the initialization of the Monsoon PowerTool automation. The ISamplingExperiment is created as a handle to a running sampling experiment which encapsulates the reference to the power monitor API and provides capabilities to read out power consumption measurements.

Upon receiving the stop notification the PowerProfilingController stops the current sampling process and publishes a SamplingResultsAvailableEvent containing the ISamplingExperiment instance to the EventAggregator.

4.2.3.2. Power Consumption Measurement Export

The ResultMerger service request handler provides the capabilities of filling the result template as presented in Section 4.2.1 and 4.2.2.4. It takes two distinct events to arrive before the ResultMerger commences operation, a SamplingResultsAvailableEvent and a ServiceRequestEvent from the AndroidProfiler carrying partly filled report.

Based on the timings for the experiment runs specified in the report the power consumption values are read from the external power monitor. The measurements of the interval [begin of earliest phase, end of latest phase) are exported. The resulting report carries the original data, that is particularly system metric targets, plus raw power measurements for each experiment run. The report is published to the EventAggregator as ProfilingResultsAvailableEvent. In the current realization the event is received by the ProfilingResultFileWriter which persists the report as Protocol Buffers message into a result file located on the local disk for further processing.

4.3. Measurement Data Preprocessing

The automated experiment execution results in multiple power consumption traces for the individual experiment runs. The reason to execute experiments in the first place lies in the desire to determining model parameters from the observable behavior. After running
4. Automated Device Power Profiling

various experiments on the mobile device while recording its power consumption the generated measurements have to be analyzed to derive the desired parameters.

The collected raw power measurements require significant amounts of memory dependent on the experiments’ durations and the measurement frequency of the power meter, e.g. the employed Monsoon Power Monitor per-default records 5000 values per second. Furthermore, the collected traces can contain unwanted artifacts resulting from side-effects of the experiment execution. In order to facilitate the model parameter extraction from the raw measurements I realized a preprocessing step.

I decided to integrate the measurements preprocessing as well as the actual model parameter extraction approach with Palladio Bench which in its most recent version at the time of writing comes with the EDP2 framework, a storage framework for measurements, e.g. taken during simulation runs [38].

Primarily, EDP2, the PowerModelSpecification presented by Stier el al. [68] and the ExperimentSpecification rely on the same concepts of specifying system metrics. Therefore, integrating the analysis with Palladio Bench allows to use the provided measurement handling capabilities. Furthermore, leveraging the existing infrastructure allows to use EDP2’s Pipe & Filter architecture [38] and benefit from attached visualization capabilities. Finally, it allows for using Palladio’s model-driven analysis workflow and closely integrate with the existing user interface.

After Experiment Description and Experiment Execution phase the resulting workflow for the third phase of the automated power profiling approach consists of three tasks: importing the raw power measurements from their binary representation, preprocessing the imported values and execute the actual analysis. Due to time constraints I decided to only implement multi-dimensional regression analysis for continuously defined functions as a versatile way of extracting model parameters.

While the analysis is discussed in Section 4.4 this section focuses on preprocessing, particularly aggregating and filtering, of raw power measurements.

4.3.1. Experiment Data Management

The Experiment Data Presentation and Persistence (EDP2) framework is an EMF-based experiment data storage which additionally provides processing and visualization capabilities. Figure 4.6 presents an overview of EDP2’s core experiment data class structure. The root of each consecutive measurement data storage in EDP2 is a Repository. There are different implementations dependent on how EDP2 stores the measurement data (e.g. in-memory, local hard disk). ExperimentRuns represent distinct experiments. ExperimentGroups and ExperimentSettings are two layers to organize the ExperimentRuns.

Actual measurement data is structured as follows. For every kind of measurement data recorded a MeasuringType references the respective MetricDescription. MetricDescriptions identify metrics and specify relevant properties. A MetricDescription is either a base metric (e.g. Power consumption) or composed from others metrics (e.g. Power Consumption over time). Base metrics are further classified in accordance with Stevens’ Scales of Measurement [67] and the type of measurement data (String, Integer or Floating Point). Furthermore, for numerical metrics the MetricDescription additionally specifies a default unit, and whether values are to be stored using a binary or using a textual representation.
4.3. Measurement Data Preprocessing

A set of measurement values that conform to a specified MeasuringType is abstracted by a RawMeasurements instance. MeasurementRanges allow to group RawMeasurements instances into a consecutive time span, specifying the range’s start and end point. The conduct of taken measurements of a certain metric for an ExperimentRun is represented by a Measurement instance, which in turn consists of an arbitrary number of MeasurementRanges.

The actual measurement values are encapsulated as JScience\textsuperscript{5} Measures. Measures represent each value together with its unit. In the context of EDP2 the JScience Measures are encapsulated in MeasuringValues which relate each measurement with its corresponding MetricDescription.

MeasuringValues are available through IDataStreams provided byIDataSources. The EDP2DataTupleDataSource allows to read measurement data from RawMeasurements instances stored in an EDP2 Repository.

All of ExperimentGroup, ExperimentSettings, ExperimentRun and Measurement are implementations of the Propertyable interface, therefore each instance provides a key-value storage AdditionalInformation.

4.3.1.1. Measurements Importer for Binary Result File

The binary, Protocol Buffers-based result format generated by the ProfilingServer uses a concept of nested messages. Therefore, the import mechanism employs a tree-parser-like scheme to interpret the file and store the measurements in the EDP2.

Protocol Buffers deserializes messages into their Java object representation. The hierarchy of deserialized nested message objects is parsed using the following approach. The

\textsuperscript{5}http://jscience.org/
two central elements to the approach are the classes ProtobufMessageParserContext and ProtobufMessageParserSwitch.

The Context is a key-value storage for extracted properties to be handed from one parsing stage to the next. It allows to pass an arbitrary amount of parameters between different stages, without having to specify a distinct parameter list and without the need for every stage to be aware of every parameter.

The property GENERATED_MESSAGE stores the current message object that is parsed. The ProtobufMessageParserSwitch decides upon the type of the message object which particular parser to call. There is a distinct parser for every message defined in the Result Protocol Buffers specification. If a parser has to process a nested message it clones the current Context, sets the GENERATED_MESSAGE property to the nested message, and hands the new context back to the ProtobufMessageParserSwitch. Table 4.1 presents a short summary of the single stages and their respective tasks.

The Approach The RawProtobufGeneratedMessageParser reads the binary result file, deserializes the contained Protocol Buffers message. It then instantiates a new ProtobufMessageParserContext, sets the GENERATED_MESSAGE property to the deserialized message and hands it to the ProtobufMessageParserSwitch. The switch decides upon the type (in this case DeviceProfilingExperiment) the appropriate parser (DeviceProfilingExperimentParser) to hand the message to.
4.3. Measurement Data Preprocessing

The DeviceProfilingExperimentParser ensures that there exists an appropriate ExperimentGroup in the active EDP2 Repository. For each of the subsumed ResourceProfilingExperiments a separate instance of the ParserContext is cloned, each having the GENERATED_MESSAGE property and a property holding the current ExperimentGroup set accordingly. One after another the ParserContexts are handed to the switch for further processing.

The parsing approach continues similarly. Whenever a parser encounters nested messages distinct clones of the currently active ProtobufMessageParserContext are created and the GENERATED_MESSAGE property updated.

4.3.1.2. The Result

After the raw power measurements import is finished, there exists an ExperimentGroup name after the device’s MountedPowerDistributionUnit. For every resource and every experiment conducted with different system metric value targets there is an ExperimentSettings instance.

There is at least one ExperimentRun per ExperimentSettings for which there is one to three PowerConsumptionTuple Measurements, one for the experiments Load phase, and optionally one for Calibration andCooldown phase.

The PowerConsumingResource’s ID as well as the set of system metric target values is stored in the AdditionalInformations of the ExperimentSettings instance. I decided to attach both as additional information to the ExperimentSetting as it then allows to have multiple separate ExperimentRuns for the same device and system metric targets. Running multiple experiments for the same settings can decrease the probability of major disturbances in the power traces.

An alternative approach would be to store the system metric target values as independent Measurement instances for the ExperimentRun. The corresponding MeasuringValues would therefore be accessible through separate IDataSource. This solution would facilitate the handling of the system metric targets’ MeasuringValues as there would be no need for providing EDP2 extensions for the annotation handling.

I decided to realize the first option for three reasons. First, the system metric target values characterize the measurements which are recorded during the corresponding ExperimentRuns. Having the device execute the same setting twice is assumed to provide similar results. Consequently, system metric targets are ExperimentSetting-specific properties and redundant storage of the same data is unnecessary. Furthermore, annotating ExperimentSetting instances with the system metric targets facilitates look-ups for particular system metric constellations, e.g. to determine the idle consumption of the device. Finally, storing the targets separate from the power consumption measurements allows to extend the concept in the future with actual observations of the system metrics, e.g. actual packet rate measured on the server.

4.3.2. Aggregate & Filter Architecture

The large amount of measurements taken during the profiling experiments leads to the requirement of measurements preconditioning. The amount of measurements can easily
be determined using the formula below, with \( n_{\text{experiment}} \) being the number of different experiment runs conducted for a resource, \( t_{\text{experiment}} \) the length of an experiment in seconds and \( r_{\text{measurement}} \) the number of power measurements per second.

\[
n_{\text{experiment}} \times t_{\text{experiment}} \times r_{\text{measurement}} = n_{\text{measurement}}
\]

For example, a CPU profiling experiment with 6 different utilization targets (0.0 to 1.0 in steps of 0.2), an experiment duration of 20s and a measurement frequency of 5000Hz would lead to 600,000 measurement values to be processed by the analysis. Although it is possible to execute regression analyses with large datasets it is time consuming and memory intensive. Therefore, aggregating measures have to be provided to reduce the overall amount of measurements before the actual analysis is conducted.

Figure 4.7.: Power consumption trace: supposedly idle CPU

The experiment duration on the mobile device is controlled by a configuration parameter and enforced using a ScheduledThreadPool. Invoking the timer potentially requires a context switch which leads to increased system load and thereby, generates additional power consumption. The consumption is still regarded as part of the experiment, as it is the timer routine that records the timestamp for the experiment end. Figure 4.7 shows a 10s power trace for a CPU profiling experiment with the CPU being idle (\( u_{\text{CPU}} = 0.0 \)). During the last 500ms there is a spike in power consumption. I attribute the increased consumption to aforementioned effort of scheduler invocations and divergences between device and server time during the experiments. Therefore, additional to aggregating, the set of measurements has to be trimmed as part of the preprocessing procedure.

Regression analyses help in determining relationships between variables. A regression analysis usually has one or multiple input variables (predictor) and an outcome variable (response variable). Based on the form of a function (e.g. linear, or exponential) the analysis tries to find the optimal parameters given a set of (predictors, response) tuples.

After importing the raw measurements into EDP2 they are stored as series of power consumption values together with the timestamp of recording. Through aggregation on the power dimension (e.g. through calculating the average) the timestamps are removed leaving a smaller set of only power consumption values. Each power consumption value is a sample for the regression functions response variable. To properly conduct the regression analysis every aggregated value has to be bundled together with the input metrics of the corresponding experiment run.

A pipeline architecture as shown in Figure 4.8 is chosen, as it provides capabilities to encapsulate the required preconditioning stages. Furthermore, the pipeline allows for new
4.3. Measurement Data Preprocessing

Figure 4.8.: Aggregate and filter architecture: the principle (CPU)

stages to be introduced dependent on the analysis in place, e.g. a stage tracking the highest and lowest measuring value.

The Experiment Data Presentation and Persistence (EDP2) provides already pipeline-based filter concepts [38] for working with measurement values. Measurement data providing entities implement the IDataSource interface and upon request return an IDataStream. The data stream encapsulates a sequence of MeasuringValues and provides the necessary iteration capabilities. The EDP2DataTupleDataSource is a particular implementation that allows to read values from the EDP2.

IDataSink is the counter-part to IDataSource and represents an entity consuming MeasuringValues. Entities that consume as well as provide values implement IFilter, which extends both IDataSource and IDataSink. Chaining multiple filters allows the building of pipeline constructs.

However, the filter capabilities currently provide significant limitations:

1. The ExperimentRun’s system metric targets are stored in its AdditionalInformation annotations. Currently, there is no direct way of accessing the annotated information through the IDSources of the EDP2 pipeline architecture.

2. IDataSource and IDataSink currently only allows connecting a pipeline stage to one predecessor.

Annotated Data Streams Limitation 1 could be circumvented using a custom IDataSource implementation which would provided MeasuringValues extended to already include the system metric target measures. Keeping in mind that there are likely more than 1000 power measurements per second this alternative were to significantly increase the memory consumption.

I resolved the issue by introducing additional annotations to IDataSource, IDataStream, IDataSink and consequently IDataFilter to mitigate problems caused by the first limitation. Similar to the EDP2-EMF-model interface Propertyable the interface Annotatable provides the key-value-pair storage AdditionalInformations, but does not have dependencies to the EMF library.
Handling of Multiple Data Streams  External power meters can only provide measurements for the entire device. As a device’s idle consumption is included in every power measurement, the analysis has to ensure that the idle consumption is only accounted for once. In accordance with Rivoire et al. [55], I include the systems idle behavior in the CPU power consumption abstraction.

Consequently, the WiFi controller’s consumption has to be determined subtracting the device’s idle consumption from the consumption under WiFi load. This step should be part of the preprocessing as the actual power model analysis only requires the additional consumption introduced by WiFi demand.

The distinct measurements for each ExperimentRun are accessed through separate EDP2 DataTupleDataSource instances. Therefore, trimming, aggregating and merging pipeline stages also exist with separate instances per ExperimentRun. In order to generate one input stream for the regression analysis the different streams containing aggregated and extended measures have to be merged into one.

Constant MeasuringValue Annotations

Following the decision to annotate ExperimentSettings with system metric targets, I introduce the type AnnotatedEdp2DataTupleDataSource, an extension to the default EDP2 data source implementation. The extended IDataSource provides access to the AdditionalInformations attached to EDP2 entities.

The extension is realized by encapsulating the IDataStream provided by the original IDataSource using the decorator pattern. Annotating data streams with the system metric targets facilitates access to the information without significantly increasing memory consumption. Figure 4.9 provides an overview of the extended EDP2 class structure.

![Figure 4.9.: EDP2 architecture extension: annotations](image)

The interface ConstantMeasureAnnotatedStream and an extension to the aforementioned data stream decorator, the ConstantMeasurementAnnotationDecorator allow to...
### 4.3. Measurement Data Preprocessing

access the annotated system metric values through native EDP2 interfaces. The ConstantMeasureMergingAdapter acts as filter in the pipeline and appends the system metric values to every tuple of the input stream (cf. MergeAnnotations step in Figure 4.8).

**Measurement Data Aggregation**

The aggregation of measurements is realized by the AggregatingAdapter which delegates all incoming MeasuringValue instances to a MeasuringValueAggregator. The structural concept of measurement aggregation is depicted in Figure 4.10. Dependent on the input metric of the adapter the aggregator is either a BaseMetricMeasuringValueAggregator or a MetricSetMeasuringValueAdapter. Both extract the encapsulated value(s) to aggregate on and delegate the process of aggregating to a JScienceMeasureAggregator. The ArithmeticAverageAggregator for example calculates a cumulative moving average over all input values. The MeasuringValueAggregator encapsulates the JScience Measure instance(s) in a BasicMeasurement or TupleMeasurement respectively. The aggregated measurements are available through the IDataStream provided by the AggregatingAdapter.

![Figure 4.10.: MeasuringValue aggregation](image)

**Measurement Data Trimming**

The DataStreamFilterAdapter provides capabilities to filter MeasuringValue instances from the input IDataStream. The decision, whether a MeasuringValue should be included in the output stream or should be filtered is delegated, similar to the aggregation process.

The decision making process is determined by a set of DataStreamFilterPredicate instances. Every predicate decides based on the MeasuringValue, its consecutive number and the overall number of values whether the value is to be included or not.

```java
public interface DataStreamFilterPredicate {
    boolean evaluate(MeasuringValue measurement, long measurementNo, long measurementCount);
}
```

Listing 4.1: The filter predicate interface

The predicate evaluation is encapsulated in the class FilteredDataStream using the decorator pattern. The iterator provided by the FilteredDataStream on request of the next element in turn pulls the next element from the decorated stream and evaluates the
predicate. In case the predicate evaluates to false the value is dropped and the next value is evaluated.

The DataStreamFilterAdapter can be configured with an arbitrary amount of predicates. As soon as an IDataStream is requested the adapter creates a FilteredDataStream instance per predicate. Due to the decorator pattern the data streams are then easily chained together, the first one decorating the stream from the original IDataSource.

**Annotation Retention**

After the extension with annotation capabilities two types of filters and adapters have to be distinguished: annotation-aware and annotation-unaware ones. In particular, the aggregation adapter and the trim filter have no reason to be aware of annotation as it would unnecessarily limit their compatibility.

Although annotation-unaware filters would properly accept an annotation-awareIDataSource as input, the annotations are not available anymore to filters further down the pipeline. The solution I propose is again based on the decorator pattern. Annotation-unaware filters are wrapped in an IAnnotatedDataFilter instance which hands the input data stream to the encapsulated filter. The output stream of the encapsulated filter is decorated similar to the EDP2AnnotatedDataTupleDataSource.

**Multiple Data Stream Handling**

There are independent sets of measurements for every experiment that has been executed on the device. After trimming, aggregating and extending of the measurements there exists a distinct data stream for each set. Considering that the system metrics values are merged into the measurement tuples there is no reason not to join the streams into a single one.

Therefore, I propose the addition of a IMultiDataStream sink interface describing pipeline components that are able to read input from multiple IDataSource instances. The concrete example of merging multiple streams into one is then provided by the MultipleSources-MergingAdapter which implements IMultiDataStreamSink and IDataSource. It iterates the IDataStream instances provided by all the input data sources and in turn provides a single IDataStream containing the union of all sets of elements.

### 4.4. Model Parameter Extraction

Model parameter extraction denotes the actual analysis that determines the model parameters based on pre-conditioned values. The purpose of pre-coditioning measurement schemes as presented in the preceding section is to facilitate the analysis by removing redundant information, condensing important aspects and reducing the overall amount of data. The model parameter extraction is an autonomous processing step that is executed after the raw power measurements have been imported and the pre-conditioning pipeline has been established. The structure of the pipeline highly depends on the concrete analysis that is conducted. Therefore, it lies within the responsibility of the concrete analysis implementation to build-up the pipeline in a manner as required.
4.4. Model Parameter Extraction

Although as part of this thesis I only implement an automated regression analysis, the parameter extraction approach is designed to be easily extensible with more specialized analyses. This section first presents the general concept of how the parameter extraction integrates with the underlying workflow. Thereafter, I present a concrete implementation with uses a (non)-linear regression leveraging external statistical software to determine the parameter values. Again the interface to the external software is kept general to allow different tools to be used. The presented implementation relies on \textit{R} \textsuperscript{6} as commonly-used, open source environment for statistical computing.

The entire process of importing raw power values, setting up the preprocessing and conducting the actual analysis relies on the \textit{Palladio Workflow Engine}. The engine integrates with the \textit{Launch Configuration} concept of Eclipse and supports users by providing common functionality, in particular for tasks interacting with EMF models.

Workflows are defined as sequence of jobs, which can be nested, and dependent on the actual type of the job determine the manner of execution. In particular, job are distinguished between sequential and parallel jobs. Furthermore, the workflow engine handles interactions with the Eclipse GUI, e.g. progress indication or user-issued cancel requests, and cares about cleaning up after (non)-successful job execution.

The blackboard pattern \cite{9} is used to provide a shared storage among different jobs of a workflow. Therefore, jobs can be designed as blackboard-aware which leads the workflow engine to inject them with a blackboard reference before execution. Every blackboard is defined by a fixed type and organized into partitions of that type. Each partition is identified by a unique name. The \textit{MDSDBlackboard} stores an \textit{EMFResourceSet} per partition, which is used to manage related EMF models and references between them. This blackboard type is of particular significance for the Palladio analyses since it allows multiple jobs to work on the same architecture models.

The model parameter extraction job consists of five stages which are executed in a sequential order:

1. Create blackboard partition holding models (\texttt{PrepareBlackboardForBMGAnalysisJob})
2. Load power models from file (\texttt{LoadBMGModelsIntoBlackboard})
3. Import raw power measurements (\texttt{ImportRawPowerMeasurementsJob})
4. Execute parameter extraction (\texttt{ExecutePowerModelGeneratorJob})
5. Store resulting models (\texttt{StoreBMGModels})

An advantage of using the share blackboard concept of the workflow engine is the clear separation of responsibilities. The fourth step (parameter extraction) only adapts the in-memory representation of the power models and appends the determined parameters. Importing from disk and storing of the result is done independently in distinct jobs.

Both the third and the fourth step are designed with extensibility in mind. The \texttt{ImportRawPowerMeasurementsJob} delegates the actual import procedure to an exchangeable importer implementation. The \texttt{ExecutePowerModelGeneratorJob} parses the loaded Profiling

\footnote{\url{http://www.r-project.org/}}
Experiment Specification and instantiates a PowerModelGenerator for each ResourceProfilingExperiment. PowerModelGenerator is a resource- and power-model-specific implementation of the actual analysis. The PowerModelGenerator approach is explained in detail in the following section.

### 4.4.1. The Power Model Generator Approach

The PowerModelGenerator allows to implement power model parameter extracting analyses for different power models and resources. It integrates the analysis with the underlying workflow and provides functionality to select the necessary input and build up the preprocessing pipeline. The pipeline creation lies within the responsibility of the power model generator as different generator can have different information requirements.

The pipeline build-up is decoupled into a distinct builder to separate the analysis from the initialization logic. The decoupling further increases re-usability since similar analyses can share the same preprocessing pipeline build-up logic.

#### Power Model Generator Input Builder

The concrete PowerModelGeneratorInputBuilder implementation suitable for regression analysis builds up a pipeline similar to the one presented Figure 4.10. Additionally, this AveragePowerConsumptionInputBuilder incorporates differentiation based on the resource type. For resources other than CPUs the calibration consumption is taken into account to prevent idle consumption from being accounted for multiple times. The pipeline is then built-up twice for each ExperimentRun, once for the consumption under load and once for the calibration consumption. An additional pipeline stage merges the two resulting streams into one by calculating pairwise differences. Example:

\[
(Load, u_{CPU} = 1.0, P = 2.0W) \rightarrow (u_{CPU} = 1.0, P = 2.0W)
\]

\[
(Calibration, rate_{WiFi} = 10Hz, P = 0.8W)
\]

\[
(Load, rate_{WiFi} = 10Hz, P = 1.9W) \rightarrow (rate_{WiFi} = 10Hz, P = 1.1W)
\]

#### Power Model Generator Switch

The decision which PowerModelGenerator is suitable for a particular resource is encapsulated in a separate entity. The PowerModelGeneratorSwitch determines based on the passed ResourceProfilingExperimentContext the appropriate generator. The context combines information on the ProcessingResourceType as well as the assigned ResourcePowerModelSpecification.

The analysis realized in this thesis employs a multi-dimensional (non-)linear regression. Therefore, any power model that can be expressed as regression functions (details in the following section) is supported by the provided analysis. In addition to the RegressionPowerModel presented in the following section an analysis mapping the PCA-in-built LinearPowerModel to the appropriate regression function is available.
4.4. Model Parameter Extraction

4.4.2. Automated Regression Analysis

Regression analysis is an approach to fit the parameters of a given function specification using a set of \((input, output)\) tuples. For example, determining the parameters \(a\) and \(b\) of a function presenting the structure \(y = a \ast x + b\) requires at least two tuples \((x, y)\). It is possible, that there is no perfect solution for a larger set of tuples. The purpose of regression analysis is to minimize the overall error in order to determine the best possible fit given a fixed function structure.

Various scientific approaches identify a linear relationship between the utilization of a system’s CPU and its power consumption. Therefore, the resulting power model can be expressed as:

\[
P_{CPU} = P_{CPU_{min}} + u_{CPU} \ast (P_{CPU_{max}} - P_{CPU_{min}})
\]

Given the set of \((P_{CPU}, u_{CPU})\) tuple determined by experiment execution and measurement aggregation it is possible to determine \(P_{CPU_{min}}\) and \(P_{CPU_{max}}\) using regression analysis.

Although the descriptions in the following section focus conduction regression analyses with R the approach is easily adaptable to use other statistical software as well.

4.4.2.1. Regression Model

Every regression model is defined by the response variables, the predictors, the constant model parameters and the inherent relationship between input and output factors. Figure 4.11 depicts an overview of the regression model class structure. While I designed the analysis with a focus on power models the automated regression approach is applicable for arbitrary models with one-dimensional response variables.

![Figure 4.11.: Regression model structure](image)

The `RegressionModel` interface represents the highest level abstraction of a regression model, independent from a concrete representation and specification of how the model is supposed to be evaluated. The `RegressionModel` interface keeps the same abstraction...
level as the PowerModelSpecification. Both specify variable input dimensions (= Measured-Factor), constant model parameters (= FixedFactor) and the output dimension (= metric PowerConsumption).

At the highest abstraction level there is no representation of the relationship between input and output parameters similar to PowerModelSpecification instances. The approach of the Power Consumption Analyzer (PCA) [68] is to identify a PowerModelCalculator based on the PowerModelSpecification instance. The calculator encapsulates a specification of how to calculate the predictions as sequence of explicit Java instructions. Although suitable to generate predictions the model representation of Listing 4.2 is not usable for regression analysis.

```java
/** Maximum and minimum power consumption of the resource. */
Amount<Power> maximumPower; Amount<Power> minimumPower;

/** Utilization metric value of the resource */
Amount<Dimensionless> utilAmount;

return minimumPower.plus(utilAmount.times(maximumPower.minus(minimumPower)));
```

Listing 4.2: Power model representation in LinearPowerModelCalculator (adapted)

The same power model can be represented using the R language for statistical computing as shown in the two examples in Listing 4.3. The first representation is limited to linear regressions (lm = Linear Model) while the second allows to specify arbitrary regression functions (nls = Nonlinear Least Squares, method of conducting the regression). In both cases POWER and UTILIZATION are assumed to be arrays of the same length containing the data from the measurement samples.

```r
# Model specification as linear regression (formula: y = a*x + b)
linReg <- lm(POWER ~ UTILIZATION)

# Model specification, defining function to fit manually
linReg <- nls(POWER ~ minimumPower+UTILIZATION*(maximumPower-minimumPower),
start=c(minimumPower=1.0, maximumPower=1.0))
```

Listing 4.3: Linear power model representation in R

Although both representations, Java and R, are similar, both pursue different objectives. Still, maintaining multiple representations of the same relation leads to increased efforts, is prone to errors and should be avoided (Don’t-Repeat-Yourself principle) [25]. Facilitating the definition of power models I employ a unified representation that can be used for the calculation of prediction values as well as easily translated into a form that is compatible with regression analysis.

The ExpressionOasis Framework provides the desired functionality while offering a intuitive and extensible expression language. ExpressionOasis applies the Composite pattern to represent expressions as nested hierarchies. Figure 4.12 depicts the power models from Listing 4.2 and 4.3 as representation using ExpressionOasis Expression instances.

7https://github.com/mohitkgupta/expressionoasis
4.4. Model Parameter Extraction

Figure 4.12.: Linear power model as ExpressionOasis expression

Figure 4.13 gives a more general excerpt of the available Expression types and their type hierarchy. Using the nomenclature of the Composite pattern, every Expression is either a leaf (NumericExpression for actual values, IdentifierExpression for variables) or a composite (UnaryOperatorExpression, BinaryOperatorExpression).

Every expression provides evaluation capabilities: NumericExpressions evaluate to the encapsulated value, IdentifierExpressions conduct value look-ups using a VariableProvider, UnaryOperatorExpressions evaluate the subsumed Expressions first and then apply their encapsulated function, BinaryOperatorExpressions evaluate both parameters first before applying their function to both of them. The FunctionExpression type uses a FunctionProvider to allow the usage of any Java method that returns values of a compatible type as part of expressions.

Figure 4.13.: Expressions structure in the ExpressionOasis framework

ExpressionOasis further provides capabilities to parse entire Expression trees from a textual representation. Valid expressions are specified in a XML-based grammar. Additionally, the parser needs ahead-of-parsing information about variable names and available functions.

The ExpressionOasis expression language can be integrated with the existing Power Model Specification concept by extending the meta-model. A subclass to the existing Resource Power Model Specification has to be declared that allows to additionally specify the
power model as string attribute. Furthermore, this would allow to validate the expression at the time of specification.

In this thesis I implemented a different approach, first, to circumvent the necessity to change the meta-model, and second, to integrate transparently with the existing calculator infrastructure. The expressions are included in the name of the Power Model Specification using a specific prefix. In this context the linear power model from above would be specified by a Power Model Specification instance as follows:

```
Figure 4.14.: Linear power model specification using ExpressionOasis
```

Two additional interfaces extending RegressionModel are introduced to reflect the presented model capabilities: StringRegressionModel and ExpressionVisitorAcceptingRegressionModel. The first one denotes that there exists a character string representation of the power model (i.e. the part of the name after the delimiter "\_\:\_\:"). The second one that the encapsulated model accepts ExpressionVisitor implementations to walk the tree structure. ExpressionVisitor is an interface that allows to define visitor implementations according to the Visitor pattern and is discussed in detail in Section 4.4.2.5.

4.4.2.2. Regression Model Parameters

The RegressionModel concept presented in Figure 4.11 distinguished two types of model parameters: ValuesDimension and ConstantModelParameter. The first type abstracts from model parameters for which measurement samples will be provided. The second type represents the parameters for which values are determined by the analysis. Instances of both types are uniquely identified by their name. Furthermore, both rely on the JScience framework to provide safety in handling values of different dimensions (e.g. Energy, Duration, Amount of Data) and units.

The regression analysis is designed to determine model parameters based on measurement values for particular system metrics. Consequently, every ValuesDimension instance is specified with respect to a concrete system metric (MetricDescription instance) to ensure a clean binding between the values of a metric and the model parameters. Binding model parameters to particular metrics allows to delegate the assignment of concrete values to the ValuesDimension instances (every instance can determine the appropriate measure from a TupleMeasurement based on its bound metric).

Values Dimensions and Constant Model Parameters in the context of power model regression are initialized based on the Measured Factor and Fixed Factor specification in the Power Model Specification. In order for a Expression-based power model specification to be parsed correctly parameter names of Measured and Fixed Factors have to correspond to the ones used in the expression.
4.4. Model Parameter Extraction

4.4.2.3. Regression Model Data Export

Extracting the values suitable for a ValuesDimension from measurement tuples is encapsulated into a ValuesExportStream instance. Every ValuesDimension instance provides a suitable implementation which is aware of the dimension’s name and metric specification. Therefore, measurement tuples consisting of multiple different metrics can be processed by each ValuesDimension individually. For each ValuesDimension the respective stream extracts the assigned metric.

The concrete implementation for the ValueOutputStream is chosen by the regression-analysis-tool-specific ValueDimension subclass. Figure 4.15 provides an overview of the R specific class structure. The RValueDimension class instantiates one of three concrete ValuesOutputStream types, since the MetricSpecification Framework as well as R distinguishes three data types: Identifier, Real Number and Integer Number or String, Double and Long, respectively. Each ValuesOutputStream instance buffers the extracted raw values until exported en-bloc to an instance of R.

![Figure 4.15: Export of measurement data to R](image)

R stores the exported data as array of the appropriate type. The array can be referenced during further commands by the name of the ValueDimension. The export is encapsulated into the class RDataValuesExporter acting as adapter to the RConnection interface which in turn is a wrapper around the Java/R Interface (JRI) library\(^8\). JRI allows to start a distinct R instance from Java and subsequently interact with it. Commands are issued similar to entering them on the console interface of R. A query string is sent to R, evaluated and potential results are returned as vector representation.

4.4.2.4. Regression Calculator Execution

So far all efforts focused on describing a regression model in an abstract manner so that it can be used both for analysis and for prediction. While the prediction aspect is detailed further in Section 5.2.2, this section focuses on executing the actual regression analysis based on the abstract model specification and the exported sample values. Similar to the export of data, the regression execution is designed with a loosely coupled interface to the regression analysis tool.

\(^8\)http://rforge.net/JRI/

63
Independent of a concrete statistics tool the in the basic structure the results concurs: generated coefficients for the constant model parameters. Furthermore, all analyses in the context of the automation evaluate EDP2 measurements (IMeasuringValue implementations). Therefore, the generic evaluateModel method of the interface RegressionModelCalculator accepts an EDP2 IDataSource and returns a RegressionResult.

Validity checking of the input data source metrics as well as the export of measurement data can done independent of a concrete model representation or a specific regression analysis tool. Required metrics are determined from the model’s predictor variables and the response variable (via the RegressionModel interface). The tool-specific data export behavior is encapsulated by the ValuesOutputStream instances as described in the previous section.

The main regression logic to conduct a regression analysis with R is provided by the AbstractRRegressionModelCalculator class. Using the Template Method pattern the actual the analysis-specific queries are assembled. Therefore, analyses extending this abstract calculator have to follow the same generic approach:

1. Import regression sample data
2. Issue a set of preparation commands (optional)
3. Issue the regression command (required)
4. Issue a command to extract results from regression model (determine coefficients)
5. Parse extracted results from R-specific representation into RRegressionResult

With exception of building the actual regression command the entire preparation and parameter extraction is handled in the abstract class. Furthermore, the construction of the regression command is split into two parts, the actual regression command (everything right of the "<-" assignment operator) and the regression variable assignment. The assignment string is also handled by the abstract calculator as the name of the variable is essential for the parameter export.
4.4. Model Parameter Extraction

The default implementation of the parameter extraction requires the R regression model to be compatible with the two R commands `coef(regVar)` and `names(coef(regVar))`. The first one returns the determined coefficients for the model parameter, the second the parameter names in the same order. Therefore, it is crucial for the model parameters to use the exact name as the respective `ConstantModelParameter` instance.

The regression analysis is implemented using R’s nonlinear least squares `nls` function. I chose to use the nonlinear regression for all types (including linear regressions) as it allows to explicitly specify the model coefficients and therefore facilitates the parameter assignment. The schematics in Figure 4.17 describe the way in which the command is assembled. The transforming of the `ExpressionOasis` tree structure to the R-compatible model string using the Visitor pattern is detailed in the next section.

```
fix
nls(POWER ~ min+UTIL*(max-min), start=c(min=1.0, max=1.0)
fix
ResponseVariable.Name ExportVisitor.ToString Loop
```

Figure 4.17.: Assembling the NLS regression model string

4.4.2.5. Regression Model Export

The tree-based `ExpressionOasis` representation of the regression function has to be converted into a textual representation to be exported into external utilities to conduct the regression analysis. Although the expressing used to build the Expression tree is already a textual representation it can not necessarily be used without changes. `ExpressionOasis` for example specifies a power operation `pow(a,b)` while R reserves a distinct operator `(a^b)`. Transforming one representation into the other potentially requires structural changes to the expression. Therefore, the required transformation exceeds the capabilities of simple string replacement.

The solution I implemented in this thesis leverages the support of `ExpressionOasis` to process all nodes in the expression tree using the Visitor pattern. Every `Expression` accepts an instance of a `ExpressionVisitor`. Upon accepting the expression calls the visitor’s `visit(Expression ex)` method passing the own node as parameter. Unary operators hand the visitor to its operand, binary operators to its two operands after having visited the own node.

Example: Passing a visitor which outputs the symbol of the node at the time of visiting would result in a textual representation in Polish notation (prefix notation). If the visitor is handed to an `AddExpression` (symbol: “+”) with two `IdentifierExpression` (identifiers “a” and “b”, respectively) it would generate the output “+ a b”.

In order to be able to generate infix notations required by R for certain operation the visitor has to keep track of its movements in the tree. The following `ExportVisitor` concept allows to convert an `Expression` into a linear sequence of any type T in prefix, infix and postfix notation and arbitrary combinations of the three. Figure 4.18 provides an overview of the class structure.
The general concept is as follows. The visitor builds up a tree of VisitorStackFrame elements similar to the expression tree by visiting all expression nodes of the latter. Each VisitorStackFrame stores an arbitrary but fixed number of child stack frames. The number of VisitorStackFrame children is determined by the corresponding Expression node’s amount of parameters. Additionally, each stack frame stores the serialized prefix, infix and postfix (part-)representation for the node.

Prefix, infix and postfix representation are provided by an ExportTriple instance which in turn is requested from an ExportTripleProvider. The ExportVisitor maintains a one-to-one mapping between expression node type and corresponding ExportTripleProvider. It is either possible to directly assign the serialized elements based on the expression node type or on the information stored in the node. For example, while AddExpression instances can be directly mapped to the prefix/infix/postfix triple ("", "+", ""), for FunctionExpression instances the name of the function determines the serialized representation.

Therefore, the two concrete ExportTripleProvider types are SimpleTriple and FunctionExpressionExportTripleProvider. The SimpleTriple is used to statically assign a serialized representation with the type of an Expression node, as it also implements the ExportTriple interface itself. The FunctionExpressionExportTripleProvider introduces another level of indirection for FunctionExpression nodes mapping function names to ExportTripleProvider instances.

After the visitor processed all Expression nodes the generated tree of VisitorStackFrame instances has the same structure as the original tree. The serialization of the expression is achieved by collapsing the newly create tree. Each VisitorStackFrame is serialized according to the following description:

<table>
<thead>
<tr>
<th>Before Serialization</th>
<th>After Serialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame (0 Parameter)</td>
<td>Prefix+Postfix</td>
</tr>
<tr>
<td>Frame (1 Parameter)</td>
<td>Prefix+[Parameter1]+Postfix</td>
</tr>
<tr>
<td>Frame (n Parameter)*</td>
<td>Prefix+$\sum_{i=1}^{n-1}([Parameter_i]+Infix)+[Parameter_n]+Postfix</td>
</tr>
</tbody>
</table>

* The serialization support of more than two parameters is currently not completely implemented

Operators: "x+y" Concatenation of x and y, ",[x]" Serialization of node x
The nonlinear regression analysis implementation requires the RegressionModel to be an instance of ExpressionVisitorAcceptingRegressionModel to guarantee the aforementioned serialization approach to work. The calculator creates an appropriate ExportVisitor suitable to the regression analysis tool and passes it to the root node of the Expression. The resulting character string representation is created by concatenating the string representations (toString() output) of all elements in the serialized sequence.

### 4.4.2.6. Regression Result Import

Upon a successful regression analysis the determined model parameters have to be imported from the analysis tool and persisted as FixedFactorValue instances in the BindingModel. The first part of the task, extracting information from the external utility, is split between the RegressionCalculator and the RegressionResult. The second part, creating corresponding model elements, lies within the responsibility of the analysis job, that initiated the regression analysis.

When conducting a regression analysis with R, resulting information is stored in conjunction with the regression variable. There are a couple of R commands that help to extract particular information, e.g. coef(regressionVariable) to get a vector of the fitted model parameters. Therefore, the AbstractRRegressionModelCalculator after executing the actual regression command issues a set of result extraction queries (per-default: coef(RegressionVariable) and names(coef(RegressionVariable))). R sends two vectors in return, one containing the coefficient values, the other the coefficient names. Parsing of the result set is delegated to the R-specific RegressionResult implementation. This RRegressionResult internally builds a map assigning the coefficient name as key to the coefficient value.

Since the regression analysis tool is not necessarily aware of units and dimensions the extracted results stored in the mapping are raw values. From the user’s perspective coefficients are identified by the ConstantModelParameter instance which hides the identifier used by the external analysis utility. Furthermore, the ConstantModelParameter instance retains for every parameter its unit and therefore its dimension. Consequently, the conversion of a coefficient’s raw value to the appropriate JScience measure is delegated to the corresponding ConstantModelParameter instance.

### 4.5. Limitations and Future Work

**Realize State-Based WiFi Power Model** The WiFi power model presented in this thesis is an approach to reflect the predictions of the power models presented by Zhang et al. [77] and Jung et al. [26] using a continuously defined mathematical function. Expressing the model as continuous function allows to reuse regression functionality which had to be implemented for the linear CPU power model. Due to time constraints it was not possible to implement additional analyses, which would have been necessary to support the power models differentiating different power states.

Transitions between states depend on changes in observable metrics (e.g. [77]: low → high, if rate\textsubscript{packet} > 15Hz; high → low, if rate\textsubscript{packet} < 8Hz). Therefore, properly determine model
parameters for the state-based WiFi power models requires to take the following into account.

- Identify different power states or assume existence of the same states
- Determine based on power measurements which power state is represented or assume based on system metric parameter settings (e.g. \( \text{rate}_{\text{packet}} > 15\text{Hz} \) determines high-power state for all devices)

Using a state-based power model allows to account for tail-states (state of increased power consumption for certain amount of time after data transmissions completed).

**Measure System Metrics For Experiment Result**  Currently, the system metric targets are taken as input into the regression analysis as specified in the ExperimentSpecification model. For the data transmission benchmark, in particular, the specified amounts of data and the target packet rate is not necessarily realized by the mobile device for different reasons:

- The product of packet rate and packet size exceeds the maximum transmission rate.
- Large packet sizes are split by the device into multiple packets.

Both cases result in different packet sizes and rates actually being generated. The impact can be significant as the regression analysis takes the specified parameters into account instead of the actually achieved ones. For example the setting for packet rate and size (1Hz, 15KB) is most likely realized as (10Hz, 1.5KB).

The problem could be mitigated by additionally measuring the packet rate and size as they receive at the ProfilingServer. Network protocol analyzers as Wireshark allow to be automated via APIs. The ProfilingServer could leverage a similar template concept as for power measurements to insert the actually measured packet rate and size during the experiment.

**Explicitly Model DVFS**  The realized linear power model for CPUs does not distinguish different CPU frequencies. Taking them into account leads to a series of linear models, one for each distinct frequency. Integrating DVFS-aware linear power models into the presented profiling approach can be achieved as follows:

- Declare the frequency as additional discrete ExperimentDimension
- Realize a component on the mobile phone that sets the CPU frequency manually to the specified metric target values. On the Android platform that can be achieved by writing the frequency into a specific file in the /sys file-system.
- Support discrete parameter dimensions in the regression analysis in order to generate separate minimum and maximum consumption values for each of the frequency stages.

While the benchmarking and profiling approach for DVFS-aware power models is a straight-forward extension, the simulation of CPU behavior must reflect the behavior of the scheduler on the mobile device. As the scheduler behavior can differ significantly (e.g. only lowest frequency, only highest frequency) a PCM model extension would be required.
**Differentiate Between Sending And Receiving** Due to time constraints the profiling approach only conducts micro-benchmarks for sending data from the phone. Many system metric based power models [26, 77] differentiate between sending and receiving.

The presented profiling approach can be extended to include a ReceiveRandomData micro-benchmark. The ProfilingServer would the provide a service that is requested as the TCPPacketSink which instead of offering a server socket would connect to a socket on the AndroidProfiler. The service would the execute a similar behavior to the SendRandomData benchmark.

Differentiated power models for sending and receiving can be realized by specifying a distinct resource for sending and receiving data which is associated with the respective power model.
5. Simulation

The response time of a software system, as well as other characteristics, highly depend on the execution environment. Consequently, the PCM provides means to model the execution environment in an abstract manner (see Section 2.3.1.3).

Optimizing software architectures of component-based cyber-foraging applications with respect to their energy consumption based only on design-time architecture descriptions requires accurate predictions to assess the quality of different design alternatives. Power models as determined using the approach described in the previous chapter can provide valuable information on the consumption behavior. Leveraging such power models requires the activity of hardware resources to be simulated.

The offloading decision for components of cyber-foraging architectures is a trade-off between increased system activity (CPU usage) through local processing and increased network activity through transmitting data to and from the offloaded application parts. Therefore, determining offloading candidates requires forecasts on whether for a particular component the energy which is consumed by offloading-related data transmissions is less than the one consumed through local execution. Consequently, in order to generate predictions on CPU and network power consumption using aforementioned power models their activity has to be simulated.

Although the PCM currently allows to model interconnections between resource containers, there is no notion of resource container specific network controllers. SimuCom offers support for the following aspects.

1. Latency for service invocations across ResourceContainer boundaries [6]

2. Throughput on network-level granularity [23]

3. Reliability of the entire network [8]

The prediction of network-related power consumption essentially requires information on the effort induced by the transmissions in per-node granularity. Section 3.3.1 already covers the need for a meta-model extension with the distinct NetworkDemandProcessingResourceSpecification. Section 5.1 focuses on an approach of leveraging component completions to transparently determine network demand.

The Power Consumption Analyzer (PCA) implementation of Stier et al. [68] has to be extended in order to support power predictions based on network demand forecasts. The realization of the concept presented in Section 3.4 is described in Section 5.2.
5. Simulation

5.1. Device-Specific Network Performance Completions

Automated architecture optimization (Section 2.4) applies changes to the models describing the architecture of the application-to-optimize. Changing components’ allocations to evaluate different deployment schemes is a natural approach to determine offloading candidates.

With changing component allocations there are potential changes to the network demand, e.g. different origins or varying amounts of data. Changes to network demand are of major interest as it is the initial hypothesis that by executing more elaborate calculations on the mobile device the amount of data to transfer could be reduced and thereby, the energy consumption reduced.

A component completions approach employing a model transformation allows to transparently account for network demand by automatically extending the input models with explicit resource demand requests. In particular, the automatic architecture optimization benefits as the required changes to evaluate different deployment schemes are localized to the Allocation model only. Furthermore, component completions allow to factor network related characteristics (e.g. TCP packet header overhead) into the prediction without the need to explicitly specify them in the architecture model.

5.1.1. The Concept

Service invocations between two components are relevant for the network demand accounting if the ResourceContainer of the invoking components is different from the one of the invoked components. In order to account for network demand it is necessary to identify connections between components that cross resource container boundaries. The amount of data which is transferred during service invocations on those connections determines the load on the network devices on the respective resource containers.

Happe et al.[23] present an approach of component completions with a message oriented middleware for the PCM. Direct connections between components (requires/ provides relationship) are replaced with a sequence of adapters routing the service invocations through three different stages of a message oriented middleware (Sender Middleware, Message System, Receiver Middleware). The adapter components are generated based on the interface defined by the original requires/provides relationship. Currently, leveraging features of the existing approach is not a viable option due to incompatibility with the most recent version of Palladio Bench.

The lightweight network demand accounting completions presented in Section 3.3.2 realizes a similar goal. The mechanism is straight-forward and consists of three steps:

1. Identify connectors crossing ResourceContainer boundaries

2. Create proxy components providing and requiring the interfaces of the identified connectors

3. Replace each identified connector with a connector to and from the corresponding generated proxy component.
5.1. Device-Specific Network Performance Completions

The remainder of the chapter uses a more precise terminology from the PCM context. Appendix A holds short summary of all the relevant model elements in presented as assistance to readers who are less acquainted with the distinct terms.

Figure 5.1 shows an exemplary situation with a ComponentB requiring services offered by a ComponentA. A service invocation would lead to the parameters being sent from the right ResourceContainer to the left one and the results the other way round.

![Diagram showing an exemplary situation of two components interconnected pre-transformation](image)

Figure 5.1.: Exemplary situation of two components interconnected pre-transformation

In order to account for network demand at least one NetworkDemandProxy component is generated, dependent on Interface B being equal to Interface A or being a supertype. The proxy component is generated with both an OperationProvidedRole and an OperationRequiredRole for either Interface A or B. Furthermore, two AssemblyContext and AllocationContext instances are created, instantiating the component and allocating it to the respective ResourceContainer. Figure 5.2 shows the situation after the component completion. For the sake of simplicity operation signatures are left out in both figures.

![Diagram showing an exemplary situation of two components interconnected post-transformation](image)

Figure 5.2.: Exemplary situation of two components interconnected post-transformation

5.1.2. Network Demand Proxy

The NetworkDemandProxy components as presented in the previous section are created automatically using model transformation. For every OperationInterface at most one proxy component is generated since the proxy component’s behavior only depends on the encapsulated interface. The generated proxy component describes a RDSEFF for every operation declared in the OperationInterface the proxy component was generated for. The
behavior encapsulated in the RDSEFF is similar for all operations, only depends on the parameters in the OperationSignature and always consists of the following three steps:

1. Calculate amount of data necessary to transmit parameters and issue explicit network demand of that size.
2. Delegate call to Providing Component (either another proxy or initial target)
3. Calculate amount of data necessary to transmit result and issue explicit network demand of that size.

There are three different design alternatives of how the generated proxy component can issue network demand:

**Directly** The network demand is determined based on the BYTESIZE characteristics of all parameters specified by the OperationSignature or the one of the result variable. The demand is issued as ParametricResourceDemand to the WiFi Controller.

**Simple Middleware** The network demand is determined as in the direct case but issued as InfrastructureCall to a middleware component instead. *(chosen alternative)*

**Elaborate Middleware** Employ middleware as presented for *Marshalling* in the approach of Happe et al [23], passing a CollectionDataType for each PrimitiveType to delegate determining the serialized size.

Issuing the demand directly results in a simplified model transformation compared to the other two choices as no additional middleware component has to be allocated and connected to the generated proxies. On the other hand the model transformation has to incorporate the ResourceInterface as that is necessary to issue demand. In order to issue demand to resources providing different interfaces the transformation needs to be changed.

I chose the second alternative over the third as the purpose of the completion is only to account for ResourceContainer-local network demand and not rebuild functionality that is already available. If more elaborate serialization behavior is desired the lightweight transformation can be combined with the more feature-rich middleware completions approach of Happe et al [23].

In order for the proxy component to determine the network resource demand the BYTESIZE characteristic has to be specified for each parameter and return value. The PCM does not require a model to provide all characterizations for all parameters but allows specification of only the necessary ones for the desired analysis. Consequently, requiring BYTESIZE characteristics for every parameter should not be mandatory since not every analysis evaluates network demand.

The execution of the network demand component completions is coupled to the existence of a ResourceContainer-local NetworkDemandProcessingResource to circumvent the problem. Therefore, the model transformation is not executed if the ResourceEnvironment does not contain any NetworkDemandProcessingResource instances. Furthermore, if a ResourceContainer does not specify a NetworkDemandProcessingResource there is no need to instantiate a proxy component (create an AssemblyContext) for that container.
5.1.3. Network Demand Middleware

The lightweight middleware provides a similar Infrastructure Interface to the Resource Interface of the WiFiController presented in Section 3.3.1. Infrastructure Interface and Infrastructure Component behave similar to Business Components with restrictions to prevent cyclic dependencies.

The Network Middleware Interface specifies two Operation Signatures: sendData and receiveData. Both operations expect the serialized size in bytes to be passed as parameter. Decoupling the specification of network demands in the Business Component’s RDSEFF from issuing the demands to the network controller resource introduces a degree of flexibility, as the network demand component completion can be re-used for different adapter models.

Furthermore, the middleware component allows to account for protocol specific overhead, e.g. TCP packets carry a header of 38 bytes if there is no additional header fields attached. Based on a specification of the Maximum Transmission Unit (MTU) the middleware can incorporate the additional overhead before issuing the concrete resource demand.

5.1.4. Example: CMU Sphinx

The principle of the model transformation is illustrated based on a small excerpt from the architecture model of the application that is used to validate the entire deployment optimization approach (see Section 7.2.1). CMU Sphinx is a component-based speech recognition application, the presented excerpt is part of a speech preprocessing pipeline.

Example Repository Figure 5.3 shows a reduced Repository model with a StreamDataSource component requiring a DataProcessor-providing entity and the StreamClassifier being one. The generated proxy component NetworkDemandLoadProxy_DataProcessor simultaneously requires and provides the Operation Interface DataProcessor to connect to the SpeechClassifier and the StreamDataSource component respectively.

Figure 5.3: Repository model of two components taken from CMU Sphinx including generated proxy.
5. Simulation

**Example System**  Figure 5.4 shows a system model in case `StreamDataSource` and `SpeechClassifier` are allocated to different ResourceContainers with only one of the containers being equipped with a network demand processing resource.

The `StreamDataSource` assembly is connected through its OperationRequiredRole to the `NetworkDemandProxy` through the proxy’s OperationProvidedRole. The proxy assembly in turn is connected to the `SpeechClassifier` assembly through its OperationRequiredRole and the classifiers OperationProvidedRole.

A second aspect depicted Figure 5.4 is the proxy component’s dependency to a `NetworkAdapterInterface`-providing entity. As outlined in the previous section the middleware decouples the proxy generation from concrete `NetworkDemandProcessingResource` types. It allows to account for protocol overhead (e.g. due to packet fragmentation) and issue demand to different resources (e.g. CPU demand for WiFi encryption).

The component completion automatically "deploys" an instance of the middleware component (creates `AllocationContext` and `AssemblyContext`, see Figure 5.5 and 5.4 respectively) to ResourceContainer instances to which a generated proxy component has been allocated.

---

**Figure 5.4.:** System model corresponding to the Repository model in Figure 5.3

**Figure 5.5.:** Allocation model corresponding to the System model in Figure 5.4

**Example SEFF**  The behavior of the network demand proxy is determined by the OperationInterface it is generated for. The proxy provides a RDSEFF for every OperationSignature of the encapsulated interface accounting for the network demand and delegating the call.
Figure 5.6 presents the RDSEFF for the void processData(SpeechData data) service of the NetworkDemandLoadProxy_DataProcessor component.

The datasize parameter issued to the NetworkAdapterInterface is determined by a single BYTESIZE characteristic as processData only takes one argument. Thereafter, the call is delegated to the component providing the DataProcessor OperationInterface. As the delegated processData call does not return a value no explicit network demand is issued for the results.

Figure 5.6: RDSEFF corresponding to proxy component in Figure 5.3

5.1.5. QVTo Model Transformation

The component completions presented so far in this section are realized employing the QVTo model transformation language. QVTo allows to define mappings between elements in the source model and elements in the target model using a declarative style, as well as to use imperative instructions to facilitate more complex procedures. Furthermore, QVTo transformation execution is already integrated with the Palladio Workflow. Therefore, it is a straightforward procedure extending the existing simulative analysis approach with the network demand accounting component completions mechanism presented in the last sections.
5. Simulation

The initialization of the transformation first loads the NetworkAdapterInterface from the appropriate model (\_networkMiddlewareInterface). As the demand accounting depends on the exact OperationSignatures the interface is identified using its fixed id. The actual middleware component is loaded from a model passed in as parameter (\_networkAdapterMiddleware) based on the interface loaded in the step before. Thereafter, the Allocation model passed to the transformation as input/output parameter is mapped to its transformed variant.

```java
main() {
    networkMiddlewareInterface :=
        \_networkMiddlewareInterface.objects()[InfrastructureInterface]
        ->any(iface | iface.id = "\_aiHA1qCxEeSlTqYyQtsD4A");

    networkAdapterMiddleware :=
        \_networkAdapterMiddleware.objects()[ImplementationComponentType]
        ->any(comp | comp.providedRoles.InterfaceProvidingEntity[InfrastructureProvidedRole]
            -> select(role | role.providedInterface..InfrastructureProvidedRole.id =
                \_networkMiddlewareInterface.id)->size() > 0);

    \_allocation.objectsOfType(Allocation)-> map processAllocation();
}
```

The model transformation is organized in the following three phases:

```java
mapping inout Allocation::processAllocation() {
    /* 1) Introduce network demand proxy components based on the referenced System model */
    self.system_Allocation.map introduceNetworkTransferProxyComponents();

    /* 2) Determine instantiated middleware assemblies that have to be allocated*/
    self.system_Allocation.assemblyContexts__ComposedStructure +=
            .map determineDeployedMiddlewareAssemblies();

    /* 3) Allocate all created assemblies to the corresponding resource container */
    self.allocationContexts_Allocation +=
        system_Allocation.assemblyContexts__ComposedStructure
            .map allocateAssemblyToResourceContainer();
}
```

Phase 1: Introduce NetworkDemandProxy Components  The introduction of the NetworkDemandProxy components requires modifications to three models: Allocation, System and Repository. The decision where NetworkDemandProxy components have to be introduced is based on Allocation and System model information. The System model specifies instantiations of components and a collection of all connectors between them. The Allocation model stores for the component instantiations the ResourceContainer they are deployed to.

For each ResourceContainer-boundaries crossing AssemblyConnector I refer to the ResourceContainer allocated with the OperationInterface-providing component assembly as the
5.1. Device-Specific Network Performance Completions

providing ResourceContainer. Similar I refer to the one allocated with the interface-requiring component as the requiring ResourceContainer.

The proxy introduction mapping primarily regenerates the set of Connectors specified by the System by keeping connectors not relevant to the completions and replacing ResourceContainer-interconnecting ones. Every interconnecting AssemblyConnector is replaced by one, two or three new ones depending on the number of NetworkDemandProxy components that are created.

The mapping determineRequiringContextOnProvidingContainer maps the unchanged AssemblyConnector to the NetworkDemandProxy instance on the same container as the providing component. In case the ResourceContainer is not suitable for network demand completions it maps to the providing component itself.

A ResourceContainer's suitability is determined by checking whether it provides ProcessingResourceSpecifications for all of the ResourceInterfaces required by the NetworkAdapter-Middleware.

The mapping determineProvidingContextOnRequiringContainer acts similar for the requiring assembly of the unchanged connector. Both mappings internally use createProxyAssembly, mapping two-tuples of AssemblyContext and OperationProvidedRole (meaning: A distinct assembly’s OperationProvidedRole) to the AssemblyContext of the corresponding NetworkDemandProxy. createProxyAssembly initializes an AssemblyContext instance based on the proxy component created by the createNetworkDemandProxyComponent mapping.

```java
1 mapping OperationInterface::createNetworkDemandProxyComponent() : BasicComponent {
2     entityName := "NetworkDemandLoadProxy_" + self.entityName + "";
3     repository__RepositoryComponent := proxyComponentRepository;
4     -- The generated proxy component(result) provides the OperationInterface(self)
5     providedRoles_InterfaceProvidingEntity += object OperationProvidedRole {
6         providedInterface__OperationProvidedRole := self;
7         providingEntity_ProvidedRole := result;
8     };
9     -- The generated proxy component(result) requires the OperationInterface(self)
10    requiredRoles_InterfaceRequiringEntity += object OperationRequiredRole {
11        requiredInterface__OperationRequiredRole := self;
12        requiringEntity_RequiredRole := result;
13    };
14     -- The generated proxy component(result) requires a middleware instance
15     var networkAdapterRequiredRole = object InfrastructureRequiredRole {
16         requiredInterface__InfrastructureRequiredRole := networkMiddlewareInterface;
17         requiringEntity_RequiredRole := result;
18     };
19     requiredRoles_InterfaceRequiringEntity += networkAdapterRequiredRole;
20     -- For every operation of the interface create the proxy RDSEFF
21     requiredRoles_InterfaceRequiringEntity[OperationRequiredRole] -> forEach(reqRole) {
22         serviceEffectSpecifications__BasicComponent +=
23            createDelegatingProxySEFFs(reqRole, networkAdapterRequiredRole);
24     }
25 }
```

Listing 5.1: The mapping of an OperationInterface to its proxy component
5. Simulation

The createNetworkDemandProxyComponent mapping presented in Listing 5.1 maps existing OperationInterface instances to a network-demand accounting proxy component providing and requiring that interface. The helper operation presented in Listing 5.2 creates the RDSEFF for a particular OperationSignature and the OperationRequiredRole the actual call will be delegated to.

```plaintext
helper OperationSignature::createDelegatingProxySEFF(delegateTo: OperationRequiredRole,
  networkAdapterRequiredRole : InfrastructureRequiredRole) : ResourceDemandingSEFF {

  return object ResourceDemandingSEFF {
    describedService__SEFF := self;

    var start := object StartAction {entityName := "Start"};
    var stop := object StopAction {entityName := "Stop"};

    -- Action that issues Network Sending Demand
    var internalActionNetworkParameter := generateActionWithParametricDemand{
      self.createDemandForSumOfInputParameterByteSize(),
      networkAdapterRequiredRole,
      networkInterfaceSendSignature);

    -- Delegating the operation call to the actual component
    var delegatingAction := generateDelegatingExternalAction(self, delegateTo);

    -- Action that issues Network Receiving Demand
    var internalActionResult := generateActionWithParametricDemand{
      self.createDemandForSumOfOutputParameterByteSize(),
      networkAdapterRequiredRole,
      networkInterfaceReceiveSignature};

    -- Adding actions to RDSEFF
    steps_Behaviour += start;
    steps_Behaviour += internalActionNetworkParameter;
    [...]}

    -- Interconnect actions
    start.successor_AbstractAction := internalActionNetworkParameter;
    internalActionNetworkParameter.predecessor_AbstractAction := start;
    internalActionNetworkParameter.successor_AbstractAction := delegatingAction;
    [...]}

  };

Listing 5.2: Creation of RDSEFF for each OperationSignature
```

The generated proxy component requires the availability of a NetworkAdapterMiddleware instance. After the createProxyAssembly-mapping of an assembly context’s Operation[Provided|Required]Role to the AssemblyContext of the corresponding proxy component, the middleware instance is identified or created if necessary.

The corresponding middleware instance is identified using a mapping from the appropriate NetworkDemandProcessingResourceSpecification. ResourceContainers can be attached to multiple LinkingResource instances (networks). The appropriate LinkingResource connecting source and target ResourceContainer is identified. The NetworkDemandProcessingResourceSpecification instance on the ResourceContainer for which the proxy is generated is looked-up based on the identified LinkingResource. The actual middleware assembly
is determined by a mapping of the NetworkDemandProcessingResourceSpecification instance. This mechanism guarantees there to be only one middleware assembly for each NetworkDemandProcessingResourceSpecification.

AssemblyContexts are allocated to ResourceContainer instances in the separate Allocation model. Generating the appropriate AllocationContext instances is separated from generating the AssemblyContext instances to keep the mappings lightweight and reduce model dependencies. In order to allow for ResourceContainers to be specified for newly created AssemblyContexts the model element is temporarily extended with an additional ResourceContainer property allocateToResourceContainer. The property only exists during the model transformation and is evaluated in the third phase.

**Phase 2: Determine Instantiated Middleware Assemblies** A middleware instance is created for a ResourceContainer the first time an AssemblyConnector connecting this container to another is processed. Therefore, the AssemblyContext instances for the middleware component is only created for a distinct ResourceContainer if there are actually NetworkDemandProxies assigned to it.

The set of middleware assemblies for a ResourceContainer is currently limited to one instance per container due to aforementioned PCM constraints. Listing 5.3 demonstrates how the transformation history lookup features provided by QVTo can be leveraged to determine the middleware instances that have been created by a distinct mapping (assembleNetworkAdapterMiddleware).

All identified middleware AssemblyContexts are assigned to their appropriate ResourceContainer using the temporary property introduced in the first phase.

```java
query ResourceContainer::getAssembledMiddlewareComponents() : Set(AssemblyContext) {
    var returnValue : List(AssemblyContext);
    self.activeResourceSpecifications_ResourceContainer
        [NetworkDemandProcessingResourceSpecification]->forEach(resource) {
            returnValue += resource.resolveIn(
                NetworkDemandProcessingResourceSpecification
                ::assembleNetworkAdapterMiddleware,
                AssemblyContext);
        };
    return returnValue->asSet();
}
```

Listing 5.3: Transformation history lookup of middleware components

**Phase 3: Allocate Created Assemblies to ResourceContainer** Modifying the Allocation model is decoupled from System and Repository model changes. The third phase generates appropriate ResourceContainer allocations (AllocationContext) based on all the AssemblyContext instances of the transformed System.

Since only AssemblyContext instances generated during phase 1 and 2 have the temporary property generating the appropriate AllocationContexts is as easy as shown in Listing 5.4. The application of the mapping on the collection of all AssemblyContext instances re-
5. Simulation

Results in a collection of new AllocationContext instances (already allocated components are disregarded).

```
1 mapping AssemblyContext::allocateAssemblyToResourceContainer() : AllocationContext
2 when {self.allocateToResourceContainer != null}
3 {
4   assemblyContext_AllocationContext := self;
5   resourceContainer_AllocationContext := self.allocateToResourceContainer;
6 }
```

Listing 5.4: Allocating newly generated component assemblies

5.2. Energy Consumption Prediction

The Power Consumption Analyzer (PCA) presented by Stier et al [68] generates power consumption predictions based on simulatively determined system metrics. The approach models technical infrastructure as composition of power providing (e.g. Power Supply Unit (PSU)) and power consuming (e.g. Central Processing Unit (CPU)) entities allowing to analyze consumption at different levels in the composition hierarchy. The composition concept targets modeling of complex data centers where there are different hierarchy levels (e.g. CPU ∈ server ∈ rack ∈ data center). For mobile devices usually multi-level modeling is not necessary. Therefore, the hierarchy presents itself as depicted in Figure 5.7. The arrows visualize references from the Infrastructure model instance to the corresponding PCM ResourceEnvironment model instance.

![Infrastructure model of Galaxy Nexus device](image)

Figure 5.7.: Infrastructure model of Galaxy Nexus device

Additionally, the Infrastructure model instance references a PowerBinding model specifying concrete parameters for the assigned PowerModelSpecification instances. The FixedFactor bindings are determined during the automated device profiling in Chapter 4.

**The current PCA architecture**  Figure 5.8 shows an excerpt of the current PCA architecture. Power consumption values can be generated for a particular entity in the Infrastructure model instance using the AnalysisPowerConsumptionAdapter. The adapter requires that a
preceding simulation determined resource activity (State of Active Resource metrics) based on the architecture specification.

![Diagram of PCA architecture](image)

Figure 5.8.: Excerpt from PCA architecture [68]

The purpose of the single entities can be summarized as follows.

**Edp2DataTupleDataSource** The data source provides the PCA with streams of raw measurements collected during simulation (e.g. <State of Active Resource, Timestamp> tuples).

**UtilizationFilter** Aggregates <State of Active Resource, Timestamp> measurements using a SlidingWindow mechanism. Determines for a specified size of the sliding window the relevant measurements based on their timestamps and calculates the fraction of the window during which the ProcessingResource was busy (a.k.a. utilization value). The measurements <Utilization, Timestamp> are generated issuing a calculated utilization value together with timestamp of the sliding window’s end. Thereafter, the window is advanced by a specified window increment and a new utilization value determined until the end window reaches the end of the input stream.

**EvaluationScope/EvaluationScopelerator** Based on the Infrastructure entity to evaluate the scope determines the necessary ProcessingResource measurements (State of Active Resource) and initializes a UtilizationFilter. Furthermore, it encapsulates the functionality to iterate the generated utilization measurements for all resources in parallel. The scope’s internal state represents a concrete point in time during the simulation and is used to provide for every ProcessingResource the utilization metric valid then.

**PowerModelCalculator** Determines for given input metric values the power consumption according to a encapsulated power model.

**PowerModelRegistry** Provides a PowerModelCalculator instance for each PowerConsumingResource. The registry is initialized based on PowerBindings and PowerModelSpecifications referenced by the Infrastructure model instance.
5. Simulation

**ConsumptionContext**  Determines power consumption value for given PowerConsumingResource at EvaluationContext's point in time. Queries EvaluationContext for appropriate metric values and PowerModelRegistry for the corresponding PowerModelCalculator.

**PowerConsumptionSwitch**  Determines power consumption value for given Infrastructure model entity. Aggregates power measures for all subsumed PowerConsumingResource instances.

**AnalysisPowerConsumptionAdapter**  Generates stream of power consumption measurements for given Infrastructure model entity. Advances EvaluationContext, queries PowerConsumptionSwitch and issues a measurement associating the power consumption value with the EvaluationContext's point in time.

The current PCA implementation only supports utilization-based power models, as the EvaluationContext is coupled with the UtilizationFilter. For each PowerConsumingResource a filter instance is automatically created, other metrics cannot be iterated. While one utilization measure is sufficient for the power consumption prediction of CPUs, the presented power model for Wi-Fi controllers requires two: network packet rate and packet size. Therefore, I propose the extension with the following capabilities:

1. Support for arbitrary measures
2. Support for multiple measures per resource
3. Support for differing measuring frequencies

5.2.1. Extension of Power Prediction Capabilities

Extending the PCA with the proposed capabilities requires adapting in particular the EvaluationScope and the encapsulated iterator. The support for multiple metrics per resource which are not necessarily taken at the same point in time requires a more flexible iteration concept. Section 5.2.1.1 focuses on the extended iteration mechanism. Supporting other metrics first requires capabilities to specify for a particular resource the desired ones. Furthermore, removing the assumption that every resource relies on utilization measurements requires decoupling the UtilizationFilter from the EvaluationContext. Providing a more flexible mechanism of supporting derived metrics is discussed in Section 5.2.1.2.

5.2.1.1. Power Consumption Evaluation Mechanism

The EvaluationScope acts as iterator over the utilization values of all resources to evaluate and is used to determine the parameters to feed into the calculators. Currently, the EvaluationContext is limited to handling one utilization metric per resource. Applying the UtilizationFilter to the raw State of Active Resource measurements of each PowerConsumingResource leads to a set of Utilization measurement datastreams.
5.2. Energy Consumption Prediction

**Extending Iteration Behavior** For each resource the utilization measurements are determined at the same point in time with the same frequency, since the same window length and window increment is used by the UtilizationFilter. Iterating multiple streams in parallel becomes easy as advancing the EvaluationScope only requires taking the next measurement of each stream. The behavior is depicted in Figure 5.9a. The utilization measurements for Resource A, B and C are (required to be) equidistant otherwise the set of measures representing the current point in time would diverge.

Taking additional metrics into account, e.g. the processing rate, which change irregularly at a lower frequency requires adaptions to the assumption of equidistant values.

![EvaluationContext Iteration Behavior](image)

**Figure 5.9:** Iteration behavior of EvaluationContext

There are two possibilities of extending the iteration behavior of the EvaluationScope, that is, the concept of how to determine the point in time and the measurements that are evaluated together.

1. Specify a fixed time increment for the EvaluationScope and advance all subsumed datastream iterators accordingly.

2. *(chosen alternative)* Advance the EvaluationScope to the smallest point in time in the future among all subsumed datastreams.

Realizing the first alternative guarantees a resulting stream of equidistant power consumption values. Following integral calculation to determine the energy consumption would not have to evaluate the timespan between two measurements and therefore, could be kept more simple. On the other hand, a fixed increment leads to problems with metric values changing in between two evaluation points in time as it would require to interpolate different values.

The second alternative better reflects the purpose of power consumption simulation. As the power consumption only depends on measured system metrics an evaluation is only necessary when at least one metric changes. Additionally, if the following power consumption analysis requires equidistant metric values it lies within its responsibility to interpolate the respective values. In particular, with respect to discrete metrics (e.g. CPU frequency) where interpolation is no viable mechanism this is the better choice.
5. Simulation

Figure 5.9b visualizes the EvaluationScope advancement for three metrics presenting irregular measuring points in time. Note that the first evaluation point in time is in between t1 and t2 as evaluation is not possible before there is a value for all metrics. Furthermore, it should be borne in mind that given the same input as in Figure 5.9a the new mechanism would result in exactly the same evaluation points in time as before the adaption.

Supporting Multiple Metrics Per Resource  Although the interface of the AbstractResourcePowerModelCalculator already specifies support for calculating a power consumption value from a set of distinct measurements the EvaluationScope only manages one measurement datastream per PowerConsumingResource. The interfaces in Figure 5.10 show the respective capabilities.

```
<<abstract>> ResourcePowerModelCalculator
+ calculate([Measurement]): Amount<Power>

EvaluationScope
```

Figure 5.10.: EvaluationScope and PowerModelCalculator interface excerpt

Supporting multiple metrics per ProcessingResourceSpecification is achieved straightforwardly changing the EvaluationScope’s getMeasurement to getMeasurements returning a collection of measurements compatible to the PowerModelCalculator. Evaluating multiple metrics per ProcessingResourceSpecification requires a selection mechanism as iterating all available metrics for a particular resource can be very inefficient.

PowerModelCalculator instances are created based on the PowerModelSpecification and the PowerBinding corresponding to the resource. Consequently, a calculator is aware of its input metrics. Extending the interface with a getInputMetrics method returning MetricDescription instances for the required metrics allows to parameterize the EvaluationScope dependent on the employed PowerModelCalculator. Therefore the EvaluationScope is extended with configuration capabilities to set the metrics which should be evaluated, that is, returned at getMeasurements, on a per-resource granularity.

5.2.1.2. Extensible Support for Derived Metrics

The adaption from the previous section enabled the EvaluationScope to handle multiple metrics per ProcessingResource and select the desired ones. Still, for resources the available metrics are currently limited to the two directly measured State of Active Resource and Resource Demand metrics, and Utilization values derived using the UtilizationFilter.

Realizing the WiFi power model presented in Section 3.2.4 requires additional metrics to be available. Section 3.4 identifies the necessary metrics as: Nominal Processing Rate of Active Resource, Processing Rate of Active Resource, Processing Unit Size and Processing Unit Overhead. Values for the metrics are available or can be determined based on other simulation measurements.
5.2. Energy Consumption Prediction

**Nominal Processing Rate of Active Resource**  The processing rate w.r.t. the resources base unit of demand specification. The information is available from the `ResourceEnvironment` model instance and constant by default.

**Processing Unit Size**  Specifies amount of data processed at once for resources that process demand in blocks of a particular size, e.g. sending of network data. The measurements are not necessarily static. In the case of network transmissions it depends on a maximum amount (MTU) and the demand issued to the resource. Currently, the block size is specified as a parameter to the network demand middleware (static).

**Processing Unit Overhead**  The processing of each unit can come with a particular overhead (e.g. ethernet frame for WiFi packets). The value is currently also determined by the network demand middleware.

**Processing Rate of Active Resource**  The processing rate w.r.t. processing unit characterizations further depends on the utilization of the resource. If no resource demand is issued then the `ProcessingResource` does not process any processing units. The measurements can be determined in relation to utilization, nominal processing rate and the processing unit characterizations.

There are basically two types of metrics additionally to be supported, characterizations of model entities (e.g. the nominal processing rate) and metrics derived from other metrics (e.g. utilization). Unifying both types and decoupling the metric providing logic from the `EvaluationScope` leads to the `ExtendedMeasureProvider` concept. Every provider encapsulates the logic to extend the `EvaluationScope`’s set of available metrics given the existence of measurements for certain input metrics. Figure 5.11 presents the realized extension of the PCA. Entities drawn inside of boxes refer to existing interfaces and classes of the indicated projects. The triangle with exclamation mark sign marks entities that were adapted in order to support this chapter’s extensions.

Leveraging the Eclipse extension point mechanism allows to determine the available `ExtendedMeasureProviders` at run-time. Consequently, extending the set of available metrics can be achieved non-invasively through providing an additional Eclipse plug-in specifying the new `ExtendedMeasureProvider` implementations.

`ExtendedMeasureProvider` implementations specify a set of required metrics and a set of metrics that can be provided given that all requirements are fulfilled. Building up the set of available metrics becomes an iterative task, as e.g. the result of multiple providers can be chained together. While the set of available metrics is growing every `ExtendedMeasureProvider` instance in turn is queried whether it can provide new metrics given the extended set of potentially available metrics.

Instead of directly providing a `Datastream` of derived measurements the `ExtendedMeasureProvider` returns a `DataSource` which in turn provides the `Datastream`. Using the additional level of indirection allows to leverage the configurable nature of `DataSources`. The `ExtendedMeasureProvider` only instantiates the appropriate entity and sets initially required configurations and all the necessary input data.

In order to support the power models required for assessing cyber-foraging architectures four `ExtendedMeasurementProviders` are implemented: `UtilizationFilterMeasure-
Utilization Measurements  The UtilizationFilterMeasureProvider encapsulates the initialization of an UtilizationFilter instance. The required set of metrics solely consists of the State of Active Resource metric.

Active Resource Processing Rate Measurements  No particular input metric is required by the ProcessingRateMeasureProvider as the Nominal Processing Rate of Active Resource value is determined by the ProcessingResourceSpecification in the ResourceEnvironment model instance.

Still, the provider assumes that at least one metric is available, as the metric measurements’ reference to their MeasuringPoint is leveraged to identify the corresponding ProcessingResourceSpecification. The MeasuringPoint specifies for a series of measurements the relation to the model entity "where" the measurements were taken.

WiFi Packet Characterization Measurements  The size of data packets sent by the mobile device on the wireless network environment. Although the MTU for Wireless LAN is 7981 bytes in experiments I never measured packets larger than 1500 bytes (Ethernet MTU). The measurements were conducted on server-side using the network protocol analyzer Wireshark\(^1\).

\(^1\)https://www.wireshark.org/
5.2. Energy Consumption Prediction

Mainly due to time constraints for simulations I assume a static packet size which is specified by the NetworkAdapterMiddleware. The StaticWiFiComponentPacketCharacterizationDataSource initialized by the PacketCharacterizationMeasureProvider provides measurements containing the characterizations for *processing unit size* as well as the *processing unit overhead*. The values are determined from the middleware model instance.

**Resource Throughput Measurements**  The processing rate w.r.t. the processing unit characterizations is determined using metrics of the aforementioned three providers. Evaluating Equation 3.3 in Section 3.4 allows to calculate the measurement values based on the incoming streams of utilization measurements, nominal processing rate and packet characteristics.

### 5.2.2. Expression-based Power Consumption Calculation

In Section 3.4 I present the concept of a unified regression power model representation for analysis as well as prediction. Section 4.4.2.1 discusses the employed ExpressionOasis framework in more detail. This section focuses on leveraging the presented expression framework in order to provide a versatile PowerModelCalculator.

The two power models presented in Section 3.2.4 can be expressed using the ExpressionOasis framework as follows:

- **CPU**: \( \text{MinPower} + \text{CPUUtil} \times (\text{MaxPower} - \text{MinPower}) \)
- **WiFi**: \( C_1 \times \exp(1/\text{Size}) + C_2 \times \exp(1/\text{Rate}) + C_3 \times \text{pow}(\text{Rate, C4}) \times \text{pow}(\text{Size, C5}) \)

After the text-based representation has been parsed into the tree-based variant (Section 4.4.2.1) values for all the place-holder variables can be set and the entire expression evaluated. The ExpressionOasis framework relies on a VariableProvider concept, that is, an independent entity that resolves a variable’s value at the moments of expression evaluation.

The variables in the expression are declared by the PowerModelSpecification instance as MeasuredFactors and FixedFactors. Variables representing FixedFactors are initialized based on the PowerBinding instance specifying the power model parameters that were determined during the profiling process. The FixedFactor variables are initialized upon instantiation of the PowerModelCalculator.

Variables capturing MeasuredFactors are bound to concrete values the moment the PowerModelCalculator’s calculate method is called with a set of simulated values.

### 5.2.3. Power Consumption Integration

Figure 5.12 depicts a concrete instantiated scenario for analyzing the power consumption in the cyber-foraging context. CPU State of Active Resource measurements are transformed into utilization values and corresponding power consumption values calculated using the LinearPowerModelCalculator provided by the PCA. The WiFi consumption prediction relies on the expression-based calculator presented in previous section. Required input values are provided by the State of Active Resource measurements as well as model entity properties.
The approach presented so far generates power consumption predictions based on PCM architecture descriptions. Assessing architectures with respect to their suitability for the resource-constraint environments requires taking the time perspective into account. The power consumption describes a device’s momentary state, while energy consumption allows better to reason on an architecture’s behavior for entire tasks.

The energy consumption can be determined calculating the finite integral over the power consumption values. The PCA already provides an extension leveraging the Simpson rule in order to integrate power consumption over time. Furthermore, the AnalysisCumulativeEnergyConsumptionAdapter supports the changes to the AnalysisPowerConsumptionAdapter’s output structure (measurements are not necessarily equidistant anymore). Therefore, the adapter can be reused without any modifications.
6. Energy-Conscious Architecture Optimization

The development of a software architecture requires making design decisions. For distributed systems an essential decision is the deployment scheme of the distinct application components. Cyber-foraging applications circumvent resource-restrictions of mobile devices by offloading application logic. Identifying the right offloading candidates is crucial as a badly-chosen application partitioning can decrease performance, increase energy consumption and badly influence other non-functional characteristics or render the entire application unusable.

The Palladio approach (Section 2.3) provides means to assess the quality of a component-based architecture using amongst others simulation-based analyses. Supporting the cyber-foraging offloading decision from the energy consumption perspective requires sufficiently accurate predictions to be determined. The concepts presented in the previous chapters can be leveraged to generate such predictions based on a PCM instance of the application architecture.

Architecture optimization using a simulation-based prediction approach requires conducting a simulation run for every potential architecture candidate. Dependent on the amount of degrees of freedom and the number of choices it is a time-consuming process as for every candidate the architecture model has to be adapted accordingly.

PerOpteryx [30] automatizes the optimization task by building upon the Opt4j framework for meta-heuristic optimization [41]. The optimization is based on an initial PCM instance of the application, a model-based description of the degrees of freedom and a specification of the optimization goals. PerOpteryx then generates new candidates and evaluates them to find Pareto-optimal solutions with respect to the specified optimization goals.

The quality evaluation of a candidate is delegated to an extensible set of analysis components. The concrete analyses that are going to be used are determined by the optimization goal specification. PerOpteryx provides the capabilities to implement new analyses and attach them using the Eclipse extension point mechanism.

6.1. Quality Dimensions Extension

Optimization goals are specified using the Quality of Service Modeling Language (QML). In contrast to the text-based QML representation PerOpteryx provides an EMF-based QML meta-model. The meta-model extends the capabilities of QML with the notion of optimization goals in addition to constraints and PCM-related requirements [46].
6. Energy-Conscious Architecture Optimization

Section 3.5 discussed the goals for an energy-conscious deployment optimization. Translating them into the QML meta-model of PerOpteryx requires the specification of the following model entities.

1. Dimension (QMLContractType model): Specification of energy consumption as quality dimension

2. ContractType (QMLDeclarations/QMLContractType model): Type of contract that supports the energy consumption dimension as QoS dimension

3. Contract (QMLDeclarations/QMLContract model): Instantiation of the ContractType specifying the Aspect Mean as optimization goal

4. Profile (QMLDeclarations/QMLProfile model): Requirements specification for UsageScenario, Contract has to be fulfilled

The EMF-based QoS goal and constraint specification can easily be extended by creating the models as described in step 1 through 4. Additionally, PerOpteryx internally differentiates a fixed amount of QualityAttributes. The currently supported attributes are limited to the following four PerformanceQuality, CostQuality, ReliabilityQuality and SecurityQuality.

QML quality dimensions have to be assigned to either of the categories otherwise they are not taken into consideration. The QML Profile model instance specifying the optimization goals for a concrete scenario is evaluated by a QMLManager. The manager scans the profile for optimization goals or constraints with respect to known QML dimensions of any of the above quality attributes. Thereafter, it identifies suitable analyses to conduct the architecture candidate evaluation.

PerOpteryx’ architecture quality analyses are decoupled from the optimization target specification. While QML models specify aspects for quality dimensions as optimization targets, an extensible set of Analysis components is used to determine actual values for the optimization aspects. Every analysis is specified for a particular QualityAttribute can support the evaluation of multiple Dimensions and different EvaluationAspects per Dimension. Still, the multi-dimensional optimization of PerOpteryx is limited to using one distinct analysis per QualityDimension.

Energy consumption can hardly be assigned to any of the existing QualityAttributes. Therefore, the new attribute EnergyConsumptionQuality is introduced and the QMLManager extended appropriately. Consequently, the energy consumption QML Dimension is assigned to be an evaluable dimension for the new EnergyConsumptionQuality attribute.

6.2. Energy Consumption Analysis

Every analysis evaluating a particular quality attribute of candidates expresses its result regarding a supported dimension as floating point value. The value is used to rank different architecture prototypes to determine the best solutions for this particular quality dimension. The specification for each quality dimension includes whether smaller or larger values are better, e.g. response time and mean time between failure, respectively.
6.2. Energy Consumption Analysis

The goal of this thesis is to automatically optimize architectures with respect to their energy consumption. The concepts presented in the previous chapters provide means to conduct simulations based on PCM architecture descriptions and derive energy consumption prediction from the results. The SimuCom analysis that I use to determine the resource demand necessary to leverage power model predictions is integrated with PerOpteryx, as well. PerOpteryx uses SimuCom as an analysis for the PerformanceQualityAttribute supporting the dimensions response time and throughput.

SimuCom analyses simulate requests to a software system based on a usage scenario model. The model specifies sequences of calls to operations provided by the system that are conducted in either closed or open workload fashion. Open workloads refer to an infinite number of users with a new user arriving regularly (frequency specified through inter-arrival time). Closed workloads symbolize a fixed amount of users, which arrive at the system, execute a scenario and then back-off for a specified amount of time (think-time). In any case the simulation executes the respective scenarios until a defined criteria requests a simulation stop (e.g. simulation reaches specific point in simulated time). The result is a series of measurements over the simulated time-frame.

Leveraging the concepts of the last chapters allows determine predictive power and energy traces based on the simulated system behavior. The energy consumption over the entire simulation time does not convey enough information to properly assess the energy consumption of cyber-foraging candidates as it does not take the execution time of distinct scenarios into account (more detailed in Section 3.5).

The architecture presented in Figure 6.1 realizes the discussed concept of additionally considering response time measurements as they denote begin and end of a user’s arrival at the system.

![Figure 6.1.: Aggregation of energy consumption values](image)

The presented aggregation relies on the Pipes & Filters architecture of EDP2 [38] and reuses extension realized as part of Chapter 4. The instantiated pipeline presented in Figure 6.1 determines the mean energy consumption per usage scenario execution in order to generate one value per architecture candidate that properly expresses the candidate’s quality. It consists of the following five stages.

1. Leverage the extended PCA to generate a stream of power consumption measurements. The measurements are created for the InfrastructureRepository node of the configured model instance, therefore considering all devices that have been specified there.
2. Calculate the integral over the power measurements to determine cumulative energy consumption measurements.

3. Additionally, take the response time measurements for the UsageScenario into account. Filter the stream of cumulative energy consumption measurements. The last consumption measurement before a usage scenario iteration begins and the first one after it finished are contained in the result.

4. Calculate the difference between each two measurements issued by the filter to determine the amount of energy consumed during the corresponding scenario iteration.

5. Aggregate the resulting measurements using the arithmetic average.

For each architecture candidate the analysis produces exactly one energy consumption measurement as result. It is converted from the unit-carrying JScience measure to the non-unit aware optimization algorithm by transforming it into a floating point value in the default unit for energy consumption (kWh). PerOpteryx uses the result to rank the different candidates and identify the ones to archive or to take over into the next generation.

### 6.3. Limitations

**Limited to Single-User Scenario**  Multiple users interacting with a system leads can lead to operations being invoked multiple times in parallel. Therefore, resource demand can be issued from non-related operation executions. Changing resource demand directly influences the power consumption predictions.

The absolute difference between the cumulative energy consumption at the begin of a user’s interaction and at its end does not necessarily include only the consumption related to demand issued by the particular user. Furthermore, the observed energy consumption in a multi-user scenario does not reflect the sum of the consumptions generated by all the users in distinct single-user scenarios.

As long as the focus lies on mobile phones the single-user assumption holds. As soon as multiple interactions with the same device are simulation-relevant, e.g. multiple phones accessing the same cloudlet, other means of determining the candidate’s quality are necessary.

**Elevated Energy-Consumption States not Considered**  The presented analysis takes the energy consumption between the start and the end of a user’s interaction with the system into account. Additional energy consumption due to potentially elevated power states beyond the end of the usage scenario iteration is currently not considered.

This assumption holds as long as none of the power models that are implemented so far rely on power states. Otherwise, a different approach is necessary if the increased consumption should be evaluated.
7. Evaluation

In this chapter I conduct an evaluation of the concepts presented in the preceding chapters. The evaluation is structured into two parts. First, I analyze the quality of the power models in relation to the measurements conducted during the profiling experiments. Secondly, I discuss the quality of the predictions generated by this approach based on a case study. For that, I evaluate simulatively determined optimization results against the measured energy consumption for the real-world deployment scheme. The evaluation part is based on analyzing the component-based cyber-foraging application Speech. Speech is a speech recognition application for the Android platform developed at the Carnegie Mellon Software Engineering Institute (SEI). It is build upon the open-source speech recognition framework CMU Sphinx [70].

CMU Sphinx is a platform-independent Java-based framework for speech recognition. It presents a component-based architecture particularly designed to facilitate exchange of distinct application components [70]. Speech recognition is a field where cloud-offloading techniques are well established (e.g. Apple Siri\(^1\)).

7.1. Automated Profiling Approach

The evaluation of the automated profiling approach is conducted independently for the CPU and the WiFi controller of a Samsung Galaxy Nexus mobile phone. The phone was attached to the Monsoon Power Monitor which in turn was connected to a Dell Latitude E6520 equipped with a Core i5-2520M and 16GB RAM, running under Microsoft Windows 8.1. The same computer also ran the ProfilingServer application. Both were connected to a secured WiFi network which was not exclusively used for testing but presented little to no load at the time of profiling.

7.1.1. Linear CPU Power Model

The CPU power model was created running the profiling experiment while varying the CPU utilization. Due to stability issues and a limited time frame the CalculatePI micro-benchmark is used exclusively. The ExperimentSpecification for the ResourceProfilingExperiment specified utilization metric targets 1.0 down to 0.0 in steps of 0.2.

Figure 7.1 shows the generated load profiles for utilization 1.0 to 0.0. The visualization capabilities are provided by the EDP2 after importing the raw measurements.

The first and the last second of each measurement series is trimmed, the average consumption calculated and a regression analysis conducted. Figure 7.2 shows a plot of the fitted power model determined by the automated linear regression.

\(^{1}\)https://www.apple.com/ios/siri/
The quality of a regression function can be assessed by analyzing the residuals. Residuals are differences between the observed response variable and the prediction value for the same input parameters. The regression analysis tries to minimize the absolute values of the residuals to generate a best possible fit of the function. Consequently, the degree to which the prediction function diverges from the real-world observations can be determined using the residual sum-of-squares (also sum of squared residuals).

For the linear power model the residual sum-of-squares is 0.028 which comes to an average error of 0.068 W.

While the predicted power consumption during idle and full-load scenarios is lower than the observed consumption it is higher during mid-level loads. Applying a slightly curved model fits better to the observed values. Therefore, the analysis is repeated with the following exponential model specification:

\[
P_{CPU} = P_{CPU_{min}} + u_{CPU}^{exponent} \times (P_{CPU_{max}} - P_{CPU_{min}})
\]  

Figure 7.1.: Power traces for utilization targets in CPU profiling experiment

Figure 7.3 shows the refined fit, the residual sum-of-squares is reduced to 0.001 (approx. 95% better) which comes to an average error of 0.015 W (approx. 78% better). The average error lies in the proximity of 1.6% of the difference between minimum and maximum consumption of the Galaxy Nexus base system (including the CPU).

In the remainder of the evaluation the exponential power model is applied to predict CPU power consumption.
7.1. Automated Profiling Approach

The WiFi power model is determined using the SendRandomData micro-benchmark. The two ExperimentDegreesOfFreedom that are adjusted are Processing Unit Size and Processing Rate which control the frequency with which the specified amount of data is sent out. The processing rate is evaluated for each of the values \{0.25Hz, 1Hz, 4Hz, 16Hz, 128Hz, 256Hz, 512Hz\}, the processing unit size for each of \{10Byte, 100Byte, 500Byte, 1000Byte, 1500Byte\}. Consequently, 35 independent experiments are executed in order to analyze the WiFi power consumption. Figure 7.4 shows a selection of resulting power profiles (not yet trimmed).

Experiment run targets together with the difference between average consumption after preprocessing of the load profile and the average consumption after preprocessing of the calibration profile form the regression inputs. Comparing the three calibration profiles Figure 7.10a to 7.10c visualizes the need to determine the calibration load for different setting individually. Due to the higher timer frequency (512Hz vs. 1Hz) the profile presents more spikes from the average around 0.7W - 0.8W to around 1.0W.
7. Evaluation

Although there is no additional load generated by the application explicitly during the calibration profiles small peaks can be seen in Figure 7.10b and 7.10c. Background tasks of different applications and/or the operating system can cause additional demand since the AndroidProfiler does not have exclusive control over the mobile device. The additional demand which is reflected in the power consumption trace. As the issuing of the demand happens infrequently it can have major impact on the result of a concrete ExperimentRun. Additional demand issued during the calibration phase increases the power consumption there and thereby decreases the difference between calibration and load consumption. Similarly, additional demand during the load phase increases the difference.

After preprocessing and aggregating the regression analysis results in the two dimensional function as depicted in Figure 7.5. With 0.071 the residual sum-of-squares is significantly higher compared to the CPU power model. The average error of 0.045 W lies in the proximity of 10% of the maximum amount of WiFi power consumption.

7.2. Speech Preparation

Speech is a cyber-foraging speech recognition application leveraging cloudlet infrastructure, developed at the Carnegie Mellon Software Engineering Institute (SEI). Speech consists of a mobile client for the Android platform and a cloudlet-based server which provides the actual speech recognition capabilities. The server requires clients to connect via TCP and send a request of uncompressed audio data. The data is stored temporarily on the cloudlet and as soon as the data transfer from the client finishes is handed to an instance
of CMU Sphinx to do the actual speech recognition. Upon a complete recognition the Speech server sends the recognized data in plain-text form back to the client.

Speech client as well as the server application are written in the Java programming language and integrate with the SEI cloudlet-platform which allows mobile clients to find suitable cloudlets in close proximity.

In order to make Speech a usable evaluation candidate two things, in particular, have to be done. First, Speech has to be extended to support partial offloading and second, a PCM model instance has to be created to be able to conduct simulation runs. Section 7.2.4 focuses the first preparation aspect, Section 7.2.2 the second.

### 7.2.1. The Sphinx Architecture

CMU Sphinx is a Java-based speech recognition framework, developed at the Language Technology Institute (LTI) at Carnegie Mellon University (CMU). Currently in its fourth version, the architectural design of CMU Sphinx focuses modularity and extensibility [70]. Therefore, the framework is structured into components which are assembled at runtime based on a XML configuration file. The component assembly uses inversion of control principles to instantiate the components and inject dependencies.

Figure 7.6 shows a coarse grained overview of the architectural structure of CMU Sphinx. Its three major parts can be identified as FrontEnd, Decoder and Linguist. In contrast to common perception FrontEnd does not refer to the user interface but comprises a pipeline of preprocessors. Sphinx as speech recognition framework does not come with a user interface but requires to be controlled by third-party application components.
The FrontEnd extracts so-called *features* from the raw audio input, which are small but significant information characterizing very short pieces of audio data. The Linguist initializes a graph-based structure (SearchGraph) that allows to identify candidates of words based on the sequences of pre-processed features. The graph is initialized from three language specific models (The LanguageModel describing a "word-level language structure" [70], the Dictionary describing the composition of words from units-of-speech and the AcousticModel mapping units-of-speech to Hidden Markov Models (HMMs) which in turn can be evaluated against audio features) [70].

The Decoder basically only controls the SearchManager which generates hypotheses for a set of input features using the SearchGraph. Tokens capture the current state in the recognition process, that is the "position" in the SearchGraph and historical information about previously processed features. A Token represents a partial speech hypothesis. The set of active Tokens (potential hypotheses after what was processed so far) are managed in an ActiveList. The set or a subset of it is handed to the Scorer to identify the one which fits the next input feature best.

Figure 7.7 depicts the FrontEnd architecture in more detail. Data is passed from one stage to the next with each component applying its encapsulated routine, starting with raw audio frames in the first stage. At the time of writing the latest default configuration, taken from the official CMU Sphinx GitHub Repository\(^2\) initializes a FrontEnd consisting of the eleven processing stages: DataSource, DataBlocker, SpeechClassifier, SpeechMarker, Preemphasizer, Windower, FFT (= Fast Fourier Transformation), Auto-Cepstrum, LiveCMN, FeatureExtraction, FeatureTransformation.

Without going too much into detail with the single stages, the raw audio input is split into small blocks, noise and silent blocks are filtered out, frequencies characteristic for speech

\(^2\)https://github.com/cmusphinx/sphinx4, 3/19/2015, commit 05e8726
are emphasized, Fourier Transformations are conducted and the speech characteristics are extracted.

The FrontEnd pipeline operates in a pull-fashion, that is, the Scorer actively requests the next feature to process from the last stage of the pipeline. Upon request, each stage pulls the next data frame from its predecessor and processes it. Therefore, the processing activity of each component is triggered by its successor in the pipeline. Consequently, every stage of the pipeline has to know its predecessor and offer a service to request a processed data frame.

Data frames passed between the different stages in the pipeline are abstracted by the interface Data. Consequently, there is a variety of different implementations of which Table B.1 in Appendix B gives a short overview.

7.2.2. Model Creation for Speech

I created an architecture description of Speech and the CMU Sphinx framework based on the technical report on Sphinx [70] and manually inspecting the source code of both Speech and Sphinx. Sphinx uses the term "component" for concrete implementations of interfaces that are instantiated and parameterized by the ConfigurationManager based on the XML-configuration. Not every architecture entity in the Spinx framework can be treated as component in the sense of Palladio and the PCM. SearchGraph and ActiveList are different in-memory data structures whose implementations realize a specific behavior. They are not explicitly deployed but passed as parameters between other components. As a result, they do not conform to Skyperski’s definition (see Section 2.3) as independent deployment is a requirement.

The current implementation of Speech initializes a new instance of the Sphinx Recognizer for every request. Loading the models from file and compiling the SearchGraph results in non-negligible overhead. For small requests the time spent initializing dominates the actual recognition process. I refrain from taking the initialization into account for cyber-foraging scenarios as I assume server providing speech recognition features will keep the language models in memory. Allocating about 1GB of permanent memory seems justifiable for a server to significantly speed up the recognition process (2-3 seconds on the test machine).

To reflect the assumption in the measured values I adapted the Speech server to reuse an existing instantiated instance of the Sphinx framework. Consequently, the validation model does not explicitly model the component Linguist as it is only relevant during the initialization of the framework. The effort spent identifying the next states in the Search-
7. Evaluation

Graph is accounted for as part of the SearchManager since the SearchManager initiates the SearchGraph traversal.

7.2.3. Model Calibration

In order to validate the presented approach a model for CMU Speech has to be created. I used a similar approach as Stier et al [68]. They profiled the execution time for the services of each component to estimate the processing resource demand. While they predicted the energy consumption of servers based on the CPU utilization alone, the energy consumption prediction for network devices additionally requires the specification of the transferred data sizes.

Network demand implicitly influences the response time as transmitting data via the network takes a certain amount of time.

7.2.3.1. Computational Complexity

The resource demand to the CPU resource directly influences the response time of a component’s operation. Assuming an application uses the entire CPU and there are neglectable influences by other applications, the demand for the model can be determined by analyzing the execution time of each modeled operation.

Profiling frameworks like perf4j\(^3\) facilitate profiling of operations and evaluation of the results. In order to determine the resource demand of the separate components of the Sphinx framework I measured the execution time of particular services while sending requests containing speech input of different sizes.

The time spent executing in one component is determined by taking the response time of the provided service and discount all outgoing calls to services of other components. After measuring the execution time per component service over multiple invocations a statistical distribution over the execution time can be generated.

Under the assumption that execution times are stochastically independent among the different components the measured execution time for each component can be described as using a probability distribution functions. Leveraging the PCM’s support for stochastic expressions (StoEx) allows to approximate the measured distribution of execution times using either probability density or probability mass functions.

The execution time of the AutoCepstrum component of the FrontEnd can be described using the following StoEx expression.

\[
\text{DoublePDF}(18.0; 0.0)(23.0; 0.01)...(353.0; 0.01)
\]

The distinct elements of the distribution can be determined by ordering the measured execution times and evaluating the respective percentile. In the presented example the slowest measured execution time was 18ms, 1% of the times were in between 18 and 23ms, and so forth. The slowest execution was 353ms with 1% lying between 62ms and 353ms.

The profiling was conducted on the unmodified version of Speech and the Sphinx framework.

\(^{3}\text{http://perf4j.codehaus.org/}\)
7.2. Speech Preparation

7.2.3.2. Data Transmission Demands

Independent from the response time measurements the amounts of data passed from one component to another were determined. The components of the FrontEnd pipeline all adhere to the same interface. Therefore, I implemented a small proxy component which measures the amount of data by serializing parameters and return values to determine the serialized byte size and then passes the data on. Leveraging the configuration-based component instantiation the DataCountingProxies can be introduced without requiring changes to the Sphinx source code.

The serialization is done either manually writing the raw data (integer and floating point values) for simple objects (e.g. FloatData, DoubleData) or using the kryo framework\(^4\) for nested structures (e.g. Tokens).

The resulting byte size characterizations are determined for each component with respect to the parameters that are passed in. Exception: DataSource and DataBlocker, as for them the size of the data handed to the next stage depends on a component configuration parameter.

7.2.3.3. Execution Environment

Aside from the resource demand description the PCM requires a specification of the execution environment, particularly the demand processing rate for the distinct resources. Therefore, the three ProcessingRate parameters have to be determined for the server’s CPU, the mobile devices CPU and its WiFiAdapter.

The processing rate of the server does not have to be calibrated as the resource demand is already specified in microseconds and the same system is used for the evaluation that was used for measuring the demand.

The calibration of the mobile devices’s CPU resource is conducted by executing the entire Sphinx FrontEnd pipeline independently on both the server and the mobile device. An audio file of 691KByte is loaded into memory to reduce potential impact of storage medium, thereafter the time it takes all stages of the pipeline to process the entire file is measured.

Processing the file took on average 227.17ms on the server, while it took an average 2804.25ms on the Galaxy Nexus. Consequently, the processing rate for the mobile device can be set to the one of the server corrected by the determined factor.

\[
rate_{proc_{mobile}} = \frac{t_{proc_{server}}}{t_{proc_{mobile}}} \cdot rate_{proc_{server}} = \frac{227.17ms}{2804.25ms} \cdot 1,000,000 = 8109.18 \quad (7.2)
\]

The determined difference of a factor of approximately 12 is a very good indicator that comparing the frequencies of the actual CPU’s does not lead to significant results (server: 2.99GHz, Galaxy Nexus: 1.2GHz)

The processing rate of the WiFiAdapter resource is determined transmitting large chunks of randomly generated data (128Mbyte) from the mobile device to the server. During the data transmission the average rate is determined using the NetBalancer traffic control and

\(^{4}\)https://github.com/EsotericSoftware/kryo
7. Evaluation

monitoring tool\(^5\). For the Galaxy Nexus connected to a wireless network using the 802.11n standard the measurements yield an average transmission rate of approx. 2.2MByte/s.

7.2.4. Speech Partial Offloading Adaption

The adaption to Speech presented in this section does not strive to be a production-ready solution but is supposed to serve as a proof-of-concept prototype. Adapting Speech to allow for more mobile-device-local computations requires moving parts of the underlying CMU Sphinx framework from the server to the mobile device.

Due to it being developed entirely in Java the CMU Sphinx framework can almost completely be deployed to the Android platform. The virtual machines executing the Java code differ significantly between desktop (Java HotSpot VM) and the mobile device (Dalvik). In particular, using live audio as speech input is not possible as the concrete input source implementation depends on parts of the Java standard library that are not available on the mobile platform. Nevertheless, the features required by the Speech client can be provided with minimal adaptions.

Running the entire Sphinx framework on the mobile device is potentially possible\(^6\) but proved impracticable. The response time for recognizing an audio file of 8 seconds increased from 8-10 seconds to over 15 minutes. The problem lies in the memory consumption during the building and traversing of the SearchGraph as the debugging output showed a significantly increased garbage collector activity. Profiling on the server shows that building up and maintaining the SearchGraph based on the default models for the English language requires about 1GB of memory.

Consequently, running the Linguist component as well as the SearchManager on the mobile device is impractical, as both require interacting with the SearchGraph. Furthermore, the Tokens passed between SearchManager and Scorer, and the result hypotheses passed from SearchManager to Decoder and from Decoder to Recognizer are deeply nested structures with references to the SearchGraph.Serializing these objects results in data transmissions of about 30MByte per Token and 20MByte per Result instance.

For the validation I adapted the Speech application to be able to run a configurable part of the FrontEnd pipeline locally on the mobile device. The remainder of the pipeline and the recognition procedure is executed on the server.

In contrast to sending the unchanged audio file to the server Speech initializes the local DataSource component, the begin of every FrontEnd pipeline. As the FrontEnd is realized as a pull-pipeline the Speech mobile client retrieves the locally processed data frames from the last stage deployed on the mobile device. The data frame is serialized and sent to the server. The server buffers incoming frames and acts as DataSource to the remainder of the pipeline.

\(^5\)https://seriousbit.com/netbalancer/
\(^6\)successful on Samsung Galaxy Nexus (Android 4.1), unsuccessful on Samsung Galaxy S4 (Android 5.0)
7.3. Evaluation of Energy Consumption Prediction

In order to evaluate the accuracy of the proposed energy consumption prediction for mobile devices I conducted a series of experiments using the speech application and compare the results to actually measured values.

The goal of this thesis is to determine optimal deployment schemes for cyber-foraging applications with respect to the energy consumption. In multiple experiments the results generated by the automatic deployment optimization approach are compared to the real-world consumption. The real-world runs are conducted similar to the execution of closed-workload scenarios in Palladio: the Speech client requests the transcription of an audio file either using the unchanged mechanism or leveraging the local preprocessing capabilities. After receiving the results the client pauses for a certain amount of time (think-time) and starts again.

All audio files are taken from the VoxForge\(^7\) database and contain recordings of a natural person speaking. All recordings present the same encoding (PCM Wave) and the same bitrate (256kbp/s). Consequently, the file size of a recording is only dependent on its length.

7.3.1. Prediction Model Accuracy

A non-negligible factor when regarding the energy consumption is the time perspective. Therefore, evaluating the accuracy of the application model with respect to the predicted response time is the first analysis.

An factor significantly influencing the response time is the transmission rate of the wireless network, as processing on the server requires data first to be transfered. Figure 7.8 compares predicted and measured response time for the different deployment options. The audio file that was transmitted in all the scenarios was the same speech recording of 6s (188KByte). Unlimited WiFi speed here and in the following experiments refers to deactivating the restricting the connection for the measured case and parameterizing the model with the average consumption determined during the calibration (see Section 7.2.3.3). The measurements for limited transmission speeds are achieved using the NetBalancer traffic control and monitoring tool.

The response time simulations show similar patterns at all transmission rates, that is, significant changes can be observed for the same deployment schemes. The impact of the change is influenced very strongly by the transmission rate.

The highest error relative to the response time values is evident in the unlimited scenario. The predicted response time increases for running the components DataSource, DataBlocker and SpeechClassifier locally, while the measured response time decreases slightly. Similarly, in the two transmission rate restricted scenarios the response time increase is more substantial for the simulation than observable in the measurements.

The divergence can be explained by the PCM model specification specifying different semantics for the offloading stage of the pipeline. While the architecture model specifies a blocking behavior throughout the entire pipeline, the prototype implementation employs

\(^7\)http://www.voxforge.org/
7. Evaluation

Figure 7.8.: Simulated and Measured Response Time at Different Transmission Speeds
7.3. Evaluation of Energy Consumption Prediction

A server-side buffer. Data frames that passed the local pipeline on the phone are sent to the server using blocking transfer routines. Therefore, for fast transmission speeds the time it takes to preprocess frames locally and transmit them to the server is shorter than the subsequent processing on the server. The measured response time decrease stems from the server being able to start the recognition process earlier compared to the completely offloaded scheme as only a single data frame has to be transmitted.

The observable difference between simulated and measured response time for the complete offloading scheme appears to be a missing a constant amount between 1.7 and 1.9s. The relative error for the unlimited scenario lies between -49% and +6%. The difference for the complete offloading scheme is (over-)compensated by an over-weighting of network traffic as aforementioned asynchronous effects are not accounted for.

The real-world measurements are determined conducting one experiment of 120s for each deployment scheme and scenario. Therefore, momentary network disturbances leading to packet loss can also introduce divergences.

Another aspect, significantly influencing the response time is the content of the input audio file. The SpeechClassifier component analyses the audio input split into multiple small data frames whether a frame contains relevant information or not (e.g. too silent, too much noise). The subsequent SpeechMarker component removes the frames which were not classified as speech. Therefore, the amount of data passing into the two stages and coming out of them highly depends on the source material.

Furthermore, the information extracted from the audio frames takes significantly different amounts of time to be processed by the SearchManager. The assumption of stochastically independent processing stages obviously also introduces a non-negligible error.

The more relevant network demand becomes, that is, the more influence the transmission of data frames has on the response time the better the predictions become from a relative perspective. The almost constant standard deviation of 1.26 for the unlimited, 1.51 for the 128KByte/s and 1.17 for the 64KByte/s support the speculation of a missing constant amount. Regarding the deviation from a perspective relative to the measured times leads to an mean error \( \frac{1}{n} \sum_{n=0}^{\infty} \left| \frac{r_{n,\text{sim}} - r_{n,\text{measured}}}{r_{n,\text{measured}}} \right| \) of 29.2%, 28.8% and 11.6%, respectively.

### 7.3.2. Network Transmission Power Consumption

I ran multiple experiments using the completely offloaded deployment scheme in order to minimize the impact of the CPU on the mobile device. Figure 7.9 presents measured and simulated power traces for three scenarios: recognition of an 10 MByte, 591 Kbyte and 188 KByte audio files, the first two at unlimited transmission speed, the third one at 64 KByte/s. For the 10 MByte scenario there is no transmission of the result as the recognition process took about 220 seconds.

The power trace for the complete offloading scheme shows the phone running idle except for times of network data transfer. The idle consumption of the system is very well reflected around 0.8W.

All three measured traces Figure 7.9a, 7.9c and 7.9e show a spike in the power consumption at the begin, and less visible for Figure 7.9c and 7.9e at the end of the network transmission. I attribute the consumption peaks to increased system activity due to loading the file into memory, setting up the network transfer (e.g. open socket) and potentially...
freeing buffers after the transmission. The first two scenarios show that the actual network transfer is results in a lower power level compared to the initial peak.

Figure 7.9a to 7.9d visualize ability of the presented approach to properly determine an increased consumption during simulated network data transfer and take the transmission duration into account. The employed WiFi power model approximates the consumption during the longer running data transmission for the restricted scenario well (measured average: 1,059mW, simulated average: 1,042mW, error $\approx 1.6\%$). For higher transmission speeds the generated power predictions underestimate the real-world consumption more significantly (measured average: 1,692mW, simulated average: 1,265mW, error $\approx 25.2\%$).

The implemented WiFi power model predicts power consumption based on the current transmission rate and the size of the transmission units. The predicted values solely depend on the system state at the current point in time. Therefore, very short transmissions deviate significantly from the prediction as the overhead for establishing the transfer exceeds the effort of the actual data transfer. This effect is visible for the result transfer in Figure 7.9c to 7.9f and also for the input file transfer in Figure 7.9e and 7.9f.

Another factor that is evident from the three scenarios is that for larger file sizes the response time is significantly underestimated. The divergence is potentially based on the model being calibrated over a set of 2000 audio files between 37KBytes and 1,122KBytes most of which were smaller than 400KBytes. Assuming the processing time for a single data frame to be independent of the overall amount of frames might lead to processing time underestimations as result hypothesis become longer for longer audio files.

### 7.3.3. Deployment-Scheme Energy Consumption Evaluation

The goal of this thesis is to assess different deployment schemes to determine the most energy-efficient ones. Therefore, it is not necessary to have 100% accurate predictions as long as the accuracy suffices to decide between two alternatives. In this section I evaluate the series of experiments conducted for response time analysis with respect to their energy consumption predictions.

The charts in Figure 7.10 depicting the energy consumption of the different deployment schemes highly resemble their pendants on response time. The duration of speech recognition requests have major influence on the accumulated energy consumption.

Regarding the relative error between predicted and measured consumption the unlimited WiFi transmission speed scenario presents the worst results. The simulated values deviation by 49% on average of the measured counterpart. The deviation for the 128KByte/s and the 64KByte/s limited scenarios is 23% and 17% respectively. The prediction values for response time are similarly too low. Therefore, after analyzing the relative order among the different deployment schemes I will assess the impact of correcting the energy consumption values by the response time deviation.

Table 7.1, 7.2 and 7.3 present relative orderings for both measured and predicted energy consumption for the different evaluated deployment possibilities. The rightmost and leftmost column refer to the respectively last component that is deployed on the mobile device while the remainder of the processing is conducted on the server.

The distinct deployment schemes are normalized based on simulation as well as measurements to the energy consumption of the corresponding complete offloading scheme. Both
7.3. Evaluation of Energy Consumption Prediction

(a) Measured: 10MByte @ Unlimited
(b) Simulated: 10MByte @ Unlimited
(c) Measured: 188KByte @ 64KByte/s
(d) Simulated: 188KByte @ 64KByte/s
(e) Measured: 591KByte @ Unlimited
(f) Simulated: 591KByte @ Unlimited

Figure 7.9.: Power profiles of unchanged speech for different speeds and file sizes
Figure 7.10.: Simulated and measured energy consumption at different transmission speeds

columns, simulated and measured values, are sorted consecutively. As long as the order is
Table 7.1.: Order of optimal deployment schemes (simulated and measured) at unlimited WiFi transmission speed ($\approx$ 2.2MByte/s)

<table>
<thead>
<tr>
<th>Stage (Sim)</th>
<th>R. Diff. (Sim)</th>
<th>R. Order</th>
<th>R. Diff. (M)</th>
<th>Stage (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>100.00%</td>
<td>100.00%</td>
<td>Complete</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>118.39%</td>
<td>Preemphasizer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>125.20%</td>
<td>SpeechClassifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>125.62%</td>
<td>SpeechMarker</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>125.64%</td>
<td>DataSource</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>127.21%</td>
<td>AutoCepstrum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LiveCMN</td>
<td>150.90%</td>
<td>130.46%</td>
<td>LiveCMN</td>
<td></td>
</tr>
<tr>
<td>AutoCepstrum</td>
<td>152.47%</td>
<td>E-1</td>
<td>AutoCepstrum</td>
<td></td>
</tr>
<tr>
<td>DataBlocker</td>
<td>155.19%</td>
<td>135.40%</td>
<td>DataBlocker</td>
<td></td>
</tr>
<tr>
<td>DataSource</td>
<td>155.49%</td>
<td>D-2</td>
<td>DataSource</td>
<td></td>
</tr>
<tr>
<td>SpeechMarker</td>
<td>156.88%</td>
<td>A-4</td>
<td>SpeechMarker</td>
<td></td>
</tr>
<tr>
<td>Preemphasizer</td>
<td>160.91%</td>
<td>B-4</td>
<td>Preemphasizer</td>
<td></td>
</tr>
<tr>
<td>FeatureExtraction</td>
<td>160.92%</td>
<td>149.84%</td>
<td>FeatureExtraction</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>182.98%</td>
<td>168.37%</td>
<td>FFT</td>
<td></td>
</tr>
<tr>
<td>Windower</td>
<td>198.76%</td>
<td>181.35%</td>
<td>Windower</td>
<td></td>
</tr>
</tbody>
</table>

the same for both columns, the same components are placed in the same row. Whenever the relative change of the simulated consumption does not produce the same order as many rows as necessary are skipped to avoid breaking the consecutivity constraint.

In the case of different orderings for simulated and measured values the Relative Order column uses an alphanumerical scheme to visualize the order among "out-of-order" stages. On the first encounter of a component it is assigned a consecutive letter. Whenever a stage appears on the other side the letter is repeated with an attached number. As long as the relative order among those "out-of-order" stages is kept the number is not increased.

Table 7.1 shows the evaluation of different deployment schemes for an unlimited transmission rate. The prediction properly identifies the complete offloading scheme as the best possible deployment option. The relative difference of the out-of-order stages A to D lies within a range of 7% (Measured) and 6% (Simulated) of the corresponding normalized reference point. Taking into account that the absolute value of the simulation reference point lies at -56% of the measured one, the increase between complete offloading and the stages A to D lies between 24.4% and 26.8% of the measured base value (the measured increase: 18.39% to 25.64%).

LiveCMN and AutoCepstrum are predicted to be different by -1.57% of the simulated reference point and 3.25% of the measured reference point. Scaling the predicted error to the absolute value of the measurement reference point allows to express the error between the two stages as $3.25\% - (-1.57)\% \times (1 - 0.56) = 3.94\%$.

FeatureExtraction, FFT and Windower are correctly predicted as most unsuitable deployment schemes.
Table 7.2 presents the same analysis for the 128KByte/s restricted scenario. As there is no need for nested relative ordering I omit attaching numbers to the respective entries.

<table>
<thead>
<tr>
<th>Stage (Sim)</th>
<th>R. Diff. (Sim)</th>
<th>R. Order</th>
<th>R. Diff. (M)</th>
<th>Stage (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoCepstrum</td>
<td>88.39%</td>
<td></td>
<td>95.26%</td>
<td>AutoCepstrum</td>
</tr>
<tr>
<td>LiveCMN</td>
<td>89.39%</td>
<td></td>
<td>97.92%</td>
<td>LiveCMN</td>
</tr>
<tr>
<td>FeatureExtraction</td>
<td>99.57%</td>
<td></td>
<td>100.00%</td>
<td>Complete</td>
</tr>
<tr>
<td>Complete</td>
<td>100.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpeechMarker</td>
<td>233.62%</td>
<td></td>
<td>121.57%</td>
<td>Preemphasizer</td>
</tr>
<tr>
<td>Preemphasizer</td>
<td>237.49%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataSource</td>
<td>264.27%</td>
<td></td>
<td>127.88%</td>
<td>SpeechMarker</td>
</tr>
<tr>
<td>SpeechClassifier</td>
<td>278.68%</td>
<td></td>
<td>147.13%</td>
<td>DataSource</td>
</tr>
<tr>
<td>FFT</td>
<td>336.23%</td>
<td></td>
<td>150.80%</td>
<td>SpeechClassifier</td>
</tr>
<tr>
<td>Windower</td>
<td>500.47%</td>
<td></td>
<td>340.37%</td>
<td>Windower</td>
</tr>
<tr>
<td>DataBlocker</td>
<td>265.39%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataBlocker</td>
<td>278.68%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataBlocker</td>
<td>265.39%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataBlocker</td>
<td>278.68%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Order of optimal deployment schemes (simulated and measured) at 128KByte/s

The deployment scheme analysis correctly identifies the two most suitable offloading schemes and the topmost four with an error of $3.32% - (-0.43\%) \ast (1 - 0.34) \approx 3.6\%$ of the measured complete offloading scenario.

SpeechMarker and Preemphasizer are reversed. Nevertheless, the prediction for both puts them within a similar distance compared to the measurements (3.87% for the simulation, 6.31% for the measurements).

DataSource, SpeechClassifier and DataBlocker form a group similar to the one in the unlimited scenario. Their relative change compared to the complete offloading scheme lies within 5.74% for the measurements and 14.41% for the simulated one.

The both candidates FFT and Windower are again correctly identified as having the worst effect on the resulting energy consumption.

It is notable that there is a significant overestimation of the less likely candidates. The measured increase for the Windower approximately 260 percentage points lower than the simulated one. Correcting for the error in the reference points (simulated value is 36% below measured value of complete offloading scheme) still leaves an overestimation of approximately 67 percentage points (simulated value is 326.9% of the complete offloading scheme measurement).

The results presented in Table 7.3 for the scenario limited to 64KByte/s allow for similar observations. As the overall accuracy of the prediction increases for slower transmission speeds less errors can be observed with respect to the ranking of different deployment schemes.

Unfortunately, the scheme predicted to be the best solution measures only to be the second best. Similar to the unlimited scenario the two offloading stages AutoCepstrum and
### 7.3. Evaluation of Energy Consumption Prediction

<table>
<thead>
<tr>
<th>Stage (Sim)</th>
<th>R. Diff. (Sim)</th>
<th>R. Order</th>
<th>R. Diff. (M)</th>
<th>Stage (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveCMN</td>
<td>69.80%</td>
<td>A</td>
<td>68.99%</td>
<td>AutoCepstrum</td>
</tr>
<tr>
<td>AutoCepstrum</td>
<td>70.06%</td>
<td>A</td>
<td>69.99%</td>
<td>LiveCMN</td>
</tr>
<tr>
<td>FeatureExtraction</td>
<td>79.55%</td>
<td>B</td>
<td>74.49%</td>
<td>FeatureExtraction</td>
</tr>
<tr>
<td>Complete</td>
<td>100.00%</td>
<td></td>
<td>100.00%</td>
<td>Complete</td>
</tr>
<tr>
<td>Preemphasizer</td>
<td>263.84%</td>
<td></td>
<td>154.43%</td>
<td>Preemphasizer</td>
</tr>
<tr>
<td>SpeechMarker</td>
<td>274.37%</td>
<td></td>
<td>172.95%</td>
<td>SpeechMarker</td>
</tr>
<tr>
<td>DataSource</td>
<td>307.69%</td>
<td></td>
<td>195.36%</td>
<td>DataSource</td>
</tr>
<tr>
<td>DataBlocker</td>
<td>315.54%</td>
<td></td>
<td>212.98%</td>
<td>DataBlocker</td>
</tr>
<tr>
<td>SpeechClassifier</td>
<td>328.94%</td>
<td></td>
<td>189.59%</td>
<td>SpeechClassifier</td>
</tr>
<tr>
<td>FFT</td>
<td>399.73%</td>
<td></td>
<td>284.40%</td>
<td>FFT</td>
</tr>
<tr>
<td>Windower</td>
<td>577.44%</td>
<td></td>
<td>436.80%</td>
<td>Windower</td>
</tr>
</tbody>
</table>

Table 7.3: Order of optimal deployment schemes (simulated and measured) at 64KByte/s

LiveCMN are reversed. The error lies within 1% of the measured consumption and 0.26% of the simulated one.

The SpeechClassifier stage is simulated to be significantly worse than DataSource and DataBlocker whilst all measurements have proven otherwise. The consistent deviation could indicate an error in the resource demand specification for the SpeechClassifier component.

**Response Time Correction** The response time has a major influence on the cumulative energy consumption of a system. As discussed in Section 7.3.1 the predicted response time has an error of 42%, 27% and 12% on average for the unlimited, the 128KByte/s and the 64KByte/s scenario respectively.

In order to better evaluate the energy consumption prediction approach I try to separate errors stemming from the power and energy consumption prediction from the ones originating from performance modeling. Figure 7.11 shows for each offloading stage the simulation error compared to the measurement values. For each deployment scheme the three bars represent the error for the corresponding scenario (unlimited, 128KByte/s and 64KByte/s).

I use the following formula to correct the energy consumption values using the deviation of the simulated response time from the measured one for the corresponding scenario and deployment scheme. Although the linear relationship is not necessarily given, I analyze the correction under the assumption that the observed power consumption can be treated as almost constant. The relationship between energy and power \( E(t) = \int_0^t P(t)dt \) for constant \( P(t) \) becomes linear with respect to the time perspective \( E = t \cdot P \). Under this assumption I employ the following correction approach.

\[
E_{\text{simulation}}' = \frac{E_{\text{simulation}}}{1 + error_{rt}} \quad \text{with} \quad error_{rt} = \frac{rt_{\text{simulation}} - rt_{\text{measurement}}}{rt_{\text{measurement}}}
\]  

\( (7.3) \)
7. Evaluation

Figure 7.11: Error between measured and simulated energy consumption before and after linear correction by the response time deviation

Applying the presented formula to all simulated energy consumption values and comparing the results to their measured counterpart results in the errors depicted by the three hatched bars in Figure 7.11. The hatched bars are colored the same way as the solid ones representing the error before the response-time correction and similarly refer to the corresponding scenario.

Response time deviation is responsible for significant parts of the error in almost all deployment schemes. In particular the error for the complete offloading scheme is reduced to 5.4%, -1.4% and -2% (from -56.35%, -34.24% and -28.51%) for the three scenarios respectively. Except for the two deployment schemes FFT and Windower in the transmission rate restricted scenarios the corrected values decrease the prediction error and for most schemes lead to significant improvements.

The visualization of the errors also makes the problem evident that leads to errors in the regarded orderings of the different deployment schemes. While the consumption for schemes DataSource to Emphasizer is an overestimation the one for AutoCepstrum, LiveCMN and FeatureExtraction significantly underestimates the values. Nevertheless, even without the response time correction the presented approach is able to determine the best candidate with little error.

7.3.4. Automatic Optimization Leveraging Evolutionary Algorithms

The evaluations of the selected deployment schemes in the different transmission rate scenarios were conducted using PerOpteryx as automation framework. In this thesis I do not only rely on PerOpteryx for its automation capabilities but for the in-built concept of automated multi-criteria optimization.

Enabling PerOpteryx to analyze at least the different scenarios I evaluated manually in the last section requires a declaration of the architectural degrees of freedom. There are 10 components that can be run either on the server or on the mobile device. For
each component a separate degree of freedom (PerOpteryx: AllocationDegree) is specified. Ten degrees of freedom with two choices each results in a design space of 1024 possible architecture candidates.

With current memory limitations there is no practical solution to execute the speech recognition using language models of similar complexity on the mobile phone. Consequently, degrees of freedom are actually only specified for components that could be run on either device. The recognition of the features extracted from a frame can only be conducted on the server. On the other hand the input data is always generated on the mobile device. Therefore, with the given application and the hardware platforms there is no reason for stages of the FrontEnd pipeline to be deployed on the mobile device when a previous stage is allocated to the server. Generally, the idea of cyber-foraging is to leverage available resources in the environment to circumvent resource restrictions of mobile devices. With this in mind running a component on the server is assumed to be advantageous, in particular, if no additional network transmission is necessary.

Using the basic evolutionary search for new candidates PerOpteryx has no way to determine that of the 1024 possible candidates only the 11 presented in the previous section are viable candidates. New deployment scheme candidates can only be generated randomly or by combining two existing candidates. With 1024 theoretically possible candidates there is little more than a 1% chance that a generated candidate is one of 11 viable deployment alternatives.

The optimal architecture candidate depends on the network transmission rate as discussed in the last sections. Specifying the rate as a degree of freedom is not an option as it is not an architecture parameter but characterizes the execution environment. Therefore, separate simulations are necessary for the distinct transmission speeds.

Conducting an undirected evolutionary design space exploration using PerOpteryx is a time consuming activity. Over multiple hours-long runs PerOpteryx did not come up with viable alternative candidates. Analyzing 11 generations of 10 candidates each took around 90 minutes. In two independent runs (with different network transmission rates), 65 candidates were evaluated for a simulated experiment of 60 seconds. Of the 65 evaluated candidates only two finished the request within the experiment time frame, one of which was the initially specified complete offloading candidate.

The automated optimization of PerOpteryx poses high resource requirements to the executing platform, partly because of the memory intensive energy/power consumption prediction mechanism. Conducting analyses is time-consuming and has to be planned ahead. While early generations are evaluated comparably fast (2-5 minutes per generation) the time increases rapidly (e.g. 10th generation: 12min, 12th generation > 30min). In comparison, evaluating the 3*11 candidates from previous section took 22 minutes (1 generation, 33 candidates, all candidates provided as start population).

### 7.4. Limitations and Future Work

**Response Time Prediction Model Improvement**  
The energy consumption is highly responsive to changes in the response time as observed during the experiments. Therefore, determining accurate performance models is crucial. Although I spent a non-negligible
amount of time calibrating the PCM instance for Speech the response time predictions were off by up to -49%. As the consumption for almost all the measured scenarios was significantly to low there appears to be a systematical error. Due to time-constraints I was not able to follow up on the error. Future work could analyze e.g. on a per-component basis where the current model’s under-estimations stem from.

Furthermore, in future work it is possible to analyze if the manner of how the demand is issued to the simulated resources has impact on the predicted energy consumption. Modeling the FrontEnd pipeline using asynchronous calls in middleware and ensuring execution of only one pipeline stage at the time using passive resource could be a viable option. In particular, using non-linear power models running on 100% resource utilization for half of the time and idle the other half results in different consumption behavior as running on 50% for the entire time.

Extend PerOpteryx with Support for Architecture Patterns  
Leveraging PerOpteryx’ evolutionary optimization without providing further guidance did not yield the expected results as the size of the design space lead to hardly justifiable simulation times. Regarding the Sphinx example, the design space could be reduced drastically if PerOpteryx were aware of the pipeline structure. As there is no reason to move computation back to the phone once a previous stage has already been offloaded.

Lehrig [37] presents a concept of describing architecture templates. Extending the DegreeOfFreedom model of PerOpteryx with architecture templates specific functionality would allow to reduce the 1024 possible allocation schemes to the 11 viable candidates that result from having exactly one cut in the pipeline.
8. Related Work

This section provides an overview of related work. I primarily distinguish between approaches focusing on the analysis and the modeling of the energy consumption of mobile devices, and approaches to analyze different partitioning schemes for cyber-foraging applications.

8.1. Energy Consumption of Mobile Devices

Energy consumption for mobile devices has been studied intensively over the last years. In this section the different approaches to model the energy consumption of mobile devices are presented.

Energy Consumption Prediction Using Palladio  Willnecker et al. [73] present an extension to the PCM’s ResourceEnvironment which allows annotating utilization or throughput based power consumption behavior. In contrast to specifying power models explicitly as in the approach of Stier et al. [68] they rely on implicit informations, e.g. linear power model for CPU. In his bachelor’s thesis Rosenthal [58] extended Palladio with basic support to predict energy consumption for the Galaxy Nexus mobile device based on model-driven simulation. He extended the PCM with capabilities to model batteries and hardware sensors, in particular, camera and GPS. While Rosenthal’s solution requires code extensions to support additional mobile devices, I employ a model-based approach to describe the devices’ energy consumption behaviors. In contrast to his work I focus on predicting the energy consumption for hardware parts which directly influence the offloading decision.

Mobile Energy Consumption Analysis  Rivoire et al. [55] compared different power models which link energy consumption to CPU utilization and other system-metrics (I/O rate for disk, parallelism). They showed that based only on CPU and disk utilization, energy consumption can be predicted with an error smaller than 10% on average. Their work did not focus mobile devices.

Carroll et al. [10] conducted fine-grained measurements on a Neo Freerunner mobile device. Using freely available circuit schematics, they measured the energy consumption of single hardware components while executing different benchmarks. Since necessary circuit schematics are not available for commercial mobile devices, the approach is not viable for a broader application.

Automated Experiment Execution  Rice et al. [54] implemented a system to automatically run energy measurements on Android mobile devices. They also implemented a test
framework which synchronizes the execution of test cases with the energy profiling using external hardware. Their goal is to analyze energy consumption on a fine-grained level to identify the influence of specific phone and network activity.

**Power Estimation and Prediction**  
Zhang et al. [77] proposed PowerTutor which estimates the power consumption for each application and hardware component while running on the mobile device. PowerTutor uses the energy consumption models created with the authors’ PowerBooter approach and the total energy consumption derived from the internal battery management interface. Similarly, Jung et al. [26] proposed AppScope, which similar to PowerTutor, is supposed to estimate energy consumption per application using energy models created beforehand using their DevScope concept. In contrast to the presented model-based prediction, both approaches determine energy consumption only for existing applications running on the mobile device.

Kjærgaard and Blunck [29] proposed PowerProf to generate power models using a genetic algorithm. They execute different training measurements, micro-benchmarks, and used the internal power management APIs to profile the consumption. The measures are fed to a genetic algorithm which determines the parameters for the power model.

**Prediction Model Creation**  
Wilke et al. [72] proposed an approach to compare the energy consumption of different applications in the same application domain. Therefore, abstract test-cases would have to be created for each application domain, which represent typical user interaction. Every application developer would have to implement the test-case. The energy consumption can then be measured and compared using their energy testing approach *JouleUnit* [71]. JouleUnit extends JUnit with the capabilities to profile energy consumption using external power meters and logging of significant events using the *Android Debugging Bridge (ADB)*.

Hao et al. [21] measured the amount of energy consumed for every byte-code instruction of the Dalvik Virtual Machine. Using their per-instruction energy model, they can provide power consumption estimates at source-code level with a variable level of granularity (e.g. per-method, per-line). In contrast to the model-based approach using the PCM, eLens requires to distinguish between different VM instructions.

Pathak et al. [51] traced system calls of the application. They represent device energy consumption through finite state machine and use system-calls as transition triggers. The finite state machine concept allows to model so-called tail power states, which they claim to be very important for wireless network modeling. Pathak et al. [51] showed that their approach can achieve better results compared to system-metric based approaches(e.g. [77, 43, 26]). Nevertheless, tracing system calls to generate the power model requires complex benchmarks to profile the consumption of every system call. Furthermore, the benchmarking and profiling process becomes highly dependent on the operating system. Therefore, every new version of the operating system that introduces system call changes, requires the benchmarking process to be altered.

Like the presented approach, Mittal et al. [43] aimed to enable energy consumption prediction during application development. They generated a power model similar to Zhang et al. [77] using micro-benchmarks. Their approach extends the development
platform emulator with the capabilities to predict energy consumption. In contrast to the presented model-based approach, they require applications to be implemented to run on the emulator.

8.2. Partitioning of Cyber-Foraging Applications

The following section presents different scientific approaches which support energy consumption minimizing application partitioning. As mentioned in Section 2.1.1 static and dynamic concepts are distinguished. Dynamic approaches determine the application partitioning scheme during run-time using profiling techniques. Static approaches make the offloading decision before the application is executed.

Chen et al.[13] propose a concept to dynamically offload parts of mobile java applications to cloud servers. The offloading decision is made based on information on the energy complexity which has to be provided through annotations. The applied energy model only regards CPU consumption.

SmartDiet [59] supports developers to identify parts of existing applications which are potential offloading candidates. Therefore, it executes the application and collects run-time information. SmartDiet requires the application developer to actually make offloading decisions.

MAUI [16] is framework which enables dynamic offloading for .net-based applications. Developers are required to annotate methods which are suitable candidates for offloading. MAUI determines the most energy-efficient partitioning scheme at run-time.

CloudClone [14] runs a virtualized clone of the mobile device on the offloading target. The optimal partitioning scheme is determined before run-time through a combination of static code analysis and profiling of application test-runs. Based on the test results CloudClone realizes fine-grained static offloading schemes for existing, non-altered applications.

SmartDiet and CloudClone are both static approaches which focus on enabling offloading mechanisms for existing applications. All of the presented offloading approaches require the application to be implemented already. The presented approach aims at determining partitioning schemes before the implementation phase.
9. Conclusion

In my master’s thesis I analyzed the applicability of energy consumption predictions using model-based performance analyses to determine among a given set of architecture candidates the ones that result in the lowest energy consumption. The focus lay particularly on cyber-foraging applications as those leverage the advantages of deploying functionality to computing resources available in close proximity to avoid device-local resource restrictions. Besides limited locally achievable performance and features not supported by the local hard- or software, limited battery capacity is a significant factor restricting mobile execution environments.

Energy conscious deployment optimization based on an architectural level assists at making offloading decisions during an early stage in the development of an application. I realized the optimization by extending the automated multi-objective software architecture optimization framework PerOpteryx [30] with the quality dimension energy consumption. Leveraging Stier et al.’s Power Consumption Analyzer (PCA) [68] I enabled PerOpteryx to assess applications’ energy consumptions based on PCM architecture descriptions and EMF-based power models specifying the consumption behavior of the execution environment. In order to facilitate the generation of the power consumption behavior specification I realized an automated profiling concept for mobile devices using model-based experiment specifications.

The developed profiling approach allows to generate power model parameters for mobile devices by monitoring its behavior during micro-benchmark execution. Deployment optimization for cyber-foraging applications requires to analyze the energy consumption of local processing as well as offloading data to be processed remotely. Therefore, I employed two regression-based power models, an one-dimensional, utilization-based CPU model and a two-dimensional, network packet rate based WiFi model. Given a specification of the model and system metric target values the mobile device is controlled, monitored and the model parameters determined accordingly.

The PCM meta-model was extended with a new ProcessingResourceSpecification type to allow for modeling of ResourceContainer-local network demand. The demand is determined based on a component completions approach which estimates the network load based on data size characteristics in the PCM model. The major advantage of using a completions approach is that network demand is accounted for transparently whenever a simulated service call is transmitted via the network.

I evaluated the presented approach with a case study. The component-based speech recognition framework CMU Sphinx [70] provided a suitable foundation particularly because mobile speech recognition applications generally leverage cloud offloading mechanisms. Adapting a mobile client (SEI Speech) to allow for different offloading schemes, simulating the corresponding energy consumption and measuring the real-world one
showed the ability of the presented approach to identify optimal deployment schemes with an error of 1% or less of the best solution.

As energy consumption is determined over time it highly depends on proper performance models. Therefore, having accurate predictions on response time of the distinct components of an architecture increases the quality of energy consumption predictions significantly. Although the raw energy consumption estimates have an average error of approximately 49% of the measured consumption a large fraction of the error is due to inaccurate response time estimates. Correcting the values by the respective deviation of the response time predictions the average error decreases to approximately 11%.

Although the initial goal was to leverage PerOpteryx’ architecture optimization based on evolutionary design space exploration, the approach proved impracticable. Dependent on the number of architectural degrees of freedom and the number of viable candidates a large number of candidates are evaluated unnecessarily. Extending PerOpteryx’ strategy for generating new candidates should be analyzed as an option as well as annotating architecture descriptions to state hidden information explicitly. Explicitly stating the existence of a pipeline pattern for CMU Sphinx can reduce the amount of candidates. PerOpteryx could take the pattern into account when generating new candidates, e.g. as deploying components of a pipeline could be limited to having the pipeline split at only one point.

Guiding PerOpteryx using distinct start population provides the capabilities of evaluating a large number of different deployment schemes comparably fast. The determined energy consumption estimates for all specified candidates can then be analyzed all together without having to explicitly modify the architecture models and manually initiate the simulations.

In conclusion, automated deployment optimization is a complex matter which either takes a very long time to analyze a sufficient amount of randomly generated candidates or requires more elaborate optimization measures. Further case studies are necessary to analyze the viability of structuring component-based cyber-foraging applications to allow for more local processing. Currently, cyber-foraging applications seldomly consist of a large number of components for which local preprocessing is actually advantageous. Particularly as increasing network speeds lower the penalty of transmitting data the incentive to do elaborate preprocessing is lowered. Nevertheless, energy consumption prediction approaches as the one presented in this thesis can provide essential information on how certain design decisions affect energy consumption behavior on an architectural level given that sufficiently accurate performance models can be created.
Bibliography


# A. PCM Meta-Model Elements

<table>
<thead>
<tr>
<th>Model Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Repository Model</strong></td>
<td></td>
</tr>
<tr>
<td>OperationInterface</td>
<td>Description of the signature of services offered by providing Components.</td>
</tr>
<tr>
<td>OperationSignature</td>
<td>Captures for a service its name, the number, name and type of parameters as well as type of return value</td>
</tr>
<tr>
<td>RepositoryComponent (short: Component)</td>
<td>Entity that actually provides service functionality described through OperationInterface; Either BusinessComponent or InfrastructureComponent</td>
</tr>
<tr>
<td>OperationRequiredRole</td>
<td>Expresses for a Component the requirement of having services described in an OperationInterface provided by a different Component</td>
</tr>
<tr>
<td>OperationProvidedRole</td>
<td>The counterpart; Relationship of a Component with the OperationInterfaces for which the component provides functionality for all of the described services</td>
</tr>
<tr>
<td>RDSEFF</td>
<td>Component specific description of resource demand for provided service. Specifies calls to required components and resource demand, dependent on parameter characterizations or results of external calls.</td>
</tr>
</tbody>
</table>

**System Model**

<table>
<thead>
<tr>
<th>Model Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssemblyContext</td>
<td>Instantiation of Component in actual system.</td>
</tr>
<tr>
<td>AssemblyConnector</td>
<td>Connection between the AssemblyContext of a Component requiring functionality of another (OperationRequiredRole) and the AssemblyContext of the Component providing it (OperationProvidedRole).</td>
</tr>
</tbody>
</table>

Table A.1.: Summary of PCM elements relevant to this thesis (I)
## A. PCM Meta-Model Elements

<table>
<thead>
<tr>
<th>Model Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resource Environment Model</strong></td>
<td></td>
</tr>
<tr>
<td>ResourceContainer</td>
<td>Abstraction from real-world executing device (e.g. server, mobile device)</td>
</tr>
<tr>
<td>ProcessingResource</td>
<td>Entity of ResourceContainer that represents a capability to schedule and process demand requests (e.g. CPU, HDD)</td>
</tr>
<tr>
<td>NetworkDemand-ProcessingResource</td>
<td>Special type of ProcessingResource, additionally associated with LinkingResource</td>
</tr>
<tr>
<td>LinkingResource</td>
<td>Model abstraction of a entire network; associated with all ResourceContainers it is interconnecting</td>
</tr>
<tr>
<td><strong>Allocation Model</strong></td>
<td></td>
</tr>
<tr>
<td>AllocationContext</td>
<td>Association of a component’s AssemblyContext with the ResourceContainer it is deployed to.</td>
</tr>
</tbody>
</table>

Table A.2.: Summary of PCM elements relevant to this thesis (II)
## B. CMU Sphinx

<table>
<thead>
<tr>
<th>Data Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoubleData</td>
<td>Encapsulates array of double precision values, carries additional information (first sample number, sample collection time, sample rate)</td>
</tr>
<tr>
<td>FloatData</td>
<td>Same as DoubleData, but encapsulates single precision values</td>
</tr>
<tr>
<td>SpeechClassifiedData</td>
<td>Encapsulates DoubleData object plus additional information on the relevance of the content for speech recognition</td>
</tr>
<tr>
<td>Signal</td>
<td>Abstraction of non-speech data carrying objects that influence the control flow</td>
</tr>
<tr>
<td>DataEndSignal</td>
<td>Emitted if input source reached its end</td>
</tr>
<tr>
<td>DataStartSignal</td>
<td>Emitted before first speech data carrying object</td>
</tr>
<tr>
<td>SpeechEndSignal</td>
<td>Emitted after sequence of speech classified data and before sequence of non-speech classified data</td>
</tr>
<tr>
<td>SpeechStartSignal</td>
<td>Emitted before sequence of speech classified data</td>
</tr>
</tbody>
</table>

Table B.1.: Overview of different CMU Sphinx data objects that pass the FrontEnd pipeline