Mango: A Model-driven Approach to Engineering Green Mobile Cloud Applications

Samuel Jaachimma Chinenyeze

A thesis submitted in partial fulfilment of the requirements of
Edinburgh Napier University, for the award of
Doctor of Philosophy

April 2017

Abstract

With the resource constrained nature of mobile devices and the resource abundant offerings of the cloud, several promising optimisation techniques have been proposed by the green computing research community. Prominent techniques and unique methods have been developed to offload resource/computation intensive tasks from mobile devices to the cloud. Most of the existing offloading techniques can only be applied to legacy mobile applications as they are motivated by existing systems. Consequently, they are realised with custom runtimes which incur overhead on the application. Moreover, existing approaches which can be applied to the software development phase, are difficult to implement (based on manual process) and also fall short of overall (mobile to cloud) efficiency in software quality attributes or awareness of full-tier (mobile to cloud) implications.

To address the above issues, the thesis proposes a model-driven architecture for integration of software quality with green optimisation in Mobile Cloud Applications (MCAs), abbreviated as Mango architecture. The core aim of the architecture is to present an approach which easily integrates software *quality attributes* (SQAs) with the green optimisation objective of Mobile Cloud Computing (MCC). Also, as MCA is an application domain which spans through the mobile and cloud tiers; the Mango architecture, therefore, takes into account the specification of SQAs across the mobile and cloud tiers, for overall efficiency. Furthermore, as a model-driven architecture, models can be built for computation intensive tasks and their SQAs, which in turn drives the development – for development efficiency. Thus, a modelling framework (called Mosaic) and a full-tier test framework (called Beftigre) were proposed to automate the architecture derivation and demonstrate the efficiency of Mango approach. By use of real world scenarios/applications, Mango has been demonstrated to enhance the MCA development process while achieving overall efficiency in terms of SQAs (including mobile performance and energy usage compared to existing counterparts).

Acknowledgments

I would like to thank my Director of Studies, Dr Xiaodong Liu; second supervisor, Prof Ahmed Al-Dubai; and panel chair, Prof Emma Hart; for their invaluable supervisory contributions throughout the period of my PhD study. I would also like to thank Dr Sally Smith for every contributed moral support, and all staff in the School of Computing and Library Department at Edinburgh Napier University (ENU), for their efforts to ensure sufficient supply of materials used for this research. Many thanks go especially to the members of the Centre for Algorithms, Visualisation and Evolving Systems (CAVES) group in ENU, for providing me with feedback and suggestions pertinent to my PhD study.

I am most grateful to my loving parents: Dr Anthony and Mrs Catherine Madu, for their dedicated investment and faith in me. Much gratitude goes to my siblings and family: Peace, Grace-Joy, Kenneth and Hadassah, for their endless encouragement, prayers and support during the period of my PhD study. My appreciation goes to Engr Raphael Akala and family, Bro Samba Bindia and Bro Henry Adomako for every love and support – I really am blessed. Many thanks to friends and families at Charismatic Renewal Ministries, Edinburgh Elim, ANCF Edinburgh and Salvation (Chinese) Church London for their encouragement and spiritual support. I am proud of them and appreciate what they contribute to my life. Finally, to those who always believe in and expect the best from me, this achievement is my way to say thanks and ubuntu[1], keep believing.

Samuel Chinenyeze PhD

Nisi sapientia frustra.

[1] Ubuntu is an African way to say 'I am what I am because of who we all are'.

Dedicated to my wonderful friend and mentor, *El-Rohi*.

Publications from the PhD Work

Conference Papers

[1] Chinenyeze, S., Liu, X., Al-Dubai, A., (2014), “An Aspect Oriented Model for Software Energy Efficiency in Decentralised Servers”. In: *2nd International Conference on ICT for Sustainability (ICT4S 2014)*. Stockholm, Sweden: Atlantis Press. doi:10.2991/ict4s-14.2014.14.

[2] Jamshidi, P., Pahl, C., Chinenyeze, S., Liu, X., (2014). Cloud Migration Patterns: A Multi-Cloud Architectural Perspective. In: *10th International Workshop on Engineering Service-Oriented Applications (10th ed.)*. Paris, France. doi:10.1007/978-3-319-22885-3_2.

[3] Chinenyeze, S., Liu, X., Al-Dubai, A., (2016), “DEEPC: Dynamic Energy Profiling of Components”. In: *the 10th IEEE International Workshop on Quality-oriented Reuse of Software (QUORS’16) in conjunction with the 40th IEEE International Conference on Computers, Software & Applications (COMPSAC 2016)*. Georgia, USA. doi:10.1109/COMPSAC.2016.90.

Journal Articles

[4] Chinenyeze, S., Liu, X., Al-Dubai, A., (2017), “BEFTIGRE: Behaviour-driven Full-tier Green Evaluation of Mobile Cloud Applications”. *Journal of Software Evolution and Processes, Special Issue on Software Engineering for Sustainability (JSEP SE4S)*. doi:10.1002/smr.1848.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1. Introduction

1.1 Introduction

The purpose of this chapter is to introduce the work carried out in this thesis. The chapter first introduces the research areas of interest and associated gaps in the form of a problem statement; and further presents the aims and objectives targeted towards fulfilling the gaps. Consequently, the body of knowledge contributed by this thesis to the research area is presented. Also presented is the research method adopted by the thesis which highlights the criteria for success. The chapter is concluded with a thesis structure.

1.2 Problem Statement

Mobile devices are increasingly gaining popularity due to their convenience of usage and portability for end-users. In current times it is possible to perform all or most daily computing requirements on mobile devices, given that there are a vast amount of mobile applications that ensure this possibility. Portability in mobile devices is realised by lightweight components such as microchips, batteries etc. which makes mobile devices resource constrained in nature compared to counterpart computing environments (such as laptops, desktops and servers). Although there is a continuous advancement in the mobile hardware industry, the applications which run on these environments offer a higher range of rich features most of which consume mobile resource (battery inclusive) and consequently energy inefficient. Furthermore, with mobile computing dependent on battery life which is highly constrained [1], many research works have investigated techniques for prolonging the battery life and reducing the extensive use of other mobile resources. Since software bloat are the key contributors to the energy consumption in computing devices [2], [3], a vast amount of research investigate ways to improve the applications running on the mobile device to use fewer resources which amount to energy/battery savings.

Energy efficiency and performance are important qualities for mobile applications because of the resource constrained nature of mobile devices. In other words, mobile devices do not possess very high computing power as other computing platforms (such as laptops and desktops), and moreover, they also possess a more constrained power supply, thereby making energy-efficiency and performance crucial mobile software qualities.

A major contributor to the energy efficiency of mobile applications is the use of the cloud (a resource-rich environment) to complement the mobile end (a resource-constrained environment), thus the goal of mobile cloud applications (MCA).

Due to the dynamic characteristics of MCAs, that is constant changing resource state and workload in MCA environments, mobile energy optimisation is now dependent on varying runtime environmental factors (such as network bandwidth and latency, data size and server conditions to mention but a few). Various approaches (details presented in the literature review) have been proposed in the literature for mobile cloud offloading, which achieve energy or performance savings in some given scenarios, however, available approaches still have their known issues and drawbacks. The gaps on MCA have been categorised into optimisation overhead, development inefficiency, overall inefficiency, and inadequate testing;

- **Optimisation Overhead**

Some approaches are inefficient to use as a result of the unaccounted *overhead cost* of the optimisation process. For instance, for an application where an optimisation approach always overshadows the offloading benefits during runtime, such application or task may not require MCA optimisation. Moreover, existing techniques in an attempt to automate the development process implement heavy-weight (custom) runtimes which contribute their own performance cost to the optimisation process.

- **Development Inefficiency**

Following from the first point, existing MCA optimisation approaches are based on custom runtimes which incur an overhead, moreover, these runtimes are required to be setup at the mobile and cloud tier prior to execution. Although these automate the MCA process, the identification of offloadable components in most approaches [4]–[6] are achieved manually (which is difficult to predict without execution). The few [7], [8] which automate the identification process targets already packaged legacy systems (with the aim of not modifying the code base).

- **Overall Inefficiency in Qualities**

Some approaches [5], [6] (which may handle optimisation overhead) may be overly constrained to mobile optimisation (i.e. with a focus on mobile energy and performance) to the extent of ignoring the efficiency of the cloud surrogate (which, in cloud computing research, is popularly investigated in terms of qualities such as cloud resource efficiency and service availability) – thereby compromising overall efficiency. Moreover, other cloud-aware offloading schemes [7], [8] do not reflect this awareness in their evaluation process.

- **Inadequate Testing**

As mentioned in point three, offloading schemes which are both aware of mobile and cloud tiers in their offloading decision making do not reflect the awareness in their evaluation. This is due to the lack of an appropriate MCA testing framework. The existing testing approach is focused on evaluating mobile performance and energy usage alone – i.e. mobile tier only.

1.3 Aim and Objectives of the Research

Driven by the above motivations and gaps, the vision of the research is to provide effective approaches for development and evaluation of MCAs – in other words improving the development life cycle of MCAs. To a wider extent, this would consequently enhance development efficiency, fine-grained optimisation, overall efficiency in terms of full-tier qualities and reliable testing.

Accordingly, the aim of the thesis is to convey the critical features required for the optimisation of MCA in a unified architecture.

Scope: The full-tier quality attributes investigated by the thesis include the popularly investigated mobile performance and energy usage metrics as mobile tier qualities. And for the cloud tier qualities, resource efficiency and availability, which are prevalent qualities in cloud computing research, are explored. In this thesis, the architecture is defined as a model-driven architecture which drives the development process based on meta-models and achieves automation via a design pattern implementing the optimisation logic for MCA offloading. The model-driven approach, at the current state of the thesis, targets the Android mobile devices and Amazon cloud. From the architecture, the frameworks for development and evaluation (which also fulfils the full-tier objective of the architecture) are derived. Thus, the thesis is classified under the sub-field of model-driven MCA optimisation. More specifically the objectives of the research are as follows:

- **To Develop an Approach for Full-tier Software Qualities and Green Optimisation in MCAs**

In order to mitigate the current gap in monolithically tiered optimisation – where mostly investigated metrics are associated with the mobile tier, and also to improve overall efficiency; the first objective will focus on a model-driven architectural approach to MCA development. Model-driven engineering (MDE) is a paradigm which exploits domain models to effectively solve a recurring problem. Therefore, the proposed model-driven architecture will allow for the specification of software qualities alongside green attributes for both mobile and cloud tiers. As a model-driven architecture, these qualities can be modelled at an earlier stage of development independent of any platform specific requirement.

- **To Develop an Approach to the Automation of MCA Development Process without Custom Runtime**

As mentioned earlier, custom runtimes in existing MCA offloading approaches incur optimisation overhead. To mitigate this overhead, custom runtimes have to be avoided in the development process. Model-driven engineering (MDE) simplifies the development process using models of design patterns (alongside tools) to increase productivity by automating processes. Consequently, the second objective will explore a model-driven engineering option in form of a framework for automating the development process of MCA. The framework will encompass the intricacies of the MCA development; possibly from analysis (involving identification of offloadable tasks) to design (involving platform independent conceptualisation) to implementation (involving the platform specific optimisation). Consequently, this ought to drive efficiency and productivity in MCA development, mitigating optimisation overhead and development inefficiency.

- **To Develop an Approach to the Evaluation and Comparison of MCAs**

Currently, there is no test framework that has been proposed for evaluating MCAs. Consequently leading to inconsistencies in the way testing is performed within the domain, thus; making it difficult to compare between MCA optimisation approaches. Moreover, evaluation in existing literature focuses on mobile-tier metrics; thus it is difficult to ascertain the overall efficiency of a given approach. The third objective is to develop a framework based on the Behaviour-driven Development (BDD) concept which can be applied to MCAs and useful for measuring its overall efficiency by taking the full-tier metrics (i.e. mobile and cloud impacts) into consideration.

Behaviour-driven (from BDD) is a concept that uses different parameters with simple clauses (*given, where, then*) to construct a scenario that defines an application. These scenarios can then be expected or compared.

Full-tier refers to test coverage with finer-granularity - i.e. including all participating tiers of an architecture. For MCAs, the concept of full-tier would involve both the mobile tier and the cloud tier.

- **To Conduct Case Studies and Evaluation**

For proof-of-concept, validation and evaluation, the final objective is to apply a series of real-life case studies and experiments to critically examine the proposed approaches and framework implementations. Furthermore, to establish a comparison with existing work the case studies are used to establish the efficacy of the proposed approach in terms of mobile tier qualities (performance and energy efficiency) and cloud tier qualities (resource efficiency and software availability).

1.4 Contribution to Knowledge

As shown above, the aim of the thesis to convey the critical features required for the optimisation of MCA in a unified architecture is achieved in four specific objectives. The first three objectives are focused on developing an approach towards the central aim – these are a direct connecting thread to the contributions of the thesis which are as follows;

- **Model-driven Approach for Integration of Software Quality with Green Optimisation in MCAs (Mango)**

The thesis proposes a novel model-driven architectural approach for MCA development. The approach specifies a structure of processes/phases for integrating software qualities and green optimisation objectives in the mobile tier (performance and energy efficiency) and cloud tier (availability and resource efficiency). As a model-driven architecture, this underlying structure is based on meta-models and design pattern. Meta-models are used by Mango to encapsulate and integrate the mobile tier logic and the cloud tier logic at a platform independent level; which includes the modelling of identified offloading tasks. Context-driven Requirements Analysis for Caller-Callee model (CRAC) is a proposed process for achieving the aforementioned meta-modelling. Furthermore, for platform specific implementation of the optimisation logic and realisation of MCA offloading scheme; Aspect Context Task Service design pattern (ACTS) is proposed by Mango. Consequently; by adopting a model-driven architectural approach to MCA optimisation; overall efficiency in qualities is achieved as optimisation logic can be both modelled in full-tier both independent of and specific for the platform. Moreover; the architecture, then makes it possible to implement frameworks to transform meta-models to specific application code, as presented in the next point.

- **Context-aware Architecture for Green Optimisation**

The core of the proposed Mango approach is the optimisation architecture. Mango architecture has been proposed in the thesis for achieving efficiency at runtime while taking the full-tier quality attributes into consideration. The context adopted by the approach are user and environmental contexts. The objective is that by context-awareness both from the user and environmental perspectives (rather than only environmental), an improved efficiency in target qualities can be achieved.

- **Model-based Selective Approach to Identification of Computation intensive tasks (Mosaic)**

In order to enhance development efficiency and mitigate the optimisation overhead due to custom runtimes; a model-driven framework/tool called Mosaic is proposed to realise the Mango approach in MCA development. Mosaic provides a set of features that realises the MDE triad; editor, language and generator [9] while taking into account the intricacies of MCA – i.e. surrounding identification and verification of offloading tasks. Mosaic provides a graph-based editor for meta-modelling which is based on XMI. Mosaic uses templates to specify domain specific language/structure which implements the design pattern proposed in Mango. And Mosaic provides a framework feature for generating application code using meta-models and templates. The transformation feature is used to verify that an optimisation process will most certainly yield benefits – consequently mitigating the optimisation overhead concern. Furthermore, development efficiency is achieved through high decoupling and automation fulfilled by the transformation process.

- **Behaviour-driven Full-tier Green Evaluation (Beftigre)**

In the absence of an appropriate MCA testing framework which considers full-tier implications of an optimisation scheme (i.e. implications on both the mobile and cloud tiers), this thesis proposes Beftigre. The novelty of Beftigre is its ability to define a structured approach to MCA evaluation based on the concept of Behaviour driven development – using annotations to evaluate and compare between systems by asserting expectations. The other key novelty of Beftigre is its interfacing structure from which is utilised; which is as an API (called BAND) that integrates with the Android mobile test framework, and an API and tool (called BEFOR) that integrates with the cloud tier. By providing a full-tier testing framework; Beftigre makes it possible to evaluate the overall efficiency of a MCA or its optimisation scheme. It also drives development efficiency given that all reporting and analysis from mobile and cloud are automated.

1.5 Research Method

The thesis adopts a combination of research methods including literature review and tool-based case studies.

Initially, comprehensive review of the current state of the art is undertaken with regard to green software, MCA offloading schemes, MCA evaluation approaches, and techniques found useful for mobile development – AOP and MDE. Through in-depth review and analysis of the latest literature, several issues and limitations are found on existing MCA optimisation approaches and evaluation approaches. These lead to the design and development of the series of novel approaches proposed subsequently.

To justify and evaluate the proposed MCA optimisation approach and evaluation approach, two prototype frameworks/tools are implemented (one for the proposed optimisation approach and another for the proposed evaluation approach) and a series of case studies are conducted. Utilising a number of distinct real-world applications adopted in existing works, experiments are conducted to evaluate the functionality, effectiveness/ efficiency and any other key target objectives of proposed approaches; as a way to practically re-iterate the benefits of the approach compared to existing works – this is a key criterion for success.

Papers have been published based on research outcome at each milestone. This enables valuable assessments of the work from other researchers in terms of contribution and justification within the field, and also leveraged as a crucial activity to assert the success/relevance of the research.

1.6 The Structure of the Thesis

The thesis is organised as follows:

Chapter 1 gives the introduction of the research including; the problem statement, the aim and objectives of the research and the contributions to knowledge.

Chapter 2 broadly reviews the relevant literature; including the background of green software, mobile cloud applications (MCA); and concepts in offloading schemes, and techniques found useful for mobile application development such as; aspect-oriented programming and model-driven engineering.

Chapter 3 is the problem definition and methodology chapter. The problems pertaining to the two main research areas; MCA optimisation approaches and MCA evaluation approaches, are presented and used to motivate this research. The methodology adopted by the thesis to address the identified gaps are also presented in this section, as a way to introduce the contributions of the thesis.

Chapter 4 presents a formal description of the MCA optimisation approach proposed by this research – called Mango; which is a model-driven architecture for integration of software quality with green optimisation in MCAs.

Chapter 5 presents a model-driven framework called Mosaic; which is used to transform models into application code based on the proposed Mango architecture.

Chapter 6 presents a MCA evaluation approach and framework called Beftigre; which is useful for seamless MCA testing (seamless through annotating mobile test class) with mobile and cloud metrics taken into account – thus full-tier.

In Chapter 7, using popularly adopted real-world applications, a series of case studies and experiments are conducted to illustrate and evaluate the efficacy of the proposed approach in terms of mobile tier qualities (performance and energy efficiency) and cloud tier qualities (resource efficiency and software availability). Also, the efficacy of the frameworks is demonstrated using the case studies.

Finally, Chapter 8 summarises the thesis by presenting the conclusions and the future work.

Chapter 2. Literature Review

2.1 Introduction

This chapter presents a background to green software and conducts a broad survey of many techniques that have been found useful for engineering mobile applications. Key techniques used are such as Task Offloading (based on Mobile Cloud Computing) – focused on green objectives, Model Driven Engineering and Aspect Oriented Programming are presented. These techniques are the foundation of the development of the proposed approach.

2.2 Green Software

2.2.1 Definition

The concept of green software [10]–[13]; sometimes used interchangeably with sustainable software [10]–[12], is derived from green IT or computing – which deals with the study and process of manufacturing, using, and disposing of computing hardware products with minimal impact on the environment. As a result of the success of green IT – hardware research, green software research began an investigation into applying 'green' principles from hardware products into software products and their processes. And also, similar to the direct environmental impact of IT hardware, regarding IT software; application inefficiencies like inefficient algorithms and resource usage e.g. high Central Processing Unit (CPU) usage, are sources of high energy consumption [11], [14]. As the total electrical energy consumption by computer equipment increases, there is a consequent increase in greenhouse gas emissions. Each client/personal computer in use generates about a tonne of CO_2 every year [15]. Therefore, software has indirect environmental implications.

The term 'Green software' is therefore used to refer to software applications that efficiently monitors, manages and utilises underlying resource(s) with minimised or controlled impact on the environment [10]–[13]. Green Software Engineering is a newly coined name and a branch of software engineering increasingly gaining interest, and which aims at improving existing software design and implementation approaches to achieve energy or resource efficient software. Green IT presents two key roles software plays in sustainability; [3], [11], [16] 1) as a tool to monitor and optimize the energy efficiency of any system production or operation process – also referred to as *greening by Software (or IT)*, and 2) as a target of energy efficiency initiatives – also referred to as *greening in Software (or IT)*. Green software engineering focuses on IT as a target of energy efficiency – i.e. greening in software. In the rest of the report, green software engineering will be used interchangeably with green optimisation, for brevity.

2.2.2 Green Optimisation Objectives

Various works such as [17], show that efficient resource usage by software leads to improved energy usage, and in several other works energy or power awareness of applications is achieved by control of system resources such as the CPU and memory. Furthermore, software does not have a direct environmental impact, but indirect impact through resources [13], [17], thus, the result of optimising applications by managing control on system resources is improved energy usage of the application. Therefore, (as presented in section 2.2.1);

- A key objective of green software is efficient resource usage and efficient energy usage [18], [19]. And energy efficiency and resource efficiency are often considered as congruent in the research.

The focus of green software on energy and resource efficiency was motivated by the success of green computing research for datacentres domain [20]–[22] – which largely targeted optimisation of datacentres for low *energy* and optimum hardware *resource* requirements.

Furthermore, software systems are often presented in terms of functional and non-functional requirements [23]. While functional requirements deal with the functionalities, capabilities, services or behaviours of the system, non-functional requirements (also known as quality attributes) deals with requirements that support the delivery of system functionalities. Examples of quality attributes are performance, accessibility, security and development efficiency to mention but a few [23].

So far energy efficiency and resource efficiency (popular targets of green optimisation) are often considered in context and conjunction with other software quality attributes. Hence the varying themes of green software research; *Performance (response or execution time)* and energy efficiency [24]–[26], optimal *accessibility* and resource efficiency [26], energy efficient *secure* systems [27], *development efficiency* and energy efficiency [3] etc. The practice of implementing green metrics as a quality attribute in the context of some other quality attribute(s) – such as performance, accessibility, security, etc. is often imbibed by current research as a means to explore trade-offs and possible consequences of green optimisation, for software quality assurance. This trade-off capability of the software product to meet the current needs of a set (required) functionality – say resource usage, without compromising the ability to meet future needs – say changing workload/performance, is often referred to as Sustainability of Software [10] – this is a core green software objective. Thus;

- Another key objective of green software is to achieve greenness as a quality attribute which finely integrates with other software qualities such as performance and availability. By finely integrates – meaning achieving green metrics with little or no performance (other qualities) compromise.

2.2.3 Artefacts and Approaches

2.2.3.1 Process and Product Artefact

Green software optimisation targets two main artefacts: the process and the product (code) [10], [13]. In this review, artefact is used to describe what is being optimised. The artefacts are sometimes referred to as assets. It is to be noted that green IT targets a more generic level of assets (which is due to the objective of using software as a means for environmental sustainability, rather than a target), such as product, processes, people, project infrastructure, and institutional context [10].

In the case of greening in software, the process artefact refers to the software development life cycle (SDLC) rather than general business processes. For example, optimising processes in green IT domain involves; managing how hardware products are used, operational decisions such as the promotion of electronic systems for business automation – ‘going paperless’, promoting teleconferencing to reduce carbon footprints etc.; which all drift from the SDLC.

The SDLC sometimes referred to as software development process (SDP) or simply software process is a process for planning, creating, testing, and deploying software applications. There exist many different software processes, but all must specify four key phases or activities that are fundamental to software engineering [23], [28];

1. Software specification – which refers to the functionality of the software and constraints on its operations.
2. Software design and implementation – which refers to the production of the software to meet specifications.
3. Software validation – which refers to the evaluation or testing of the software to ensure it does what the customer wants.
4. Software evolution – which refers to the evolvability of the software to meet changing customer needs.

2.2.3.2 Conceptual and Algorithmic Approach

Conceptual techniques such as architectures or models present a comprehensive plan required for achieving green software [29], and they could span through multiple phases of the software development life cycle. An example is the GREENSOFT model which adopts a layered approach to software sustainability; to structure concepts, strategies or guidelines, activities and processes for i) green software and ii) it's engineering [13]. With the GREENSOFT model, the aspect of the model which focuses on the engineering of green software (i.e. ii, as marked in the statement above), adopts a lifecycle approach to investigate optimisation concepts for various phases of SDLC. In practice, however, existing green software conceptual models do not integrate well with SDLC or software implementation in specific application domains, and thereby not utilised for specific application domains. Consequently, varying literature propose green software solutions which target a specific application domain by focusing on optimisation of the software product (as an artefact). In such cases, optimisation is achieved through implementation of efficient algorithms [30] – i.e. algorithmic approach to green software engineering.

Algorithmic approaches are techniques that directly apply to or make changes to the software code. These include i) refactoring for efficient resource usage, ii) use of energy aware custom runtimes which manipulate the programs execution or code base, or iii) green compilers or IDEs.

Refactoring techniques aim to make changes to the structural composition of the system in such a way that the new code base or optimised component uses less resource to accomplish the same or even more tasks. In green software, optimising the code base can warrant structural change which leads to a more optimised architecture. For example, the research in [31], through comparison of two commonly used distributed architectural patterns shows that the choice of architecture adopted in a software program affects its energy consumption. *Custom runtimes* are additional codebase – often independent of the functional features of the system – implemented in a software application to aid its efficient use of resource or energy. E.g. [32], [33]. Custom Runtimes are often used for executing custom optimisation logic which is otherwise foreign to the base runtime of a program. The runtimes may comprise monitors (power monitor; for energy awareness or resource monitor; for resource awareness) which monitor different environmental state in order to make an optimisation decision at runtime. *Green compilers* are used for generation of optimised codebase for efficient use of resources or energy [30], [34], [35]. Green compilers are targeted towards specific resources such as

CPU optimisation or GPU optimisation and therefore are often vendor specific as well as resource-type specific.

2.2.4 Green Software Application Domains

Green software has been explored in a number of domains, this have been broadly classified into three; desktop environments, cloud computing environments, and mobile environments. The focus of the classification is green software (not green IT hardware, which may include other domains such as embedded systems).

2.2.4.1 Desktop environments (End-user applications)

Different utility programs or applications of the same software category (i.e. fulfilling the same functions, e.g. browsers) have been shown to have varying energy or resource consumption [36]–[38]. Moreover, consumption or bloat in large applications is mostly due to deeply layered frameworks around which they are built – especially in scenarios where only a few of the features of such frameworks are utilised by the application [2]. To address such runtime bloat leading to energy inefficiencies in end-user applications, refactoring approach is proposed to use only components within a framework, that are being utilised by the application[2]. Furthermore, a number of end-user resource or energy monitoring applications have been proposed in the literature [36], [39]–[41] to keep track of resource or energy usage of applications. With the aid of such monitors, users can control applications that consume excess resource, a few of the monitors can also be set to automatically turn-off programs which are not being utilised at any point in time.

2.2.4.2 Cloud computing environments

Green software engineering has been applied in the context of cloud computing – which spans through public (datacentres), private and hybrid clouds. Due to high energy demands in data centres caused by increasing demand for cloud computing services, several approaches (e.g. [30], [42]–[44] etc.) have been proposed as a solution towards efficient management of cloud

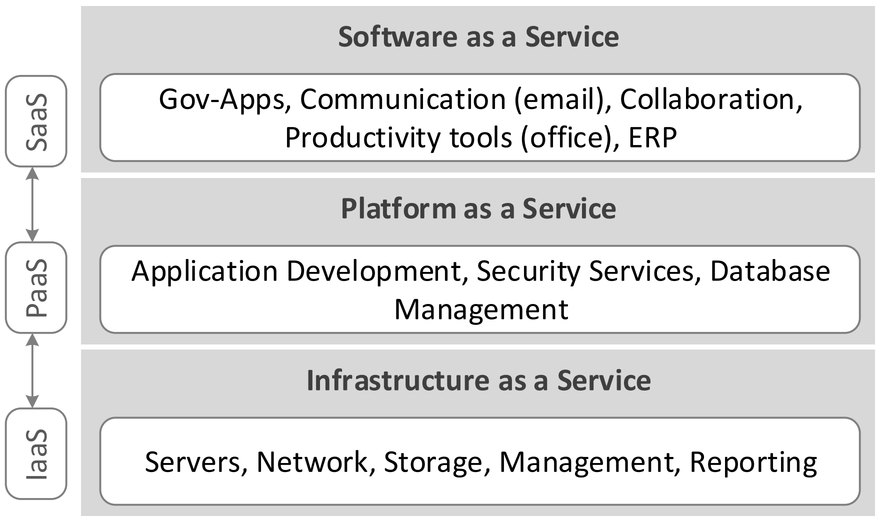

Software as a Service
Gov-Apps, Communication (email), Collaboration, Productivity tools (office), ERP
SaaS
Platform as a Service
Application Development, Security Services, Database Management
PaaS
Infrastructure as a Service
Servers, Network, Storage, Management, Reporting
IaaS

Figure 2.1 Cloud Service Models [48], [49]

resources, of which a popular software-based approach is the load balancing approach [30], [45]–[47]. Load balancing deals with even distribution of workload across interconnected servers to mitigate overutilization (high consumption/runtime bloat) or underutilisation (runtime waste) of resources [45]–[47].

Cloud computing, also referred to as 'the cloud', provides three main service models – Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [48], [49] as presented in Figure 2.1. As the key objective of cloud computing (from the cloud provider perspective) is outsourcing services based on pay-per-use – much similar to the telecommunication 'Pay as you Go' mobile cost-effective model, the consequence has frequently led to increasing cloud resource consumption in usual (non-optimised) scenario or day-to-day operations [30], [45], [46]. Most cloud providers (especially, of IaaS) adopt green ICT models which focus on hardware sustainability – such as datacentre cooling mechanisms, re-design of datacentres for energy management by sensors. However, hardware sustainability techniques do not directly handle the resource misuse of consumers. As a consequence, administrative software systems (which include power-aware algorithms such as shown in [30]) are being adopted and are built in a way to monitor, and efficiently allocate tasks to servers – an application of green software to the cloud, however, focus is on using software as a means to attain resource efficiency (i.e. greening by software). Recent advances in green software (which focuses on optimised software; greening in software), has also had a positive impact on the cloud (SaaS and PaaS). In which case, software services provisioned by the cloud are not only orchestrated externally (to minimise excess resource usage by high demand on a resource) but are also optimised based on the awareness of the internal software features (which minimises resource usage based on user context-awareness) [46]. For example; Chinenyeze et al. [47] presents an aspect-oriented model for energy efficiency (AMEE) in decentralised servers. AMEE is focused on the use of Aspect component for resource optimisation through load-balancing based on awareness of application context – i.e. finer-grained application control.

2.2.4.3 Mobile environments

In mobile environments – such as smartphones and tablets – software applications are used to monitor resource utilisation and control the execution of mobile applications, in such a way as to extend battery life. A typical scenario is the use of such applications (e.g. [50]) to monitor and shutdown services which are not being used – these are examples of greening by software. Moreover, advances in green software have investigated techniques for improving the greenness of mobile application at finer granularity [51], [52] such as simple algorithmic/rule-based optimisation, computation offloading, context-aware resource management within applications etc. With the ever increasing demand on the mobile device and its resource constrained nature, the research on green mobile optimisation is experiencing continuous attention and interest by academic research and the industry. With advances in cloud computing, the popular green software optimisation approach for mobile devices is towards the use of cloud as a surrogate to enhance execution of mobile tasks – this phenomena is commonly known as Mobile Cloud Computing (MCC). This thesis focuses on the mobile cloud applications aspect of mobile domain.

2.3 Mobile Cloud Applications

Mobile Cloud Applications (MCAs) are mobile applications optimised by the use of cloud computing as a surrogate for execution of resource-intensive tasks. Thus, MCAs are a product of the MCC paradigm.

The mobile tier of the MCA is composed of the mobile device, whereas the cloud tier is popularly implemented as clouds or fogs (cloudlets). Fogs or cloudlets are installations of small datacentres at designated locations and connected to larger cloud servers via the internet. Fogs are much closer to the end-user device than the cloud; with the aim of providing mobility at the cloud tier [53].

A number of MCC research also proposes the use of mobile services at the cloud tier – which is similar to cloud services but provisioned by a collection of mobile devices. In other words, mobile devices are considered as providers of cloud, making up a peer-to-peer network as in [54]–[56]. This is also a form of fog computing, however, the focus is on the use of mobile devices for cloud provisioning, rather than cloudlets.

From a greening in software perspective, MCAs are realised through a technique known as Offloading [1], [53]. Consequently, various research proposes offloading schemes around the optimisation of mobile performance and energy usage, focused on the use of high computation resources as a surrogate – whether as clouds or as collaborating mobile resources, i.e. fogs, [4]–[7], [52].

2.3.1 MCA Offloading Schemes

Task offloading is an algorithmic mobile optimisation technique that involves the transfer of computation or resource intensive tasks of a mobile application to a remote system (cloud or fog) with higher processing capability for execution [52], [57]. Existing offloading schemes employ both code refactoring techniques (i.e. transformations) and the use of custom runtimes; thus an algorithmic approach (as presented in section 2.2.3). Custom runtimes can

Table 2.1 Comparison of offload models (derived from [4]–[7], [59])

System	Identification Mechanism		Decision Maker		Offloading Mechanism	
	Type of Transformation	*Level of Granularity*	*Type of Threshold*	*Used Parameters*	*Offloading Type*	*Automation System*
Kwon et al. [4]	Manual (Annotation)	Low	Static	Resource	Cloning	Custom Runtime
MAUI [5]	Manual (Annotation)	Low	Dynamic	Resource	Cloning	Custom Runtime
Hassan et al. [7], [8]	Automated	High	Dynamic	Resource (full)	Partitioning	Partial Runtime
Native Offloader [59]	Automated	High	Dynamic	Resource	Partitioning	Partial Runtime

N.B. This table is not an exhaustive list of the models, but a list of distinct representative models – consisting unique characteristics.

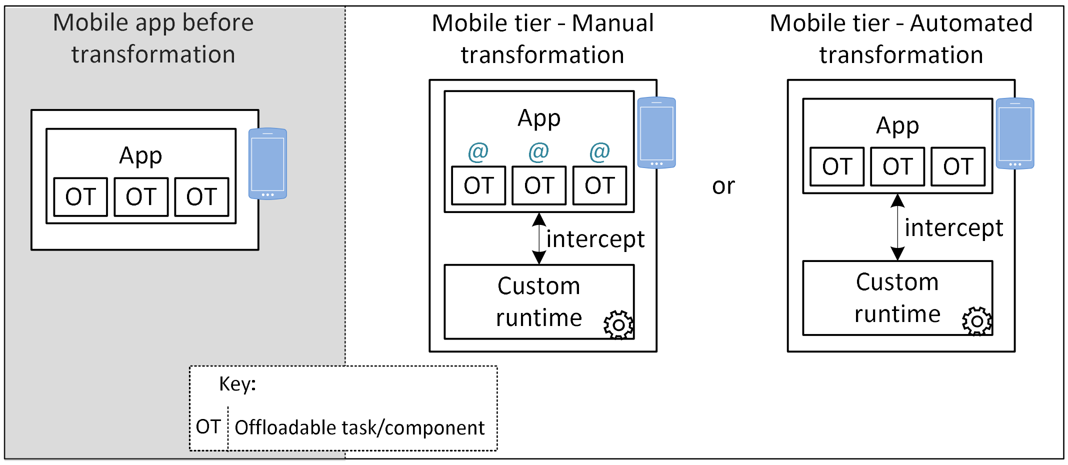
execute as background processes which encapsulate the MCA offload model for a mobile platform. Android is the most popularly investigated mobile platform for MCA as shown in the literature, e.g. [4]–[7], [52], [58] to cite but a few.

MCA offload models or schemes typically consists of three key components which are; identification mechanism, decision maker and offloading mechanism. These components can be defined by a number of properties (as shown in Table 2.1).

2.3.1.1 Identification of Offloadable Task

The identification mechanism as shown in Table 2.1 is defined by two properties: the type of transformation and the level of granularity.

- The type of transformation refers to the technique used to analyse and mark tasks as offloadable. This can be either manual or automated.
- The level of granularity is defined by how many of the three MCA components an offloading model takes into account during the process of identification of offloadable tasks.



Mobile app before transformation
App
OT
OT
OT
Mobile tier - Manual transformation
App
@
@
@
OT
OT
OT
intercept
Custom runtime
or
Mobile tier - Automated transformation
App
OT
OT
OT
intercept
Custom runtime
Key:
OT Offloadable task/component

Figure 2.2 Types of Transformation in MCA (derived from [4]–[7])

To transform a mobile application into an MCA, identification of offloadable tasks is a sine qua non activity. This can be achieved either manually or automatically.

Schemes classified by *manual transformation* require source code modification for identification of offloadable task. As illustrated by Figure 2.2, in manual transformations, *annotations* are used by the developer to identify methods of the code that are resource intensive [4], [5]. The challenge with the manual identification of offloadable components is that it is difficult to ascertain which components are actually resource-intensive prior to execution/runtime. Moreover, a manually identified task may be tightly coupled to a resource constrained code (even if the identified task is actually resource-intensive). Also, since manual transformation does not follow any systemic approach (such as static or dynamic analysis) to identify offloadable task, it therefore cannot account for the general impact of other MCA components. Thus, manual transformation possess low level of granularity in identification of offloadable tasks (e.g. [4] and [5]).

Schemes classified by *automated transformation* do not require source code modification in the identification of offloadable tasks. The automated transformation approach makes use of *static* and *dynamic analysis* of the application to identify the offloadable tasks [7], [8]. The purpose of the static analysis is to filter out methods that are resource-constrained or tightly coupled to other resource-constrained methods. Static analysis is achieved by performing a call-graph analysis on the bytecode of the application (whether packaged or not). The purpose of the dynamic analysis is to estimate that a statically identified offloadable task yields benefit when executed remotely in the cloud. This estimation is achieved by comparing the local execution time of the offloadable task against its remote execution time. While the static analysis does not require execution of the program, dynamic analysis requires execution of the program, and also requires that the offloadable task is setup in the cloud prior to the analysis.

Since current automated transformation do not require source code modification (i.e. no need for annotations), the custom runtime stores the method signatures of offloadable tasks and intercepts any methods at runtime, which have their signature stored in the repository of the custom runtime [6], [7]. Automated transformations are explored for legacy systems – where source code may not be available for refactoring, while manual approaches are explored mostly in application development or scenarios where source code is accessible.

As shown by Table 2.1, automated transformations are marked as possessing high (implying, better) level of granularity in the identification of offloadable components as it includes runtime (dynamic) analysis.

2.3.1.2 Decision Making

The decision maker as shown in Table 2.1 is defined by two properties: the type of thresholding and used parameters.

- The type of thresholding refers to the adaptive capability of the offloading model – which can be either static (where thresholds are fixed) or dynamic (where thresholds adapt to environmental state, based on machine learning algorithms).
- Used parameters refers to the type and number of parameters (or environmental factors) used for implementing the thresholds. Some of

the factors include mobile CPU availability, network bandwidth, latency, etc. Details are presented in section 2.3.2.

Decision making is a feature in offloading schemes and used to decide when to offload or when not to offload. Decision making can be based on simple algorithms such as static thresholds [4] or can be as complex as machine learning algorithms including the use of multi-layer perceptron [7]. In either case, these algorithms employ varying environmental factors in offload decision making. Hassan et al. [7] suggests that the more the environmental factors are considered in the decision-making process, the greater the depth of accuracy of the decision maker. However, accuracy is traded-off for an element of overhead due to much monitoring as shown in [7]. An important objective of this thesis is to minimise overhead by using a time-based context-aware approach (a key feature of the later presented Mango approach). The intention is that by using time as a singular parameter for thresholding, the overhead resulting from extensive monitoring of many resources can be reduced.

2.3.1.3 Remote Execution of Offloadable Task

The offloading mechanism as shown in Table 2.1 is defined by two properties: the offloading type and the automation system.

- The offloading type refers to the mechanism by which the identified offloadable task is offloaded to the cloud.
- The automation system refers to how automation of all MCA components/process are achieved.

The characterisation by remote execution of offloadable tasks is also referred to as the *offloading mechanism* of the MCA offloading scheme by some literatures [4], [7]. This feature describes the structural composition of the cloud-tier after the MCA refactoring process. To execute the offloadable tasks remotely, the cloud tier can either be setup as a clone of the mobile device (i.e. cloning) or as independent components executed remotely (i.e. partitioning).

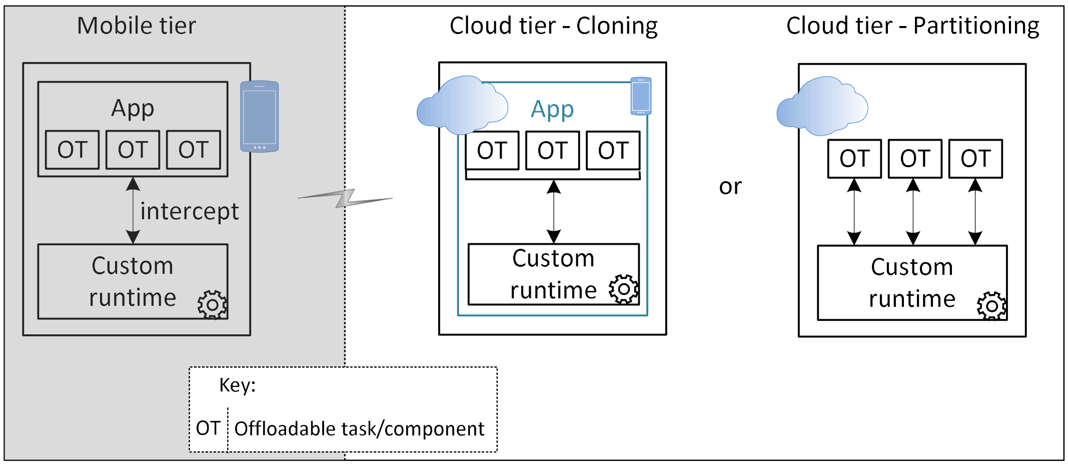


Figure 2.3 Types of Offloading Mechanism in MCA (derived from [4]–[7])

Cloning [4]–[6] involves the setup of a virtual mobile device in the cloud (as illustrated by Figure 2.3). The full mobile application is also installed on the virtual device and executes remotely at the same time as the local application. The cloning approach works by state synchronisation/checkpointing. In other words, when a check pointed state (i.e. thread) is reached, a snapshot is created for fault-tolerance and the state of execution is offloaded to the cloud which continues execution on the virtual device (on the cloud), after which the final state (of remote execution) is synchronised with the local state.

Partitioning [7], [52] involves the setup of identified offloadable task as independent components in the cloud. In partitioning, virtual device is not required. Partitioning works by using sockets to transmit execution parameters to the cloud. The component in the cloud listens for socket connections and processes the mobile request using the parameters sent. Response is in turn sent to the mobile tier after execution using socket API.

2.3.2 Environmental Factors affecting MCA Decision Making

Varying offloading schemes proposed by existing literature makes decisions based on (monitoring) a collection of varying environmental factors of MCA (such as, data size as in [4], network bandwidth and latency as in [5], etc.) The awareness or monitoring of environmental factors is also referred to as context awareness [60]–[62]. As shown in Table 2.2 , these factors are proposed by existing literature as impacting MCAs and used for offload decision making.

Table 2.2 Decision making factors in MCAs.

System	Used Parameters (Resources monitored for thresholds)						
	Mobile CPU availability	*Mobile memory availability*	*Cloud CPU availability*	*Cloud memory availability*	*Network bandwidth*	*Network latency*	*Data size*
Kwon et al. [4]							✓
MAUI [5]					✓	✓	
Hassan et al. [7], [8]	✓	✓	✓	✓	✓	✓	✓
Native Offloader [59]		✓			✓		

- Mobile CPU availability

Mobile CPU availability is measured in percent and is of particular importance for computation intensive tasks. In other words the lower the percentage CPU availability, the greater the chance of mobile energy consumption or performance compromise for a computation intensive task. Thus; the objective is to execute a (computation intensive) task on the mobile device when the percentage CPU availability is higher or at least above a set threshold. Mobile CPU availability is obtained programmatically by examining the */proc/stat* files in Android to compute the percentage CPU available.

- Mobile Memory availability

Mobile memory availability is measured in percent and is particularly of importance for data intensive task. In other words the lower the percentage memory availability, the greater the chance of mobile energy consumption or performance compromise for a data intensive task. Thus; the objective is to execute a (data intensive) task on the mobile device when the percentage memory availability is higher or at least above a set threshold. A higher bound mobile CPU and memory availability are useful for determining when to execute a task on a mobile device. Mobile memory availability is obtained programmatically by examining the */proc/meminfo* files in Android to compute the percentage memory available.

- Cloud CPU availability

Cloud CPU availability is measured in percent and is particularly of importance for computation intensive task. In other words the lower the percentage CPU availability, the greater the chance of mobile energy consumption or performance compromise for a computation intensive task. Thus; the objective is to execute a (computation intensive) task on the cloud when its percentage CPU availability is higher or at least above a set threshold. The notion is that avoiding offload to the cloud when the cloud CPU is overworked can curtail mobile performance compromise. Cloud CPU availability is obtained programmatically by examining the */proc/stat* files in a Linux-based server to compute the percentage CPU available.

- Cloud Memory availability

Cloud memory availability is measured in percent and is particularly of importance for data intensive task. In other words the lower the percentage memory availability, the greater the chance of mobile energy consumption or performance compromise for a data intensive task. Thus; the objective is to execute a (data intensive) task on the cloud when its percentage memory availability is higher or at least above a set threshold. The notion is that avoiding offload to the cloud when the cloud memory is overworked can curtail mobile performance compromise. A higher bound cloud CPU and memory availability are useful for determining when to offload a task to the cloud. Cloud memory availability is obtained programmatically by examining the */proc/meminfo* files in a Linux-based server to compute the percentage memory available.

- Network Bandwidth

Network bandwidth is the average rate of a successful data transfer through a network communication path. It is measured in bits per second and achieved programmatically by sending packets to and from the server to measure the bandwidth. The objective of monitoring the bandwidth is to offload a task when the bandwidth is higher than a set threshold. The notion is that the higher the bandwidth the greater the tendency for mobile energy or performance savings.

- Network Latency

Network latency is the time interval or delay between request and response over a network communication path. It is measured in milliseconds and similar to bandwidth, it is achieved programmatically by sending packets to and from the server to measure the latency. The objective of monitoring the latency is to offload a task when the latency is lower than a set threshold. The notion is that the lesser the latency the greater the tendency for mobile energy or performance savings.

- Data size

Data size is the size of the data transmitted over the communication network. It is measured in series of bytes (i.e. B, KB, and MB) and can be achieved programmatically by checking the byte size of the request packet prior to client socket transmission. The objective of monitoring the data size is to offload a task when the data size is lower than a set threshold. The notion is that transmitting larger data packets over the network could result in increased energy usage or performance compromise.

As mentioned earlier in section 2.3.1.2, the challenge of making offload decisions by monitoring the environmental factors is that it contributes performance overhead in MCAs at runtime. An important objective of this thesis is to minimise this overhead by using execution time as the key factor for making offload decisions. Further details behind the concept are presented in Chapter 3.

The popularly investigated metrics in the literature for MCA offloading schemes are performance and energy efficiency. These are also observed in the objective of the monitored environmental factors presented above. The

Green metrics of MCA investigated by this research are presented in the following section. Note that the environmental factors are different from the MCA green metrics, as the environmental factors (presented in this section) are used for deciding when to perform an offload, whereas the MCA associated green metrics (presented in the following section) are used to evaluate the efficiency (or useful savings achieved) within the MCA.

2.3.3 MCA Associated Green Metrics

Following the green software objective (section 2.2.2), the green metrics have been identified as energy and resource efficiency [18]. However, the core investigated green metrics for MCA is *mobile energy efficiency*, as the focus of MCA offloading schemes is optimisation of the mobile application. Moreover, the MCA also involves the cloud-tier thus this research also presents *cloud resource efficiency* as a relevant green metric for MCA. Furthermore; green software objectives also investigate trade-off based on other software qualities. And the popularly investigated software quality in the literature for MCA offloading schemes is *mobile performance*, in this research *software availability* is also investigated (at both mobile and cloud tiers) as a relevant software quality for MCAs.

2.3.3.1 Mobile Performance

According to Bass et al. [23], performance is defined to be how long it takes an application to respond to an event. The key drive to the advancement in mobile computing is the portability of mobile devices – which is defined by fluidity and ease of operation [57], [63], [64]. From a user perspective, the ease of operation or usability of a mobile application is critically dependent on its performance. Thus performance is a crucial metric in MCA (as considered in the MCA literature e.g. [4]–[7], [52] to cite but a few), moreover mobile performance is popularly explored in the context of MCAs as a trade-off software quality to mobile energy efficiency.

In MCA, mobile performance is often measured by computing the difference between the time of call (or request) to an offloadable task and the time of result (or response) after execution of the offloadable task. While *call* and *result* refer to a scenario where the offloadable task is executed on mobile, *request* and *response* refer to when offloaded to the cloud. Time (mentioned above) is a representative of timestamp – which is often computed programmatically using the Java timestamp utility as in [31], and is measured in ms.

2.3.3.2 Mobile Energy

According to Johann et al. [34], energy efficiency is the ratio of useful work done to used energy. In other words, it is the amount of energy incurred for executing a task.

Energy efficiency is derived from three quantities; power, time and work done [3], [34] – in this way, energy efficiency is used in the comparison of two or more entities where their useful work done is likely to vary, as the case of [31], [34]. However, in a situation of comparison between entities of similar work done or singular evaluation, energy efficiency is congruent to energy usage (i.e. using two quantities; power and time). Consequently, the term energy efficiency and energy usage (or energy used), is used interchangeable in MCA research. Power Tutor [65] is a popularly adopted model or tool used by the literature [4]–[7], [52] for mobile power monitoring.

2.3.3.3 Cloud Resource

Achieving cloud resource efficiency in a mobile cloud environment requires care so as to not compromise mobile performance. (As highlighted by the objectives of the cloud-based environmental factors – section 2.3.2). Resource efficiency in servers (the cloud) is often achieved through load balancing [47] (see section 2.2.4.2).

Although cloud resource efficiency is not often explored in the research on MCAs, investigating resource efficiency/usage for cloud can be achieved using the core impacted resource of the cloud – i.e. CPU and memory resource. Moreover, these relate to the key aspects of application taxonomy in MCA (i.e. computation and data intensive taxonomies, see section 2.3.4). Thus for cloud resource usage of MCAs; percentage CPU utilisation and memory utilisation are the key metrics. Percentage CPU utilisation and memory utilisation can be measured by examining the */proc/stat* and */proc/meminfo* files respectively in a Linux-based server.

2.3.3.4 Software Availability

According to Bass et al. [23], availability is the probability that a system will be operational when it is needed. In other words, availability is concerned with avoiding system failure. Most research does not take software availability into consideration in the implementation of MCA schemes (this category of schemes use only network exception catch, e.g. [5], [6], [52]). Moreover, a few studies which consider availability investigates only at the mobile tier, e.g. [4]. Availability is achieved at the mobile tier by implementing a time limit to how long the mobile device can wait on the cloud to complete the execution of a request. When the time limit is elapsed the execution is made on the mobile tier. Availability can also be implemented in a similar manner for the cloud tier – this is presented by Mango in this research. While the time limit for mobile tier targets the total elapsed time – which includes the network to-and-fro communication time and the cloud execution time, the time limit for the cloud tier targets only the cloud execution time. Thus for a finer granularity, the time limit used at the mobile tier is more than that of the cloud tier.

Availability is an execution quality [66], and thus realised by several time associated measures such as mean time to failure (MTTF), mean time to repair (MTTR) and failure rate [67], [68]. MTTF and MTTR are measurable in ms. Furthermore, since performance – an execution quality – is popularly used in the mobile tier, in this research we explore availability for the cloud tier – as shown later in the case studies in Chapter 7. Availability in the cloud tier is measured using MTTR – which is achieved by measuring the time (in ms) it takes for the cloud tier to execute a task (or optimally react) in an adverse condition.

Table 2.3 Application Taxonomy

System	Taxonomy (based on case studies used in the literature)		
	Computation-intensive	*Data-intensive*	*Hybrid*
Kwon et al. [4]	NQueen [69], Mezzofanti [70]	Picaso [71], MatCalc [72], MathDroid [73], ZXing [74]	Droidslator [75]
Hassan et al. [7], [8]	Mezzofanti [70], JJIL [76], OsmAnd [77]	ZXing [74]	Droidslator [75]
Dpartner [52]	Linpack [78]	-	Andgoid [52], XRace [79]

Note that the references appended to the Apps links to the source code or google play app.

2.3.4 Application Taxonomy

Application taxonomy defines the classification for applications in which MCAs have been explored in the literature. Application taxonomy for MCAs has been derived by exploring the case studies used in the evaluation of offloading schemes in the literature; as shown in Table 2.3. Furthermore; the offloading schemes used to generate the taxonomy vary across the sampled schemes in 1, and are based on Android applications. POMAC[2] [7], [8] characterised under automated transformation schemes. EFDM[3] characterised under manual transformation schemes and cloning schemes. DPartner is characterised under partitioning.

MAUI [5] demonstrates a resource-intensive application (with fast battery consumption) by implementing features within a synthetic application to perform a large bulk-data transfer over the Wi-Fi interface, consume the entire CPU, and keep the display backlight on. Such resource-intensive applications motivate the MCA research. As shown in Table 2.3, these applications can be classified in three taxonomies according to their resource consumption, they are: Computation-intensive, Data-intensive and Hybrid applications.

[2] POMAC: Properly Offloading Mobile Applications to Clouds
[3] EFDM: Energy-Efficient and Fault-Tolerant Distributed Mobile Execution

2.3.4.1 *Computation-intensive Applications*

Computation-intensive applications are a class of mobile applications that are highly (or significantly) dependent on the computing power (i.e. CPU resource) of the mobile device. An application or process is categorised as being highly dependent on the CPU usually based on the high frequency of the use of the CPU resource within a given time, for example:

- Tasks that iteratively compute a mathematical function, in categories of benchmark applications and complex algorithmic applications e.g. Linpack, NQueen.
- Tasks that frequently acquire and compute sensor data within the mobile device, in categories of GPS applications and games e.g. OsmAnd, Mezzofanti.

The core benefits of offloading schemes are realised with computation-intensive applications; as this class of applications consumes the most battery (or energy) from the mobile device. Most computation intensive applications fall into the category of gaming applications and media processing applications (i.e. image/video processing applications such as face recognition apps, optical character recognition apps etc.) as shown in the literature, for example, [5], [7], [8], [80].

2.3.4.2 *Data-intensive Applications*

Data-intensive applications are a class of mobile applications that devote most of their processing time to I/O and manipulation of data. Due to the focus on manipulation of data, these applications make more use of memory and network than the processing power. In other words,

- tasks that hold and frequently read in-memory data for computation, in categories of arithmetic computations, e.g. MatCalc.
- tasks that offload large dataset across the network, in categories of barcode decoders and face recognition applications, e.g. ZXing and Picaso.

In most scenarios data-intensive applications do not consume significant mobile energy, unless in situations of poor communication network or situations where they also require extensive computation – i.e. hybrid applications. Studies [4], [8] have emphatically shown that offloading applications that do not consume significant mobile energy (such as data intensive applications), can result in mobile performance compromise or even a slight increase in energy usage.

2.3.4.3 Hybrid Applications

Hybrid applications are applications where the identified offloadable task is both computation intensive and data intensive. A typical example of hybrid applications is video streaming, e.g. Droidslator, and online gaming applications. This category of applications consumes both network resource intensely while performing CPU demanding tasks.

As mentioned earlier, offloading such tasks is likely to save energy only if they consume significant mobile energy.

2.3.5 MCA Evaluation Approach

The MCA architecture generally comprises three sets of components [4], [5], [7], [52], these are, the mobile offloadable components (MCs), the server components (SCs) and the components of the Offloading scheme/model. Mobile offloadable components are first identified[4]. These components are subsequently replicated on the server (as server components) to improve mobile resource usage. The third set comprises the offloading scheme components – which handle decision making based on monitoring data of the current environmental state to predict when offload process is beneficial. The scheme can be launched in both tiers – mobile and cloud tier (as shown at the top middle section of Figure 2.4, i.e. (b)). The evaluation section of Figure 2.4 (i.e. (c)) illustrates the mobile-centric architecture scenarios method of

[4] The analysis for offloadable classes is necessary due to the inadequacy of the cloud to perform some mobile specific functions such as those tied to sensors, cameras, GPS etc.; and also to identify the application features that are resource-intensive.

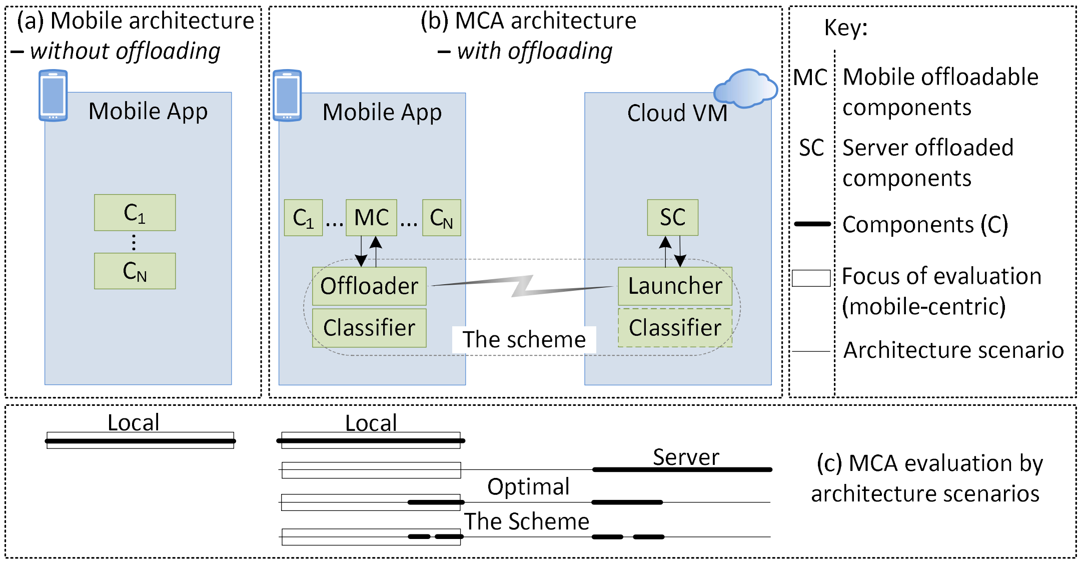

(a) Mobile architecture – without offloading
Mobile App
C1
CN
(b) MCA architecture – with offloading
Mobile App
C1 ... MC ... CN
Offloader
Classifier
The scheme
Cloud VM
SC
Launcher
Classifier
Key:
MC Mobile offloadable components
SC Server offloaded components
Components (C)
Focus of evaluation (mobile-centric)
Architecture scenario
Local
Local
Server
Optimal
The Scheme
(c) MCA evaluation by architecture scenarios

Figure 2.4 MCA architecture with mobile-centric architecture scenarios

evaluation adopted by current literature, which is based on two features: the focus on mobile-tier and the use of architecture scenarios.

2.3.5.1 Focus on Mobile-tier

To evaluate the efficiency of such systems (MCA with offloading schemes) two abstract phases are taken into account, they are mobile device test and server test; however, most research focus is on the mobile end [4], [5], [7], [52]. For example, various works [4], [7], [52] achieve rigorous tests by randomization of environment conditions such as network (bandwidth and latency) and server/cloud resource (CPU and memory), using throttling and load generation respectively, in order to evaluate mobile power usage and performance. Consequently, the results of the evaluation do not highlight the scheme's implications on the cloud tier (the focus of evaluation is depicted by the solid block around each scenarios in section c of Figure 2.4). Although the implementation of the MCA offloading schemes takes into full account the architectural change, the evaluation of the schemes does not clearly account for the impact of the scheme on the full system tiers. This and further challenges pertaining to the current MCA Evaluation approach are presented in section 3.3.

Table 2.4 Architecture Scenarios for MCA Evaluation

Summary of Scenarios	**ST** [4] **Specific Scenarios**	**POMAC** [7] **Specific Scenarios**
Local	Smartphone only	OnDevice
Server	Offloading w/All objects	OnServer
Optimal	Offloading w/Necessary objects (delta)	Optimal
The Scheme	Offloading w/Threshold check	POMAC

N.B. The summary column is used to match the scenarios from the literature [4] and [7].

2.3.5.2 Use of Architecture scenarios

The literature uses architecture scenarios to evaluate MCA offloading schemes. These scenarios currently vary between literatures. For example [7] defines four scenarios (OnDevice, OnServer, Optimal and POMAC) to evaluate POMAC. While [4] defines five scenarios (Smartphone only, Offloading w/All objects, Offloading w/Necessary objects, Offloading w/Necessary objects - delta, and Offloading w/Threshold check) to evaluate its proposed static thresholding scheme. Although the variability of architecture scenarios in current literature poses a challenge to comparing existing schemes (see section 3.3.1), these can be summarised based on similarities between scenarios as shown in Table 2.4 and illustrated by Figure 2.4.

Local is the execution of the application without any offloading. *Server* is a scenario where all offloadable objects are always executed on the server. *Optimal* is a scenario where only assessed objects are offloaded. *Assessed objects* are the objects identified as computation or data-intensive. *The Scheme* is based on extending the previous optimal scenario with decision-making[5] mechanisms for offload. It refers to the proposed offloading schemes in the literatures.

[5] Decision making is the check on the environment conditions of the communications which influence the offloading. Decision making mechanisms can be based on single (static) thresholds [4] or predictive learning [7].

2.4 Aspect Oriented Programming

2.4.1 Definition and Terms

Aspect-oriented programming (AOP) provides a component-based approach to the implementation of crosscutting concerns[6] [81]. The concept of code injection at points of execution is one of the core features of AOP. A number of libraries implement the AOP concept, such as AspectJ for Java and PostSharp for .NET. AOP provides two types of crosscutting;

- Dynamic crosscutting – which modifies the behaviour of the program. Dynamic crosscutting is fulfilled by join points (via pointcuts and advices) which makes it possible to define dynamic structure of crosscutting concerns [81]. in AspectJ and,
- Static crosscutting – which modifies the static structure of the types (classes, interfaces, and other aspects) and their weave-time behaviour [81]. Static crosscutting is fulfilled by forms known as introductions or inter-type member declarations.

AOP is often used for safety checks, logging and other concerns that can exist across class or method definitions – hence crosscutting concerns. The crosscutting concerns in which the AOP technique is explored is largely based on the modification of software behaviour; thus dynamic crosscutting dominates the use of AOP [81]. In AOP a weaver is used to apply aspects (or weave crosscutting concerns) into a target object to create a new, proxied object. The two types of weaving are as follows;

- Static weaving – performs the weaving before compilation. It is efficient in producing highly optimised woven code whose speed is comparable to that of the code written in traditional methodologies (without AOP techniques) [82].

[6] A crosscutting concern refers to a requirement or program which (although in most cases is not a functional requirement of the system) is or has potential to be *recurrent* in various parts of the program.

- Dynamic weaving – also known as load-time-weaving (LTW), dynamic weaving is performed after compilation; in load-time or runtime. Dynamic weaving is useful for reloading objects during execution [83] and as shown by Chinenyeze et al. [84] it is also particularly efficient in energy evaluation of components of similar structures.

From a Java programming context, AspectJ is the canonical open-source Java library which finely implements AOP concepts. It adds to Java a few new constructs: pointcuts, advice, inter-type declarations and aspects.

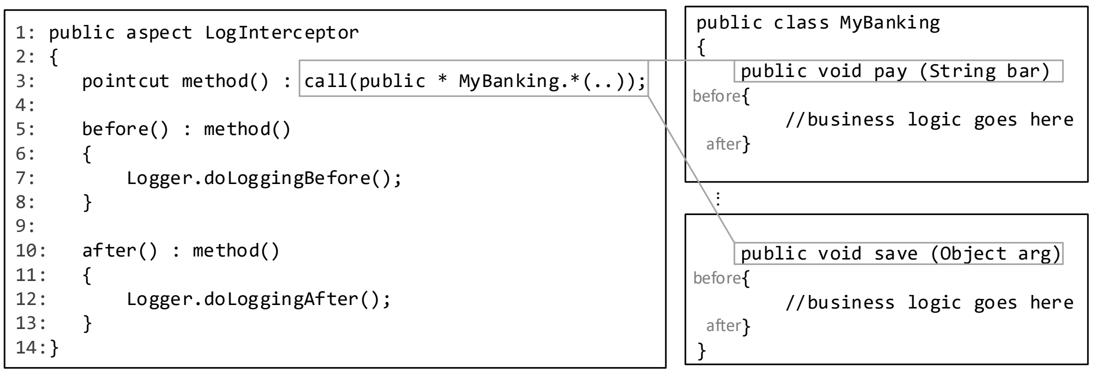
Aspect. An aspect is an encapsulation of these new (aforementioned) constructs and acts as the unit of modularity for crosscutting concerns, – analogical to Java classes, in behaviour. As seen from the examples in the following section, crosscutting code is implemented once as aspects (`LogInterceptor` of Figure 2.5 and `Runner` of Figure 2.6).

For clarification of some other AOP associated terms, some definitions have been given by exemplification in the following section.

2.4.2 AOP by Example

2.4.2.1 Dynamic Crosscutting

The Logger example presented in Figure 2.5, demonstrates how AOP can be used (in a dynamic crosscutting manner – using pointcuts) to alter the dynamic behaviour of a system (without directly modifying the original source code).



```
public aspect LogInterceptor
{
    pointcut method() : call(public * MyBanking.*(..));

    before() : method()
    {
        Logger.doLoggingBefore();
    }

    after() : method()
    {
        Logger.doLoggingAfter();
    }
}
```

```
public class MyBanking
{
    public void pay (String bar)
before{
        //business logic goes here
 after}
    ⋮
    public void save (Object arg)
before{
        //business logic goes here
 after}
}
```

Figure 2.5 Dynamic crosscutting in AOP (showing pointcut and advices)

The example presented in Figure 2.5 is based on the traditional AspectJ syntax. Some AspectJ extensions also provide annotation based syntax for AOP as shown in [81].

Pointcut. A pointcut is a program construct that selects join points and collects join point context or data. In object-oriented programs join points consists of operations such as method calls, method executions, object instantiations, constructor executions, field references and handler executions [81], [85]. Pointcuts and advice dynamically affect program flow [85] and will be adopted for our model implementation. And within the aspect, the point of execution where the code is to be woven is specified – as a pointcut, (Line 3 of `LogInterceptor`).

Advice. In AOP behaviours are added using Advice. An Advice is the actual snippet of code that can be executed before, after or around the pointcut. In other words, an advice defines the code to execute upon reaching selected point(s) of execution. For example, the specifier keywords `before` and `after` in lines 5 and 10 respectively are used to specify when the logging advice in lines 7 and 12 are to be executed within pointcuts.

2.4.2.2 Static Crosscutting

The Runner example presented in Figure 2.6 illustrates how AOP can be used in a static crosscutting manner to change the structure of a class. As mentioned earlier, while dynamic crosscutting changes the way that a program executes – using join points, static crosscutting affects the static structure of the program – using an introduction.

Introduction. An introduction is a member of Aspect that defines or modifies a member of another type such as a class. The *declare* keyword used in line

```
1: public aspect Runner
2: {
3:    declare parents: Sample implements Runnable;
4:    public void Sample.run() { ... }
5: }
```

```
public class Sample
{
    public void method ()
    {...}
}
```

Figure 2.6 Static crosscutting in AOP (showing declare and inter-type)

3 of Figure 2.6 is an example of an introduction. While join points are used for dynamic crosscuts, introductions are used for static crosscuts.

In the example above (in Figure 2.6) the Runner aspect makes the Sample class runnable by first; declaring that the Sample class fulfils the Runnable interface, using the declare keyword on line 3. And second; defining the appropriate inter-type `void run()` method as `public` on line 4.

2.5 Model-Driven Engineering

As mentioned earlier (in section 2.3.1), existing offloading schemes employ the use of custom runtimes; with the aim of automating the optimisation process of MCA. These custom runtimes, however, contribute overhead during MCA execution (as presented later in Related Work; Chapter 3). In traditional software development, there are many approaches to accelerate design and development. Among these approaches, Model-driven Engineering (MDE) has received attention because it provides abstraction through high-level models (e.g. UML), consequently facilitating the implementation of (complex) software [86]. MDE and Model-driven development (MDD) are often used interchangeably, the same is applicable to this section.

2.5.1 Definition and Terms

Model-driven engineering (MDE) focuses on exploiting domain models to effectively solve a recurring problem. Consequently, MDE simplifies the development process using models of design patterns (alongside tools) to increase productivity [9], [87]–[89]. Popular objectives of MDE are the realisation of generic models, i.e. platform independent models (PIM) and specific models; i.e. platform-specific models (PSM) and the transformation between the two. PIM is the most abstracted form of a model, and are the blueprints from which PSM are derived. At PSM, software artefacts can be realised using tools. The popular features that make the triad of platform/domain-specific modelling (PSM) are: editor, language and generator [9].

An *editor* allows users to modify the template of the model or program the language for the model. The domain specific *language* (DSL) ensures that the program is correct by defining a language or template structure. While the *generator* is used to generate executable implementations of the model – a process also known as transformation. It is also useful for generating additional software artefacts that are synchronised with the model. Furthermore, they synthesise artefacts from models to ensure consistency between application implementation and information related to functional and quality requirements captured by the models [9], [87].

In other words; MDE is applicable to various SDLC phases, thus; it can be used to specify a sequence of models from requirements to features, and from both of these to architecture (which includes modelling qualities) [87], [89]. The transformation process is based on respective meta-models. A meta-model is a model of model. Meta-modelling is the process of generating meta-models. It involves the development of rules and structures used for modelling a set domain problems. In other words, meta-models explicitly describe how domain-specific models are built.

2.5.2 MDE Technologies for Mobile

The MDE technologies for mobile are grouped in two headings – i) those that have been explored in Java mobile applications (J2ME), which are generally often applicable to all traditional java applications which run on JVMs, and ii) those that are applicable to mobile platforms with different VMs from JVM; such as Android running on Dalvik Virtual Machine (DVM). Based on up to date knowledge none of the explored MDE technologies have investigated the MCA domain towards mobile optimisation.

2.5.2.1 Technologies for Java Mobile

Carton et al. [90] investigates the use of AOP techniques and MDD for modelling context; targeted towards J2ME applications. The modelling in [90] is achieved through Eclipse Modelling Framework (EMF). EMF provides a Java/XML-based modelling and code generation framework. Transformation in EMF (i.e. the generator) is based on Java Emitter Templates (JET) to transform PSM to code. The templates are validated based on plain Java classes, moreover, the JET learning curve for the construction of templates can be quite steep, as the full structure of the class has to be constructed using a templating language. Thus, EMF is compatible with Java applications, and this includes J2ME applications – i.e. Java Mobile Edition group of applications, as shown in [90]. EMF, however, is not compatible with the popular current day mobile platform – i.e. Android. Although Android applications are developed in Java language they do not run on the traditional JVM (they run on DVM), moreover, new concepts such as Activity, Service and other Android specific APIs are introduced; some of which are due to the presence of varying mobile sensors [91], [92]. Consequently, EMF was not used in this research.

2.5.2.2 Technologies for Cross Mobile

As mentioned earlier, due to the uniqueness of the Android platform, EMF which was adopted for J2ME mobile applications in [90] could not be seamlessly applied in Android; consequently varying generators e.g. [93]–[96] are proposed to target transformation in regards to Android mobile applications. These are categorised based on how they perform modelling/transformation;

- UML-based Transformers

UML-based transformers are a class of transformers that achieve model transformation based on UML models. Usman et al. [93] propose a generator called MAG (Mobile App Generator) for the transformation of multiple mobile platforms – Android and Windows phone. MAG is based on UML modelling. MAG performs the transformation using a state pattern which takes as input; template classes and state machine with UML profile, in order to produce as output; specific mobile application classes for Windows or Android phone.

Parada et al. [94] extends the GenCode tool (which is naturally targeted towards Java code like EMFs) [97] to generate Android code based on class and sequence diagrams. Thus [94] like [93]; are based on UML modelling. The structure for the application (java files/classes) are generated from the class diagrams. The extended GenCode in [94] also allows for generation of Android API components such as Activity and Services (a limitation of EMF used in [90]). The behaviour of the application is generated from the sequence diagrams; such as method invocations and loops; however, some operations such as mathematical operations and variable assignment are not generated.

- Graph-based Transformers

Graph-based transformers are a class of transformers that achieves model transformation based on graph models. Lamhaddab et al. [95] uses MDE in graph models (MDEG) as a practical solution for reverse engineering across different mobile platforms (i.e. cross-platform development). MDEG represents complex systems as models (usually XMI: XML Metadata Interchange) in graph format. This provides more flexibility in transformation as shown by [95] – which demonstrates MDEG's flexibility in cross-platform transformation. For cross-platform transformation, a specific application in a platform is considered an instance of a meta-model. Instances of meta-models are annotated with tags or annotations for platform specific transformation.

- DSL-only Transformers

DSL-only transformers are a class of transformers that achieves model transformation only based on a custom domain specific language. They also target a similar aim of cross-platform mobile development through modelling, however, modelling is based on a language, not UMLs or graphs. Example of such tools/frameworks are applause [98], Automobile [99], AXIOM [96], [100], MD^2 [101] and Mobl [102]. These approaches are aimed at generating native code for different platforms through DSL, with the exception of Mobl which is based on the Web (i.e. transformed application is based on HTML5, CSS and Javascript) which is native feeling (though not native) application [102], [103].

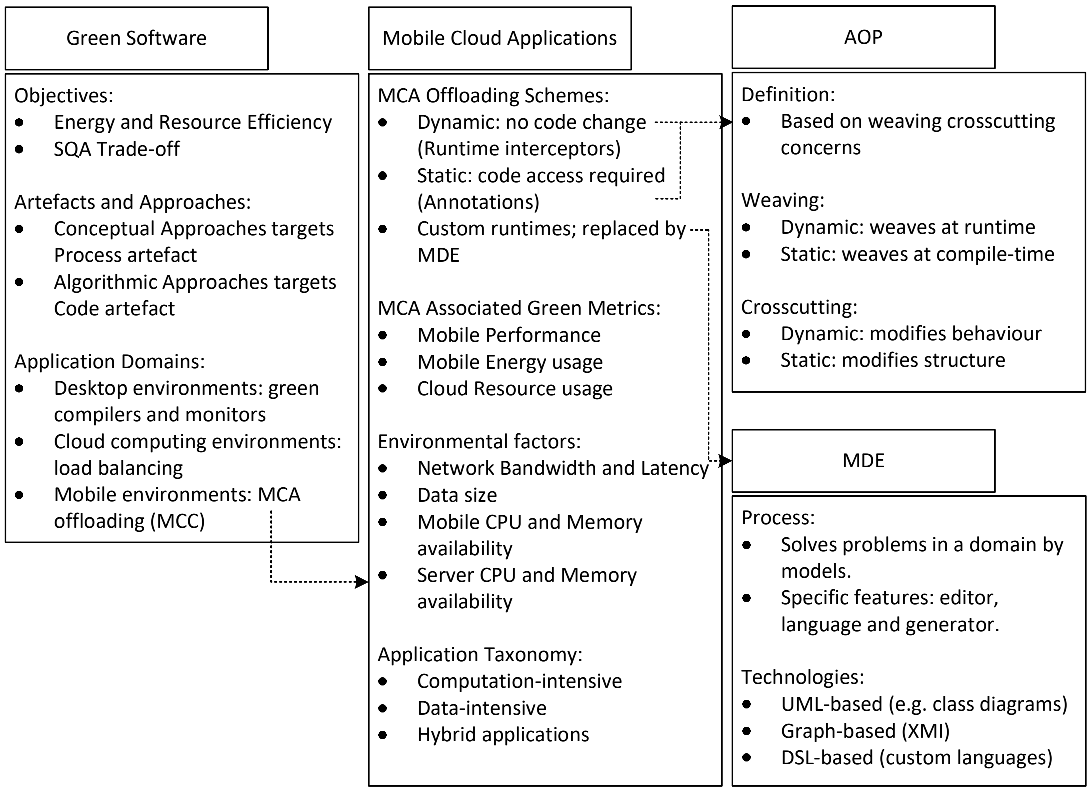

Green Software
Objectives:
Energy and Resource Efficiency
SQA Trade-off
Artefacts and Approaches:
Conceptual Approaches targets Process artefact
Algorithmic Approaches targets Code artefact
Application Domains:
Desktop environments: green compilers and monitors
Cloud computing environments: load balancing
Mobile environments: MCA offloading (MCC)
Mobile Cloud Applications
MCA Offloading Schemes:
Dynamic: no code change (Runtime interceptors)
Static: code access required (Annotations)
Custom runtimes; replaced by MDE
MCA Associated Green Metrics:
Mobile Performance
Mobile Energy usage
Cloud Resource usage
Environmental factors:
Network Bandwidth and Latency
Data size
Mobile CPU and Memory availability
Server CPU and Memory availability
Application Taxonomy:
Computation-intensive
Data-intensive
Hybrid applications
AOP
Definition:
Based on weaving crosscutting concerns
Weaving:
Dynamic: weaves at runtime
Static: weaves at compile-time
Crosscutting:
Dynamic: modifies behaviour
Static: modifies structure
MDE
Process:
Solves problems in a domain by models.
Specific features: editor, language and generator.
Technologies:
UML-based (e.g. class diagrams)
Graph-based (XMI)
DSL-based (custom languages)

Figure 2.7 Summary of Review and Scope of the Thesis

2.6 Summary

This chapter presented the background on green software engineering, mobile cloud applications, aspect-oriented programming and model-driven engineering as shown in Figure 2.7. Green software was introduced as applications that efficiently utilises resources during runtime. The consequence of which would be efficient resource usage and efficient energy usage – thus meeting an objective of green software. Also, a second identified objective or concern of green software is its integration with software quality attributes such as performance.

2.6.1 Green Software Engineering and MCA

Energy and resource efficiency have been considered a software quality attribute (since the advent of green software engineering), alongside other quality attributes like performance and security. In MCA, however, the metrics combination popularly explored are mobile performance and energy efficiency (performance representing a software quality, and energy-efficiency representing a green metric). Moreover, MCA is not a monolithically tiered domain as it is composed of the mobile and cloud tiers which are both faced with unique challenges – i.e. resource constrained challenge for the mobile domain, and the challenge of efficient provisioning anchored to the cloud domain. Consequently, in MCA, the aforementioned challenges have to be fully taken into consideration when investigating MCA metrics:

- To observe the efficiency of a MCA offloading scheme on resource-constrained mobile device – performance and energy usage is considered as metrics for the mobile tier.
- To observe the efficacy of a MCA offloading scheme for cloud provisioning – resource usage and software availability are considered as metrics for the cloud tier.

Thus, this thesis investigates full-tier qualities for MCA. Furthermore, to achieve and monitor the aforementioned qualities, this research explores the following:

- An architecture for efficient MCA offloading based on the full-tier qualities. The aim of the architecture is both for development-efficiency and high decoupling (Achieved through MDE and AOP techniques).
- An evaluation approach for MCA based on the full-tier qualities (the details on the motivation is presented in Chapter 3).

2.6.2 Opportunities for AOP in MCA

AOP has been explored in mobile computing for implementing crosscutting concerns or weaving application-ready (off-the-shelf) features into an application as in [104]. AOP however until now, has not been explored in the context of MCAs (in achieving green software for mobile applications).

AOP is popularly used in dynamic crosscutting, and the Logger example in Figure 2.5 is one of many applications of AOP concepts in software engineering. Success has been achieved in the use of AOP to address security checks, performance, transaction management etc. [81] as crosscutting concerns, some of which are concerned with software quality attributes such as security and performance. In a similar way with regards to MCAs; offloadable tasks are concerns associated with mobile energy efficiency. Thus AOP aspects could be employed in the implementation of offloadable tasks as cross cutting concerns – this can offer two key benefits to MCAs;

- Source decoupling

Aspects can help decouple MCA optimisation logic from the code base of legacy systems. This means that no major modification is required to be made on the source code of existing systems (especially the offloadable components) since pointcuts are used to identify join points of offloadable components.

- Reusability

Following from the source decoupling, only aspects will be subject to code refactoring or changes pertaining to offloadable tasks. This means that aspects can be reused for different MCAs as they are independent of the application code base. In other words, for any MCA an aspect component will remain the same structurally, with changes only made to the pointcut which refers to the offloadable task(s) of specific MCA. Furthermore, this *reusability support* allows for the development of tools to engineer/customise aspects for specific MCAs – in a way where aspects can be designed as templates given that it is structurally reusable. This can be accomplished by Model-driven Engineering.

Aspects are applied in the design pattern (later referred to as ACTS) which is used to realise the proposed architecture of this research.

2.6.3 Opportunities for MDE in MCA

Various technologies, e.g. [93]–[96] have been presented in the literature on MDE for mobile development; existing works, however, do not cater for software qualities and thus are not focused or applicable to MCA domain.

Thus a gap still exists for the application of MDE in MCA development for its automation so as to mitigate the need for custom runtimes and yet achieve an automated and flexible development process. Furthermore, as current contributions in the literature have demonstrated that the use of MDE can enhance abstraction and automation in generic mobile development [93]–[96], insights are consequently drawn to use MDE for MCA, in its automation of development (i.e. Development efficiency) as follows (in fulfilling the triad of PSM – discussed in section 2.5.1);

- By Editable MCA Meta-Models (*Editor*)

As presented in section 2.3.1, the identification of offloadable tasks is a sine qua non condition for MCA development. Thus a MCA meta-model can be used to incorporate offloadable tasks into a model. Furthermore, the quality objective of MCA in terms of optimisation aims (which are mobile performance and energy efficiency, cloud resource efficiency and availability) can be integrated into the model. For greater flexibility, a graph-based model can be used with MCAs, as graph-based models are based on XMI as shown by [95]. Consequently, an MDE editor for MCA is proposed by this research based on a graph model. As identification of offloadable tasks is a core activity in MCAs (see section 2.3.1) – this is also achieved in modelling in this study. The modelling framework comprises of two meta-models; the call-graph (which specifies the offloadable tasks and its properties) and the graph-based model (later referred to as Caller-Callee Model, which integrates the optimisation aims/attributes into the model).

- By Reusable MCA Templates (*Language*)

As presented in section 2.5.1, DSL is used to specify structure in a program through templates. Moreover, MDE models are driven with design patterns using tools; to increase productivity [9], [87]–[89]. MCA is a domain centred on offloadable tasks; a structure can thus be specified for MCA (for the purpose of MDE) by the use of a design pattern which encapsulates offloadable tasks and quality attributes as logic. This encapsulation can consequently be presented in the form of templates for reusability. This research proposes a design pattern – later referred to as ACTS (for the aforementioned Caller-Callee model) which is realised as templates. In order to accomplish reusability, the templates are implemented with placeholders and tags which are representative of the properties of the meta-model.

- By Automated MCA Code Generation (*Generator*)

The core MCA meta-model (i.e. the Caller-Callee model) and the MCA template (i.e. based on ACTS pattern) are used to generate application code for the MCA. As a MCA generator, the generator has to be aware of the optimisation objective in order to determine scenarios where the MCA will yield actual benefits. In this research, the generator is packaged as an API/framework, and the automated code generation which performs an evaluation (on the model) for asserting that MCA will yield actual benefits is later referred to as Quality Verifier. Thus the transformation is achieved in two phases; at the design (i.e. modelling) phase and at the verification phase.

The MDE architecture in the research is presented in later sections as Mango and the framework as Mosaic framework. And the MCA evaluation (different from model evaluation) is fulfilled by Beftigre framework (a test framework). The following chapter presents the methods adopted for the contributions of this research.

Chapter 3. Problem Statement and Methodology

3.1 Introduction

Having presented in the previous chapter, a background for the study based on a wide body of literature, this section focuses on formalising the main problem or gaps of the study. Consequently, the methodology adopted by this thesis to address the problems are presented. Also the chapter presents the objectives while introducing the contributions of the thesis.

The first set of gaps and methodology is associated with the approaches used for designing MCAs – i.e. offloading schemes. The second set is associated with the *evaluation* approach adopted in the research on MCAs.

3.2 MCA Optimisation Approach

3.2.1 Gaps in existing approaches

This section presents the gaps relating to MCA offloading models/techniques for improving the performance (in the context of execution time) and energy usage of mobile device applications. The existing challenges in the literature are presented, in this section, in terms of overheads in the components that make up the generic MCA architecture (illustrated in Figure 3.1).

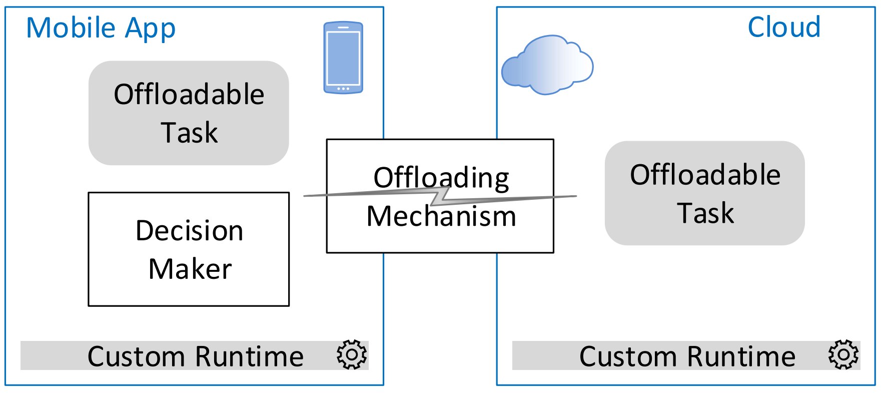


Figure 3.1 MCA architecture based on custom runtime (in the literature)

3.2.1.1 Challenges of Identification Technique

A task is identified for offload if it possesses chances of performance or energy improvement when executed remotely – i.e. its remote execution time is lesser than local. A key *constraint* impacting the performance gain of an offloadable task is dependence on mobile-only resources – such as sensor or camera, etc.

Zhang et al. [105] adopts a shortest path algorithm to identify an optimal cut which minimises offloading overhead. However, this does not take into consideration, the aforementioned *constraint* when identifying an offloadable task. Elicit [8] uses the shortest path approach for identifying offloadable tasks, and by taking into account the constraint, provides a better performance gain. In the literature offloadable tasks can either be identified manually using annotations or automatically through static/dynamic analysis as shown in Table 3.1. The latter is more development efficient and accurate [8], [59].

Furthermore, not all offloaded tasks prove to be performance or energy efficient, particularly the data- intensive applications, as shown in the literature [7], [8], [52]. Thus raising a question to the effectiveness of the approach used

Table 3.1 Comparison of offload models (derived from [4]–[7], [59])

System	**Identification Mechanism**		**Decision Maker**		**Offloading Mechanism**	
	Type of Transformation	*Level of Granularity*	*Type of Threshold*	*Used Parameters*	*Offloading Type*	*Automation System*
Kwon et al. [4]	Manual (Annotation)	Low	Static	Resource: data size	Cloning	Custom Runtime
MAUI [5]	Manual (Annotation)	Low	Dynamic	Resource: bandwidth and latency	Cloning	Custom Runtime
Hassan et al. [7], [8]	Automated	High	Dynamic	Resource: (full)	Partitioning	Partial Runtime
Native Offloader [59]	Automated	High	Dynamic	Resource: Mobile memory & bandwidth	Partitioning	Partial Runtime

N.B. This table is not an exhaustive list of the models, but a list of distinct representative models – consisting unique characteristics.

in the *identification* of offloadable task. Two notions can be deduced; either the task in concern was wrongly identified as offloadable or there was an overhead during run-time which was unaccounted for during the identification process. The second notion is likely a more valid point, because if the decision making and offloading components add an overhead during runtime, which was not considered during identification, then a particular offloading task may never yield performance or energy-efficiency benefits – in a scenario where the runtime overhead overshadows the offloading gain. Existing work [4], [7], [8], [52], [80] to the best of knowledge does not take into account all MCA components during the identification of offloadable tasks. It can be argued that the decision maker would effectively allow such scenarios to execute locally. The fact, however, is that the decision-making process is a required precondition; thus overhead would have already been made. Also, although the automated transformation schemes provide a higher level of granularity in identification of offloadable tasks (as shown in Table 3.1) it is still prone to overhead as it does not consider all the MCA components. Thus an effective identification technique must take into account all the MCA components of Figure 3.1.

- Problem I: The techniques for identification of offloadable tasks do not evaluate the overhead of the overall offloading model, and therefore are prone to overhead during runtime.

3.2.1.2 Challenges of Decision Maker

A good way to understand the decision-making component is, as a kind of monitor and comparator, rather than just a set of if else conditions.

As a monitor: conditions are executed by checking (i.e. monitoring) the *actual* environmental state. In MCA a given environmental state is defined by different factors/parameters which are; mobile device CPU and memory availability, network bandwidth and latency, cloud CPU and memory availability, and transmitted data size; as used in the literature [7], [8], [52]. Notice that the parameters employed by existing work are resource based (i.e.

CPU, memory, and network) as shown in Table 3.1. For accuracy in the decision-making process, it is critical that these factors be captured by the decision maker.

Some research such as [7] takes into account all the aforementioned factors for decision making, thus providing more decision accuracy with respect to the awareness of the environmental state. Most works, however, only consider a single or couple of the factors for decision making, for example data size alone [4], or a combination of bandwidth and latency [5]. Whichever combination of factors are used, an overhead is added to the application performance. For example monitoring for network bandwidth and latency, requires sending packets to and fro the communicating endpoints (e.g. [7], [8]); which contributes its own overhead. Thus the more factors that are considered the more overhead but the greater the accuracy in decision making.

As a Comparator: the actual environmental state obtained by measuring the aforementioned environmental factors are compared against a (set of) predetermined value(s). Hassan et al. [7] uses a machine learning algorithm – specifically multi-layer perceptron, as a comparator due to the use of multiple factors in the approach. Expected environmental factors are obtained as training data, collected for offload condition – i.e. when remote execution time is less than local, and non-offload condition. Kwon et al. [4] uses a single thresholding approach on data size for the offloading condition. This is, however, ineffective given the varying factors affecting MCA, and the unpredictable nature of the environment. Cuervo et al. [5] uses a linear regression model on bandwidth and latency – which also fails to compare other factors.

As mentioned earlier, the decision-maker component decides when to offload, with the offload condition satisfying a scenario where remote execution time is lesser than the local execution time. Consequently, the core factor is the elapsed execution time, however as this factor cannot be determined explicitly before runtime, the environmental factors with the support of learning models (or dynamic thresholds) [5], [7] or static thresholds [4] are adopted in the literature. Thus an effective decision-making process must effectively predict time with minimal overheads.

- Problem II: The accuracy of the decision maker is improved by monitoring a series of resources, and performing a threshold comparison dynamically (i.e. at runtime). However, dynamically monitoring many resources contributes runtime overhead, especially as this decision needs to be performed whenever an offloadable task is to be executed – whether it is finally offloaded or not.

In `Mango,` the elapsed *time* (from real-time prediction) as the decision parameter is used, so as to eliminate the overhead of resource monitoring for multiple resource parameters. Furthermore, the *time* parameter is dynamically set by Context (for decider) and Profiler Aspect (for threshold) as shown later in Chapter 3.

3.2.1.3 Challenges of Offloading Mechanism

Many offloading models implement their offloading mechanisms as runtime engines. There are however two key categories to the offloading mechanisms used in the literature, they are; cloning and partitioning. See Table 3.1.

Cloning as the name implies involves execution of a virtual device on the cloud. It is based on checkpointing – i.e. adding fault tolerance by saving snapshots, and thus creates more overhead in offloading due to state synchronisation (requiring as much as approximately 100MB data transfer [6]).

Partitioning makes use of remote procedure calls. Unlike the cloning approach which requires the virtual device running on the cloud, the partitioning approach only requires the offloadable component executing on the cloud; thus more efficient (saving energy and time) compared to cloning.

Offloading by partitioning has been adopted by both dynamic and static optimisation processes in the research. Dynamic optimisation [8] – i.e. optimising at bytecode level or runtime is most useful for optimising legacy/existing systems, making it difficult to utilise from an earlier design phase of the development process. Moreover, these approaches implement complex decision makers. Static optimisation models – optimising at the source level, can be adopted in the development process. However, their key challenge is the dependence on custom runtime engine, e.g. [4], [58]. As such deeply layered frameworks contribute to runtime bloat [2], [3], [106], an appropriate mechanism, would need to be simplified – preferably without dependencies on custom runtimes to minimise overheads. Thus; `Mango` adopts a model driven approach; eliminating the need for a custom runtime.

- Problem III: partition based approaches (which is considered more efficient of the two offload categories) are currently not applicable to the development process as they are built for runtime and solely focused on refactoring. This also makes it challenging to adopt existing models in real-world application development.

3.2.2 Methodology and Research hypotheses

3.2.2.1 Methodology to Solving the Challenge in Identification Technique

Following the challenges presented relating to the technique for identification of offloadable tasks; it is critical that the process for identifying offloadable tasks be composed of both the decision making and offloading components. In other words, a task is identified for offload if and only if the combined overhead of the decision-making component, offloading mechanism and remote execution is lesser than local. Thus the expectation is that;

> **H1:** *Offloading any task which compromises the aforementioned condition will always compromise performance, even if the remote execution time is less than that of local.*

Consequently, a tool called Mosaic (abbreviation for – model-based selective approach for identification of Callees) is proposed to effectively identify offloadable tasks during development, by taking into account the aforementioned constraint. The methodology behind Mosaic is the use of

MDE concept in identifying offloadable tasks. Offloadable tasks can be presented as meta-models (later discussed as Caller-Callee Model), and other MCA components (such as decision making and offloading components) are applied to the meta-model as templates (later discussed as Aspect-Context-Task-Service design pattern). Consequently, the model can be evaluated with all components taken into account.

3.2.2.2 Methodology to Solving the Challenge in Decision Maker

As mentioned earlier the core purpose of the decision maker is the ability to predict the elapsed time prior to offload, so as to know if there will be gain or loss given the current environmental state. This prediction is currently obtained from different factors, and using learning models (contributing overheads) or inaccurate thresholds. As a way to curtail the complexities and overhead of learning models and inaccuracies of thresholding based on environmental factors, the thesis proposes a time-based context-aware decision making approach. The two key concepts behind the approach to solving the decision maker problem is;

- Time-based concept: execution time is the only factor being monitored and thus eliminates the overhead of monitoring other MCA environmental factors.
- Context-aware concept: context-aware feature is used to ensure accuracy of the time threshold which compares against the execution time. Further details on the contexts employed for the decision making is presented in section 4.4.

As the proposed approach makes use of elapsed execution time as the factor for decision making (at the mobile and cloud tier), the expectation is that;

> **H2:** *By use of a time-based decision-making process at both mobile and cloud tier, software qualities can be achieved at both mobile and cloud tiers.*

3.2.2.3 Methodology to Solving the Challenge in Offloading Mechanism

To address the challenge of overhead caused by the custom runtime, this thesis proposes the use of sockets as the offloading mechanism and demonstrate that sockets finely integrate with the research objective of achieving software qualities (i.e. in support of the second hypothesis – H2). Furthermore, existing custom runtime engines help integrate/automate the implementation of decision making and offloading processes into the MCA. Similarly, the proposed Mosaic framework automate the development process, however, to eliminate the overhead of custom runtimes, Mosaic model-driven tool is designed to integrate with the mobile integrated development environment (IDE) as a library.

> **H3:** By use of an MDE tool integral to the IDE, the overhead of complex runtime can be mitigated and also MCA components can be generated on-demand during development.

3.3 MCA Evaluation Approach

3.3.1 Gaps in existing approach

In this section, examples are used from the research to highlight the problems of the currently used MCA evaluation approach (i.e. the architecture scenario approach) in the evaluation and comparison of offloading schemes. Thus, deriving the goals and novelty of the proposed solution – later presented in Chapter 6.

A Motivating Example:

Consider a situation in the development of mobile cloud application, the choice of an offloading scheme would be a critical decision, as it is the core functionality which transforms a mobile app to a MCA [4], [5], [7], [52]. Assuming the development team chooses to use an existing scheme, they will need to evaluate and compare between existing offloading schemes. Two offloading schemes have been selected, one based on single thresholding [4] – ST for brevity, and another based on multi-layer perceptron (MLP) [7] –

Table 3.2 MCA evaluation and comparison by architecture scenarios

Arch. Scenarios	**ST** [4]		**POMAC** [7]	
	Elapsed Time (ms)	**Used Energy (J)**	**Elapsed Time (ms)**	**Used Energy (mJ)**
Local	49331.55	86.59	3930.33	4854.24
Server	27673.79	63.86	34873.15	19839.42
Optimal	17486.63	44.73	3986.21	4845.10
The Scheme	10347.59	41.33	4242.32	5085.80
Local % diff.	130.65	70.76	-7.64	-4.66
Server % diff.	91.14	42.84	156.62	118.38
Optimal % diff.	51.30	7.90	-6.23	-4.85
Note: *Local % diff., Server % diff.* and *Optimal % diff.* is the % difference of the scheme in comparison to Local, Server, and Optimal scenarios respectively. A negative value is used to signify loss in energy or performance. Note that the metrics presented; i.e. elapsed time and used energy; are for the mobile tier.				

known as POMAC. Also selected is an optical character recognition (OCR) Android app; Mezzofanti – used in the source literature [4], [7], to validate the schemes. From the source literature [4], [7], the computation intensive offloadable component is the OCR functionality. The data presented in Table 3.2 is obtained from the source literature using WebPlotDigitizer [107].

To achieve the evaluation of individual schemes and comparison between the schemes (ST vs. POMAC), mobile-centric architecture scenarios provided by the source literature are used. Mobile-centric; meaning that the approach provides green metrics results for only mobile tier (i.e. performance and energy usage). Using mobile-centric architecture scenarios which are prevalent in the research [4], [5], [7], [52] however possess challenges which make it difficult to come to a satisfactory conclusion for both schemes, in terms of evaluation and comparison.

The problems identified for mobile-centric architecture scenario are grouped under three headings (the problem numbering continues from previous section as they are all relating to MCA):

- Problem IV: Variability of architecture scenarios.
- Problem V: Inconsistency in evaluation results of scenarios for an offloading scheme.
- Problem VI: Coarse-granularity of evaluation.

3.3.1.1 Variability of architecture scenarios (making it difficult to compare between offloading schemes)

The literature use varying scenarios to evaluate proposed schemes, e.g. [7] defines four[7] scenarios to evaluate POMAC, while [4] defines five[8] scenarios to evaluate ST. Therefore, to establish a basis for comparison, scenarios will have to be matched (as presented earlier in section 2.3.5.2). This process introduces complexity in comparing schemes especially since scenarios which may be congruent (by inference) may have slightly different definitions from each other (based on the actual literature implementation). This introduces difficulty in communicating varying scenarios between the development teams, and also a challenge to the comparison.

3.3.1.2 Inconsistency in evaluation results of scenarios for an offloading scheme

To evaluate POMAC, [7] defines four scenarios. The efficiency of POMAC is evaluated by comparing the POMAC scheme against other defined architecture scenarios, using % difference. Deducing from Table 3.2, for energy usage it can be concluded that POMAC is approximately 5% inefficient compared to both local and optimal scenarios and 118% efficient compared to server scenario. Although the local and optimal % differences seem to arrive at the same conclusions, there is no clear relationship between the scenarios. This is shown by ST which has approximately 71%, 43% and 8% energy improvement based on local, server and optimal respectively. This challenge makes it difficult to weight a scheme based on easily verifiable values or conclusions.

[7] Four scenarios are defined by [7] for evaluating POMAC, they are OnDevice, OnServer, Optimal and POMAC.

[8] Five scenarios are defined by [4] for evaluating ST, they are Smartphone only, Offloading w/All objects, Offloading w/Necessary objects, Offloading w/Necessary objects (delta), and Offloading w/Threshold check.

3.3.1.3 Coarse-granularity of evaluation

Different literatures use different levels of experimental rigour. For example; [7] performed a more rigorous experiment for POMAC evaluation (as the scheme is based on MLP), compared to [4]'s experiment for ST which is not as rigorous. Comparing ST energy with POMAC (using optimal scenario as reference) gives approximately 8% gain in ST and 5% loss in POMAC. The case may be that in adverse environmental conditions ST scheme fails to save mobile energy (which is true from later experiment conducted – in Chapter 7). Also since the analysis is mobile-centric, it fails to provide the overall implications of a scheme's decision to or not to offload. This challenge poses difficulty when deciding schemes with overall efficiency (i.e. mobile as well as cloud resource aware). The Case Studies (in Section 7.6) later shows that with full-tier evaluation one can better understand if a scheme just keeps offloading to server, or if it checks server availability (i.e. robustness).

3.3.2 Methodology for a solution

To propose a solution for the identified gaps in the existing MCA evaluation approach (i.e. the mobile-centric architecture scenarios approach), this thesis adopts concepts from Behaviour Driven Development and Fine-Grained Testing.

3.3.2.1 Behaviour Driven Development (BDD)

A key difficulty in evaluation of MCA identified above is the variability of architecture scenarios – which also makes it difficult to compare between offloading techniques. Which shows that, since there is no standard as to the scenarios to use for justifying efficiency of an offloading technique, different literatures/techniques use different scenarios. As a solution to the aforementioned difficulty in varying scenarios, this thesis presents an approach for MCA evaluation and comparison based on the factors surrounding typical MCA scenarios.

Table 3.3 Simplified MCA Evaluation by Use of Environmental Factors

Evaluation by Architecture Scenarios Approach			**Evaluation by environmental factors**	
Scenarios	**S1 Time (ms)**	**S1 Energy (J)**	**Elapsed Time (ms)**	**Used Energy (mJ)**
A % diff.	x_1	y_1	xx	yy
B % diff.	x_2	y_2		
C % diff.	x_3	y_3		
Assuming *A, B and C to be representative of scenarios, such as Local, Server and Optimal.*				

For example; for a typical scenario, whether server, optimal or scheme, the factors surrounding the efficiency of the application are mobile CPU and memory availability, server CPU and memory availability, network bandwidth and latency [7], [8]. Rather than evaluate schemes by comparing against different scenarios which are all affected by the aforementioned factors, this research proposes evaluating and comparing schemes on the bases of the factors themselves which affect the schemes.

Furthermore, the implication of the proposed idea of the research is that to evaluate an offloading scheme S1, a result can be presented thus:

- the performance and energy usage of S1 is x and y respectively, given the aforementioned factors. This is a more simplified and easy to interpret approach as shown in Table 3.3.

Rather than:

- the performance and energy usage of S1 is x_1 and y_1 respectively, compared to a scenario (A, which however is affected by its own uncontrolled factors), and x_2 and y_2 compared to another scenario (B, which is also affected by its own uncontrolled factors), and x_3 and y_3 compared to another scenario (C, which is also affected by its own unique factors). Using varying scenarios for evaluation introduces unnecessary complexities as shown in Table 3.3.

And then to compare a second offloading scheme of interest say S2 to the previous, S1, the process would be performed as follows:

- given that the factors of S1 and S2 are closely related compare S1 to S2.
  Assuming that S1 is more efficient, then the result can be presented as thus;
- S1 is x% and y% more performance and energy efficient than S2 given the factors.

Rather than:

- Compare S1 to A and S2 to A; then S1 to B and S2 to B; then S1 to C and S2 to C.
  Assuming that S1 is more efficient, then the result can be presented as thus;
- S1 is x% and y% more performance and energy efficient than S2 in A, *and/or*
  S1 is x% and y% more performance and energy efficient than S2 in B, *and/or*
  S1 is x% and y% more performance and energy efficient than S2 in C.

'and/or' meaning that in most cases S1 might not be more efficient in all the compared scenarios, thus it is difficult to establish a concrete result for comparison using varying scenarios.

Notice that for the proposed approach; the syntax is '**given** *factors* **then** *assert results*'. The above syntax is the core of behaviour driven development (BDD). Thus, the behaviour-driven technique is used to address the first two identified challenges (i.e. the problem of variability of scenarios and inconsistency of results). Consequently, the thesis proposes Beftigre (explored in Chapter 6) which adopts the BDD concept and simple clause approach, to simplify the comparison and evaluation of offloading schemes, and thus simplifying software design decisions.

Behaviour-driven development (BDD) is a design approach to aid collaboration between non-technical contributors (such as business analysts, or users) and software engineers. Consequently, BDD gears towards more verifiable and collaborative test process by being able to compare expected behaviours with actual results, following standard simplified scenarios – constructed by simple language clauses, GIVEN, WHEN and THEN [108].

3.3.2.2 Full-tier as the new Fine-grained testing for MCA

Also presented as a key challenge to current multi-scenario approach to evaluation of MCA, is the mobile-centric nature of the evaluation process. Thus, only the impact of an offloading scheme on the mobile device is estimated. However, MCA is composed of mobile and cloud tiers. Therefore, to address the coarse-granularity of current approach, an effective solution must take into consideration the mobile as well as the cloud resource impact of an offloading scheme. [34], shows that a fine-grained approach to energy measurement (using counters) can reveal specific energy usage in relation to specific points of execution. Similarly, to identify specific implications of an offloading scheme on an MCA, this research proposes *Beftigre* (presented in details in Chapter 6) which adopts fine-grained measuring across the mobile tier (using Markers, to measure energy usage and performance) and cloud tier (using Metrics Collector, to measure CPU and memory usage). The approach adopts concept of fine-grained software testing to present the implications of an offloading scheme on the mobile tier as well as on the cloud tier. By evaluating the system as a whole the Beftigre approach can detect whether an offloading scheme is aware of both mobile and cloud resource consumption. The full-tier objective of the approach is also assisted by the BDD concept.

3.4 Summary

This chapter presented the current state of the art in mobile cloud applications development. And consequently highlighted the key issues with the domain – in terms of *optimisation approach* and *evaluation approach*.

The chapter presented six challenges (as enlisted below) faced by the research on MCA which are addressed in this thesis. The first three problems are associated with existing optimisation approaches (summarised in Table 3.1), and the latter three are associated with the evaluation approach used for MCAs.

I. Inability to evaluate the overhead of overall offloading model, which in turn results in performance overhead.
II. Multiple parameter based decision-making (with intension of accuracy in environmental prediction) which leads to runtime/performance overhead.
III. The optimisation algorithms are highly dependent on runtime, and thus difficult to apply to development process.
IV. Inconsistency in evaluation results of scenarios for an offloading scheme.
V. Variability of architecture scenarios (making it difficult to compare between offloading schemes).
VI. Coarse-granularity of evaluation – focused on mobile implications of an MCA or its offloading scheme.

To address the identified gaps in existing optimisation approaches this research proposed a model driven architecture – Mango, which;

- Ensure that identified offloadable tasks will most certainly yield benefits, during optimisation, prior to final deployment.
- Is based on execution times as opposed to multiple environmental factors as parameters. And employs the use of sockets to implement optimum execution. Thus the approach does not seek (or monitor) best path of execution (requiring extensive resource monitoring, thus

causing overhead), but adopts a good-fit path with respect to execution time. That is to say that; decision to offload is made based on time, and control of remote execution is achieved based on time (as threshold) – this is further explained in Chapters 4 and 5.

- Is based on model-driven engineering. And thus mitigates the overhead caused by custom runtimes. The proposed Mango approach is composed of a model and design pattern, which supports the architecture for development as well as legacy optimisation. Consequently, as an MDE approach, an MDE framework, Mosaic, is also proposed to expedite development/optimisation.

To address the identified gaps in the scenario-based evaluation approach the research proposed a behaviour-driven full-tier approach – Beftigre, which;

- adopts the BDD concept and simple clause approach, to simplify the comparison and evaluation of offloading schemes and, thus, simplify software design decisions. This is based on the use of the actual environmental factors as parameters for evaluation rather than varying scenarios. Notice that; the environmental factors are applied now in the evaluation process (for finer granularity) rather than into the optimisation process (which can cause performance overhead every time offload decisions are made);
- adopts the concept of fine-grained software testing to present the implications of an offloading scheme on the mobile tier as well as on the cloud tier.

Conclusively, the research investigates solutions for i) an optimisation technique for MCA and ii) evaluation technique for MCAs. The aforementioned solutions (both architecture and evaluation) are presented in the following chapter – in a unified approach (called Mango).

Chapter 4. Mango Architectural Approach

A Model-driven Context-aware Architecture for MCA

4.1 Introduction

This chapter presents Mango: model-driven approach for integration of software quality with green optimisation in MCAs. Mango is the core contribution of this research. The major aim of the approach is to provide an architecture[9] which seamlessly integrates software *quality attributes* (SQAs) with the green optimisation objective of MCC, at both the mobile and cloud tiers (i.e. full-tier). Also, as MCA is an application domain which spans through mobile and cloud tier; Mango architecture therefore takes into account the specification of SQAs across the mobile and cloud tiers. Most importantly, in the architecture, these attributes are integrated into the system as a way to improve efficiency by applying them in a contextual manner – thus, context-aware. Mango is also a model-driven architecture, thus resource intensive tasks and their SQAs necessary for optimisation, can be modelled and transformed into code base by MDE tool. As shown in Figure 4.1, two frameworks are derived from Mango approach, they are Mosaic and Beftigre. Chapter 5, presents the MDE approach and framework, called Mosaic, useful for the modelling and transformation of Mango architecture.

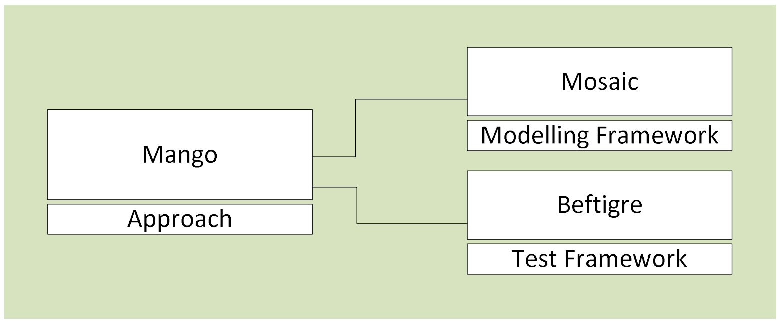
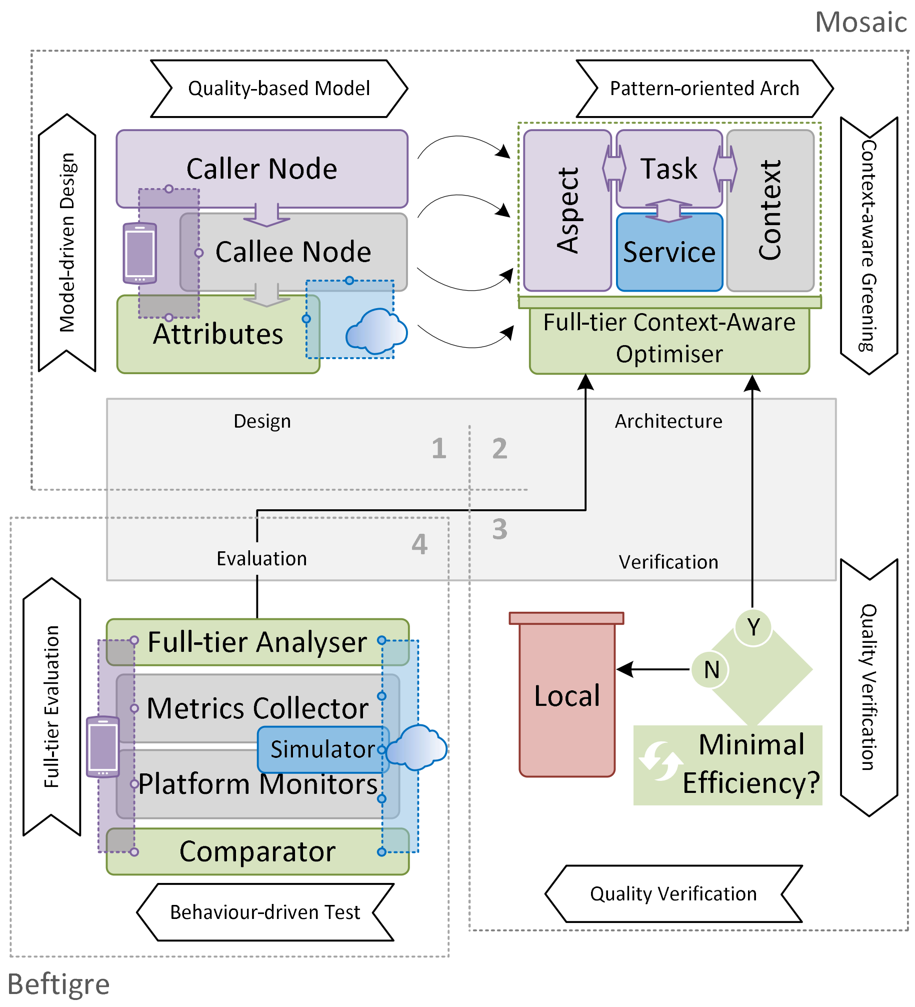


Figure 4.1 Mango and Derived Frameworks

[9] In the thesis, both the overall approach and the architecture are referred to as Mango. The first is called Mango approach which yields the latter, called Mango architecture.

As mentioned earlier, the Mango approach targets quality attributes spanning through the mobile and cloud tiers. As much as achieving the full-tier quality objective within the architecture, the test/evaluation process also has to take into consideration the full-tier software quality objective of the architecture.



Key:

🟪 **Mobile components**: are associated with the mobile tier.

🟦 **Cloud components**: are associated with the cloud tier.

⬜ **Shared components**: are associated with both the mobile and cloud tiers.

🟩 **Green Logic**: accomplishes green optimisation and is associated with both tiers.

Architecture: denotes the local/mobile-only architecture (red), and the Mango architecture (green).

Activities: denotes novel activities/contributions in the phases of Mango approach.

[Arrows] **Flow**: denotes the flow of information, and/or connection between components.

Figure 4.2 Mango Approach

Chapter 6 details the full-tier evaluation approach and framework for mobile cloud applications, called Beftigre. Although the full-tier evaluation concept of Beftigre is derived from the Mango approach, it is also suitable for testing existing offloading schemes due to the full-tier nature of MCAs (i.e. involving mobile and cloud tiers) – and consequently addressing the mobile-centric challenge of current MCA evaluation approach.

4.2 Overview of the Approach

4.2.1 Concepts and Components

The Mango approach (Figure 4.2) splits the development of MCA into four phases: design, architecture, verification and evaluation. The design phase introduces a model-driven design to the development process. It is composed of the Caller-Callee model. The architecture phase introduces the concept of context-aware optimisation. The Mango architecture is pattern oriented and introduces the use of Aspect Context Task Service (ACTS) pattern in the development. The verification phase is used for quality verification – to verify the suitability of the architecture for the application being developed. The design, architecture and verification phase are realised by Mosaic framework (see Chapter 5). The evaluation phase introduces full-tier and behaviour-driven concepts for MCA evaluation. Full-tier evaluation makes it possible to evaluate the MCA at a finer granularity which takes into consideration metrics from both mobile and cloud tiers. Behaviour-driven evaluation makes it possible to provide a consistent and reliable comparison between other approaches or counterpart techniques.

As shown in Figure 4.2, Mango Approach groups the design components and architecture components into two key features, which are the Caller-Callee Model and the ACTS design Pattern respectively. Model-Driven Engineering (MDE) focuses on exploiting domain models to effectively solve a recurring problem. Consequently, MDE simplifies the development process using models of design patterns (alongside tools) to increase productivity [88]. With the aim of proposing a simplified and effective solution, Mango (Figure 4.2) adopts MDE for effectively representing offloadable components with software qualities as a model – called Caller-Callee model. ACTS design pattern is proposed to describe the functional aspects of the architecture. Caller-Callee model is a model representation of the ACTS design pattern, however at the design phase, as shown in Figure 4.2. In this section qualities and quality attributes are used interchangeably to both refer to Software Quality Attributes (SQA).

4.2.2 Benefits of the Approach

The Mango approach (Figure 4.2) aims at achieving efficiency by treating the identified overheads/challenges in MCA as a software engineering problem; rather than merely an optimisation problem. The intention is that by addressing the earlier discussed concerns (presented in Chapter 3) as a software engineering problem, the need for complex optimisation or refactoring processes (fulfilled by custom runtimes) would be mitigated; thus resulting in a fine-grained, simplified, yet effective solution.

Furthermore, the core benefits of Mango approach have been presented in the following listing. These benefits highlight the novel contributions of the approach, and are evaluated in the case studies section. They are as follows (the last two points are specific to the evaluation phase);

- Full-tier efficiency.
- Variability awareness.
- Suboptimal awareness.
- Development efficiency.
- Full-tier evaluation.
- Robustness of test.
- Reproducibility of test.

4.2.2.1 Full-tier Efficiency

Full-tier efficiency is achieved at all phases of the development. At the design phase, full-tier efficiency refers to modelling of qualities for mobile and cloud tier. At the architecture phase, full-tier efficiency refers to the context aware decision making based on the mobile and cloud tier qualities. At the verification phase full-tier efficiency refers to the ability to verify/capture the overhead of overall offloading model for energy/resource usage or performance savings at mobile and cloud tier, thus addressing problem I of Chapter 3. The full-tier efficiency in the aforementioned phases are captured in the Caller-Callee model and ACTS pattern. Details of how the Caller-Callee model provides full-tier efficiency as a solution to problem I (in chapter 3) is presented in section 4.3.3.

4.2.2.2 Variability Awareness

Variability awareness refers to the capability of the Mango architecture to adapt (or make decisions) in varying environmental conditions with minimal overhead, whether normal or adverse conditions, in order to achieve software target qualities. Variability awareness is achieved through *context-aware optimisation* logic – which is the logic of the Context component of the ACTS design pattern.

The context-aware optimisation is based on the use of execution time as the single core parameter for decision making, consequently mitigating the overhead from measuring multiple environmental parameters. Thus, the variability awareness benefit addresses the existing decision making challenge (problem II of Chapter 3).

4.2.2.3 Suboptimal Awareness

Suboptimal awareness refers to the capability of the Mango approach in avoiding situations where offloading does not yield benefits. Suboptimal awareness is achieved by quality verification of Mango approach (i.e. the Quality verifier and Selective Analyser in Mosaic framework).

This benefit is made possible due to the fact that MCA components can be flexibly integrated into a meta-model (which is engineered to code). The behaviour of the model can then be used to determine if an offloadable task will justifiably yield benefits. Since the implementation code is based on a meta-model it can be forward engineered or reversed (depending on whether or not the offloadable task passed the verification). Thus Quality verification addresses the problem inability to evaluate the overhead of overall offloading model (problem I of Chapter 3).

4.2.2.4 Development Efficiency.

Development efficiency refers to the ability of the Mango approach to seamlessly and effectively achieve the MCA transformation process during development with no custom runtime intervention. This is achieved through the model-driven approach in Mango with meta-modelling and transformations – further detailed in the Mosaic framework (Chapter 5).

By adopting a model-driven approach, Mango addresses problem III of Chapter 3 – which refers to the difficulty of the adoption of optimisation algorithms due to tight-coupling to custom runtimes. Furthermore, as a model-driven approach, Mango focuses conceptualisation of implementation logic on the meta-models and templates, thus highly fostering reuse (at platform independent level – i.e. models and platform level – i.e. templates).

4.2.2.5 Full-tier evaluation

Full-tier Analyser (at the evaluation phase) addresses the coarse granularity problem of the existing evaluation approach (problem VI of Chapter 3) by providing an evaluation mechanism which takes into account the mobile and cloud tier, and their respective qualities (full-tier qualities) presented in the design phase. Thus, Full-tier Analyser (in evaluation) directly maps to full-tier optimisation (in architecture), as shown in Figure 4.2. The full-tier efficiency in the evaluation phase is captured within the Full-tier Analyser of Beftigre framework.

4.2.2.6 Robustness of Test

Evaluation in Mango approach is highly robust to evaluate the efficiency of MCA, as well as capture differences between two compared MCA schemes or approaches. The proposed Beftigre approach is robust to produce full-tier results as well as capture test scenarios based on environmental parameters which can reused as tests. The behaviour-driven features (through use of annotated clauses) of Beftigre evaluation approach makes it robust in capturing test results effectively, thus addressing the problems (IV-V of Chapter 3) associated with current evaluation approach.

4.2.2.7 Reproducibility of Test

Evaluation in Mango approach also makes it possible to produce test results which are consistent/reproducible (in other words, arriving at a conclusion which is unbiased by the testing environments, or environmental factors). This is achieved by control measures proposed by the evaluation approach (called Beftigre – further discussed in Chapter 6). Inconsistency and variability problems of the existing evaluation approach (i.e. problem IV and V of Chapter 3) are addressed by the reproducibility benefit.

Other benefits of the architecture are presented in the summary section. The rest of the chapter presents the approach in details based on the four phases of the approach (illustrated in Figure 4.2).

4.3 Designing the Model

Context-driven requirements analysis for Caller-Callee model (CRAC) presented in Figure 4.3, is not an alternative requirements analysis approach. CRAC can be viewed as a complimentary requirements analysis approach. This is because its main purpose is to identify offloadable components in an MCA, in order to generate the Caller-Callee model – which applies SQAs to the identified offloadable components.

CRAC involves three phases; requirements listing, component classification and caller-callee modelling.

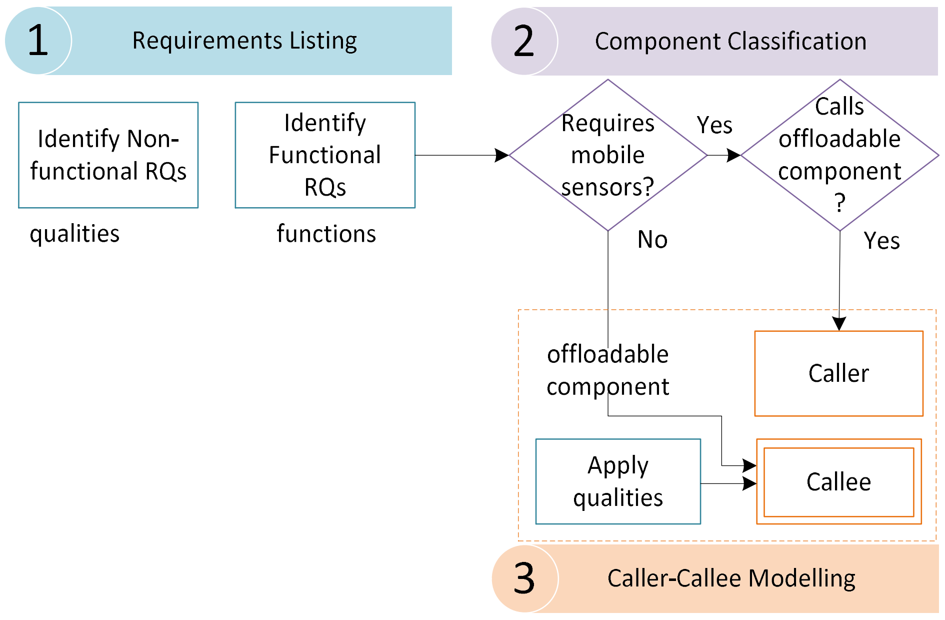


Figure 4.3 CRAC Process

4.3.1 Phase 1: Requirement Listing

This phase involves the presentation of both functional and non-functional requirements. The non-functional requirements are the SQAs which the system is to adhere to. In Mango approach four SQAs are considered for MCA optimisation as shown in Table 4.1 – these are mobile performance, mobile energy-efficiency, cloud resource-efficiency and software availability. The functional requirements are the features of the application, defined by their functional purpose as shown in Table 4.2.

Example: In a face detection application comprising of three requirements; face capture, face detection and face tagging. The requirement listing is presented below:

Table 4.1 Non-functional (N) Requirement Listing

Requirements Listing		**Tiers**
N1	Performance	Mobile
N2	Energy	Mobile
N3	Resource	Cloud
N4	Availability	Cloud and Mobile

Table 4.2 Functional (F) Requirement Listing

Requirements Listing		Functions
F1	Face capture	Capture an image
F2	Face detection	Detect face from the capture
F3	Face tagging	Add a description to the detected face

4.3.2 Phase 2: Component Classification

As shown in Figure 4.3, phase 2 is an extension from (or based on) the functional requirements activity of phase 1. Phase 2 involves the identification of the Caller (i.e. the sensor-centric component that references the Callee) and the Callee (offloadable component) from the functional requirements listing.

Identifying the Callee. The Callee (which is the offloadable component), is identified based on the assumption that any component that does not rely on a mobile-constrained[10] resource is an offloadable component.

Identifying the Caller. After a Callee is identified then the Caller is identified as the requirement which directly references (or makes calls to) the Callee requirement.

At the requirements phase, these components are identified qualitatively by abstractly deducing from the functional description provided for the requirement. (The component classification are refined quantitatively by static and dynamic analysis – using the Mosaic framework, presented in Chapter 5).

Table 4.3 Component Classification

Requirements Listing		Classification
F1	Face capture	Caller of F2
F2	Face detection	Callee
F3	Face tagging	-

[10] Mobile-constrained resources are resources that are constrained to mobile devices, and consequently cannot be provision on the cloud. E.g. client-only APIs and resources such as GPS, camera, microphone, and other sensors.

Example: based on the face detection application, Face capture can be speculated as sensor-centric as it would involve using the camera to capture an image. Assuming the description added to faces are obtained from the user's mobile contact, then face tagging would be sensor-centric as well, since it depends on the client-APIs – i.e. for retrieving mobile contacts. Face detection requirement however can be speculated as the offloadable component (Callee) since it runs an algorithm on the captured image, for detection. Thus face capture is the Caller of face detection, as shown in Table 4.3.

As presented later in Mosaic (Chapter 5), the component classification phase is inherently fulfilled by static and dynamic analysis of application source code. The output of which is a call-graph artefact, useful for Caller-Callee modelling.

4.3.3 Phase 3: Caller-Callee Modelling

This phase involves the integration of the non-functional requirements (i.e. the SQAs from phase 1) with identified offloadable components (from phase 2) at the mobile and cloud tier of the MCC system. The model fulfils the SQA integration objective of Mango architecture. Therefore, designing a Caller-Callee model (Figure 4.4) involves specifying nodes and qualities for mobile and cloud tiers.

A problem identified with existing MCA approaches is that of the inability to evaluate the overhead of the overall offloading model, consequently resulting in performance overhead (problem I in chapter 3). The Caller-Callee model

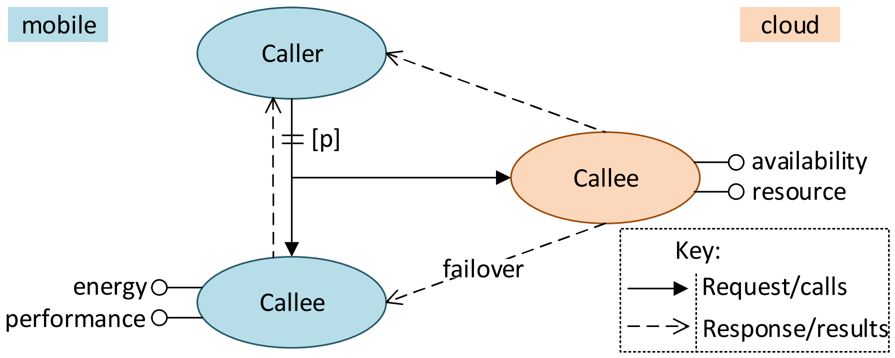


Figure 4.4 Caller-Callee Model

provides full-tier efficiency as a solution to this problem. This is achieved by encapsulating all the features of the MCA transformation in the meta-model, which involves both the mobile and cloud tiers. The model consists of both the nodes and the logic for achieving set qualities, thus encapsulating the overall offloading logic at the early stage of development – design phase.

4.3.3.1 Specifying nodes

The Caller-Callee model (Figure 4.4) is specified using two node types; the Callee node (being the offloadable component) and Caller node (the component referencing or calling the Callee). Furthermore, in specifying the nodes, two representational Callee nodes are modelled – one for the mobile tier and one for the cloud tier. This is to model offloading scenarios of the MCA which can be executed at mobile or cloud tier.

4.3.3.2 Specifying qualities

The quality specification in the Caller-Callee model involves two activities;

1) *Specification of attributes in tiers*. At the Callee nodes, all qualities to be implemented for mobile and cloud tiers are assigned. For example, as shown in Figure 4.4, performance and energy efficiency attributes are for the mobile tier, while availability and resource-efficiency attributes are for the cloud tier. In the Caller-Callee model, qualities are assigned to Callee(s) of tiers in form of tags, as shown in Figure 4.4.
2) *Specification of priority attribute*. Priority attribute is the mobile tier attribute used to execute the application at runtime. This is the *static context* for the application (see *Context* in *ACTS Pattern* section). Priority attribute is specifiable by the user, in other words, it is based on user preference. Within the Caller-Callee model, priority attribute is represented on the call or request[11] link to the Callee, which signifies that priority attributes have to be available before the call to the Callee

[11] Request refers to the reference made to the remote Callee (i.e. the cloud Callee), while calls refers to the reference made to the local/mobile Callee. Similarly response and results are feedbacks received from Callee at the cloud and mobile tier, respectively.

in order for optimisation decision to be made. From Figure 4.4, the symbol "[p]" denote that performance is prioritised over energy-efficiency. Priority specification is particularly useful for the mobile tier, as it implements qualities which can be traded-off between each other. For example, performance can be traded off for energy.

The response/result links connect from Callee to Caller. This signifies that the Caller continues execution after response from the Callee. The response link from remote Callee to local Callee is used to signify failover feature – in case of any remote errors or network disconnection, execution is passed back to local Callee rather than the Caller to avoid wrong results (i.e. for data integrity) or prevent application crash.

Decoupling priority attributes from generic attribute specification provides flexibility in application usage, so that users can define the context of execution for applications, as opposed to an undisclosed context; popularly used in the research – which assumes energy saving need for any execution scenario. For example, a benefit is that; with the decoupling a user can choose performance as priority in a scenario of high battery availability, and energy-efficiency with an intention to save battery usage.

As presented later in Mosaic (Chapter 5), the Caller-Callee model is defined as a generated ACTS source template implementing the Mango architecture with specified SQAs – based on the call-graph artefact of phase 2.

4.4 A Pattern Oriented Architecture for Context-aware Optimisation

ACTS (which stands for Aspect, Context, Task and Service) is an architectural pattern which Mango architecture uses to implement the Caller-Callee model. The purpose of the pattern is to provide an implementation skeleton for the Mango architecture which spans through the mobile tier and cloud tier. ACTS is composed of four components; Aspect, Context, Task and Service. The

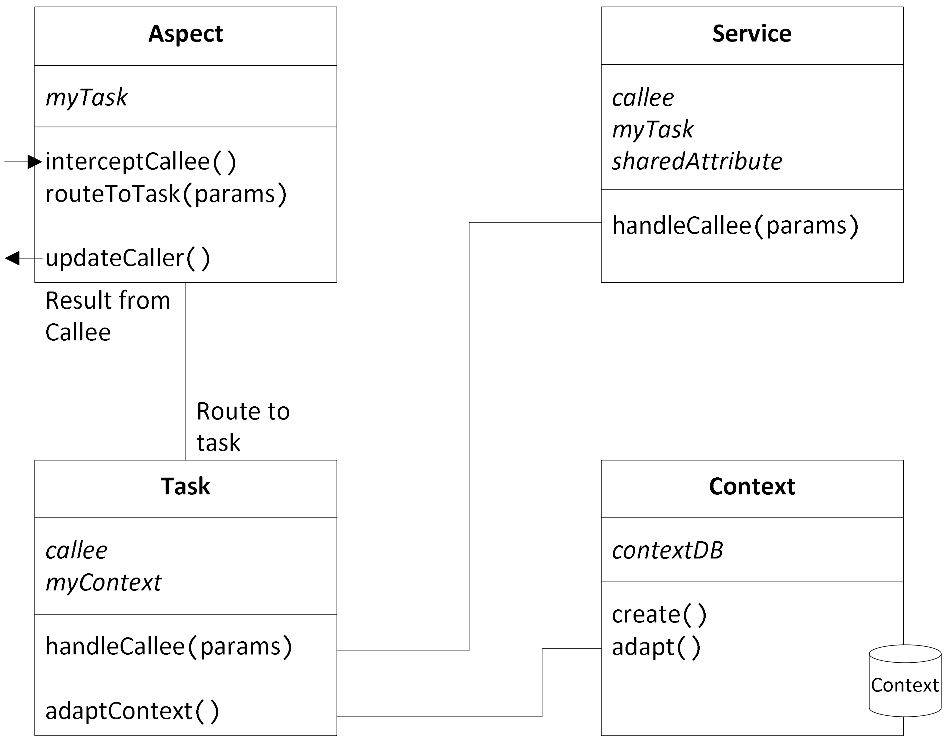


Figure 4.5 ACTS Pattern (Class Diagram)

Class diagram of Figure 4.5 illustrates the functions of the components described below in details.

4.4.1 Aspect: *Dynamic Crosscutting Component*

The Aspect component is an AOP Aspect used to intercept calls made by the Caller to the Callee, and then routes execution to the task component (to either offload or execute locally) using parameters required for the Callee execution. After the execution, the Aspect returns results to the Caller to continue execution. The three aforementioned functions fulfilled by Aspect is illustrated in Figure 4.5 by interceptCaller, routeToTask(params) and updateCaller, respectively. The ACTS component which Aspect directly collaborates with is the Task – i.e. by routing to it. The interceptCaller and updateCaller are fulfilled by Aspect *around* pointcut. To support use of Aspect on the mobile tier, AspectJ Android plugin[12] is used.

[12] *AspectJ Android Plugin:* https://github.com/uPhyca/gradle-android-aspectj-plugin

Why AOP Aspect? In Aspect Oriented Programming (AOP), Aspects are used to implement crosscutting concerns – such as logging, transaction, and security[81]. The purpose of using Aspect in ACTS pattern is to implement offloadable component (i.e. Callee) as crosscutting concern – so that an optimisation applied to an identified Callee will apply to all aspects of the program where that Callee is used. Also importantly, the use of AOP makes `Mango` architecture useful for legacy systems, as Callees are not required to be manually identified or modified. Thus, eliminating the need for multiple implementation and reducing development time. For example; Kwon et al. [4] proposes use of annotation to annotate the identified Callee methods – which implies that a developer needs to keep track of different occurrences of an offloadable tasks; this is impractical and development inefficient.

4.4.2 Context: *Representation of User and Environmental Contexts*

Context, a key principle in mobile pervasive systems [64] refers to the circumstances that form the settings for an event, or simply the elements of user's environment that are relevant for the application [64], [109]. This is similar to the MCA environmental factors mentioned in the review (see section 2.3.2), however in Mango the term 'Context' is used for better insight. Contexts used in Mango are grouped into two categories; Static and Dynamic Context.

The static context is the specified quality attribute considered as the priority attribute for executing the application at any given time. For example, given that energy and performance form the possible static context for the mobile tier, specifying energy at any given time will execute the Callee for mobile energy-efficiency. Static contexts are however reliant on the dynamic context for decision making.

The dynamic context is the basis on which the decision making of the MCA is made. It is the core logic for the decision maker, and is analogous to the 'use of environmental factors' approach adopted by the related work. It is composed of elapsed time (of mobile and cloud execution) and execution mode (i.e. for mobile or cloud execution) generated at runtime. The Context component of ACTS is used to specify logic for persisting, retrieving and adapting the dynamic context (as shown in Figure 4.5). The purpose of storing the dynamic context is so that they can be used in decision making of subsequent executions within the Task component.

In a nutshell, the static context is the priority attribute specifiable by the user, while dynamic contexts are elapsed times (from mobile and cloud tiers) and execution mode obtained during runtime – further explained in Algorithm 4.1. Within an Android application, Contexts are stored in Shared Preferences as

Algorithm 4.1 Adaptive Context-aware Decision Maker

```
Require: mode, mt and ct in Context DB.
 //By default mode in Context DB is set to mobile

 1: overhead = 0   //performance as priority SQA

 2: if mode == 'decider' then
 3:     if (mt + overhead) > ct then
 4:         runOnCloud
 5:     else
 6:         runOnMobile
 7:     end if
 8: else if mode == 'mobile' then
 9:     runOnMobile
10:     mode ← 'cloud'
11: else if mode == 'cloud' then
12:     runOnCloud
13:     mode ← 'decider'
14: end if

15: runOnMobile
16:     t1 = start time
17:     execute Callee on mobile
18:     t2 = finish time
19:     mt ← t2 - t1  //mt is mobile elapsed time
20: end

21: runOnCloud
22:     try
23:         t1 = start time
24:         execute Callee in Cloud
25:         t2 = finish time
26:         ct ← t2 - t1  //ct is cloud elapsed time
27:     catch: runOnMobile
28: end

29: mode ← 'mobile' :: on user command
```

it is a simple persistent key string storage option with most minimal overhead compared to other options (such as SD card, SQLite or remote storage).

4.4.3 Task: ***Context-aware optimisation component.***

The core function of the Task component is to execute the offloadable component i.e. handleCallee(params) of Figure 4.5. Task is the component which transforms a mobile app into MCA, by providing; socket implementation for remote execution of Callee, and adaptive context-based decision maker which makes use of the Context component of ACTS to decide when to offload. Based on the decision the Task executes the Callee on the mobile or cloud tier and returns the result to Aspect. After the Callee is handled, Task updates the Context database with dynamic context, to ensure that executions are based on recent knowledge – particularly important due to the unpredictability of MCA environment.

Why Task? ACTS Task component is implemented as a subclass of Android AsyncTask API – which is used to perform background operations without manipulating UI thread. The reason for implementing Task as an AsyncTask class is so as to avoid resource overhead of handling socket connections on the main UI thread – which could also crash the application.

Algorithm 4.1 provides the algorithm used for decision making within Task component. The decision-making process uses;

- Context DB to persist and retrieve dynamic context.
- Static context to prioritise an SQA during decision making.
- Dynamic context for adaptive decision making.

Using Context DB: The execution *mode*, *mt* and *ct* are persisted in and retrieved from the Context DB. For example (see Algorithm 4.1) lines 10 and 13 updates the mode in Context DB, and lines 19 and 26 updates the *mt* and *ct* respectively. Whereas, lines 2, 8 and 11 retrieves mode from the Context DB to check its state, line 3 retrieves/makes use of *mt* and *ct* from the Context DB.

Setting static context: The algorithm starts by setting the static context variable; i.e. *overhead*, which is 0 for mobile performance as priority attribute. For mobile energy as priority attribute, the overhead is obtained as a percentage (*x%*) of the mobile elapsed time (*mt*), i.e. *x% × mt*. The purpose is to specify the maximum performance overhead permissible for energy optimisation (therefore 0 signifies that no overhead is allowed hence performance as priority attribute).

Adaptive decision: Three execution modes are defined; *decider*, *mobile* and *cloud* mode (see Algorithm 4.1). At any mode of execution; whether an offload is done (lines 21-28) or not (lines 15-20), the elapsed times are always updated (i.e. lines 19 and 26). Thus, ensuring that the decision making process is up-to-date with changing environmental state. The initial execution mode of the Context DB is *mobile* mode – this is the mode of the first execution of the application.

Therefore when an application is executed initially, with the initial mode being set as *mobile*; thus satisfying the condition of line 8; line 9 and 10 is executed. Line 9 runs the Callee on mobile and updates the elapsed time (*mt*) in the Context DB (as shown in line 15 to 20). Line 10 then updates the mode in the Context DB to *cloud*.

On the next execution of the application, as the mode is set to *cloud*, lines 12 and 13 is then executed. Line 12 offloads to the cloud and updates the elapsed time (*ct*) in the Context DB (as shown in lines 21-28). Line 13 then updates the mode in the Context DB to *decider*.

On subsequent executions the *decider* mode uses the stored context for deciding when or when not to offload. The *decider* mode executes lines 3 to 7. The decider makes use of the *overhead* variable (at line 3) set earlier by static context (line 1), when making offload decision. If the sum of the mobile elapsed time (*mt*) and *overhead* is greater than the cloud time (*ct*) then the Callee is offloaded to the cloud for execution. As mentioned earlier, a 0 overhead (as used in Algorithm 4.1) signifies mobile performance as priority attribute, thus adding 0 to *mt* in line 3 results to the original *mt*. However, for energy efficiency as priority attribute; a non-zero *overhead* is added to *mt*, before being compared (>) against *ct*. Increasing *mt* using the non-zero *overhead* is a way to force energy-saving by offloading, despite performance compromise. In other words increasing *mt* by a percentage (using the *overhead*) is a way to prioritise offloading for the purpose of mobile energy savings, with performance trade-off (of the % of *overhead*).

Line 27 of Algorithm 4.1 implements the failover modelled in Fig 3. This is useful in case of a situation where the offload encounters an error. In such situation, the Callee is executed on the mobile tier. Note that lines 21 to 28 of Algorithm 4.1 is a simplified version of Algorithm 4.3 – which presents the complete algorithm for runOnMobile.

Also important; the adapt function in the Context component can be used to reset the ContextDB mode back to mobile execution mode; which refreshes the optimisation logic. Resetting can be useful for updating the values of the modes to most recent environmental condition for better accuracy in the optimisation logic. Thus the purpose of line 29 in Algorithm 4.1. As shown in line 29, the control is handed to the user. In other words, the user can refresh the logic at any point in time.

4.4.4 Service (and Shared Context)

Services are cloud implementations of Callee; for which execution is based on the shared context. Services receive parameters from the offload request from the Task component, handle the remote execution of the Callee and sends a response back to the Task component. Two key software qualities implemented for the cloud tier, as shown in the Caller-Callee model, are software availability and resource-efficiency. The term *Shared Context* is used here to denote the fact that the cloud tier qualities are shared for effective integration of the qualities – i.e. to minimise overhead on the application performance. Context sharing for the cloud tier quality attribute is either

- between cloud tier services (i.e. Service and Service) or

- between the cloud tier (Service) and mobile tier (Task)

Service to Service Context Sharing: Cloud qualities can be integrated into an MCA by implementing the cloud qualities (within a Service) to collaborate with another Service. In other words, the Service acts as a surrogate to the cloud tier Service which originally handles offload requests. This is exemplified by line 4 of Algorithm 4.2 which redirects the execution of the Callee to an alternate Service in a situation where the CPU is overworked – for resource

Algorithm 4.2 Shared Context for Cloud tier, Service

```
Let: x  represent available CPU in %
     y  represent CPU Threshold in %
     z  represent Time Threshold in ms

 1: result = null
 2: Thread t:
 3:     if x < y then
 4:         result = alt Service //Service to Service
            //do nothing, for Service to Task
 5:     else
 6:         result = execute Callee
 7:     end if
 8: end
 9: t.start()
10: t.join(z)
11: if t.isAlive() then
12:     t.interrupt()        //Service to Task
13: end if
14: send result to mobile
```

Algorithm 4.3 Shared Context for Mobile tier, Task

```
Let: z  represent Time Threshold in ms

 1: runOnCloud
 2:     try
 3:         t1 = start time
 4:         result = null
 5:         Socket s = new Socket(host, port)
 6:         s.setSoTimeout(z)
 7:         write Callee params through socket
 8:         result = read result from cloud
 9:         t2 = finish time
10:         ct ← t2 - t1  //ct is cloud elapsed time
11:         if result == null then
12:             runOnMobile
13:         end if
14:     catch: runOnMobile
15: end
```

efficiency. The CPU threshold represent the minimum available CPU on the cloud tier which can process the Callee request with minimal overhead. The alternate service implements a version of the Callee, however does not require redirection logic (and quality implementation), it only acts as a server surrogate to the main Service and returns the result to the main Service.

Service to Task Context Sharing: Cloud qualities can be integrated into an MCA by implementing the cloud qualities (within Service) to collaborate with the mobile tier (Task). In this type of context sharing, the Task component provides complementary features to enforce the cloud tier qualities. These complementary features are implemented within the method which handles the offload (i.e. runOnCloud) in the Task. For example; Algorithm 4.2 provides the snippet for context sharing in the cloud tier, which implements the resource efficiency and availability qualities.

According to Bass et al. [23], availability is the probability that a system will be operational when it is needed. In other words, availability is concerned with the consequences of a system failure. In Algorithm 4.2, a time threshold (*z*) is used to specify the maximum expected elapsed time for the cloud tier to complete execution of the Callee. The time threshold is therefore used to determine availability in the cloud tier. Consequently exceeding the time threshold (which is maximum elapsed time) signifies unavailability – which can be due to any number of reasons such as high demand of the software, intensive resource use/over-utilization or system failure. The performance of the Callee can also be affected by network (bandwidth and latency) which may not be manageable by the availability feature within the cloud tier (since network is external factor to the cloud processing resource). Consequently, the mobile tier acts as a surrogate to ensure availability and performance using the socket timeout (line 6) – in which case an exception is thrown to the catch clause, which runs the Callee on the mobile tier – as shown in line 14 of Algorithm 4.3.

As shown in Algorithm 4.2, the Callee execution is implemented within a thread, so as to use the thread join(*z*) and interrupt() methods to implement availability. The join method waits at most *z* milliseconds for the thread to complete. While the interrupt method implemented after the join call ends the process of the thread (see lines 11-13) if not completed within the time, *z* specified by join (in line 10). If the thread is terminated prior to completion, the result sent to the mobile (on line 14 of Algorithm 4.2) is null. The mobile complements the cloud tier Callee availability quality by executing the mobile Callee node if the result is null (as shown in lines 11-13 of Algorithm 4.3). Similarly, the mobile tier can also be used as a surrogate for cloud resource-efficiency quality, by not redirecting to alternate Service when the cloud CPU is overworked (i.e. highlighted line 4 of Algorithm 4.2), in which case a null response is sent to the mobile; and consequently, Callee execution is handled by the mobile.

4.5 Quality Verification

Quality Verification (Figure 4.6) is an evaluation process which is used to determine if an identified offloadable component (Callee) will certainly yield benefits when offloaded. The purpose of performing the quality verification after the architecture phase is to include the optimisation logics of the Mango architecture in the verification process. Consequently, phase 3 of Mango approach ensures that the remote execution time including the decision overhead within the Mango architecture, does not compromise the performance of the mobile application – at least in normal environmental conditions. Furthermore, quality verification is performed dynamically (i.e. at runtime), further details are presented in Mosaic (Chapter 5).

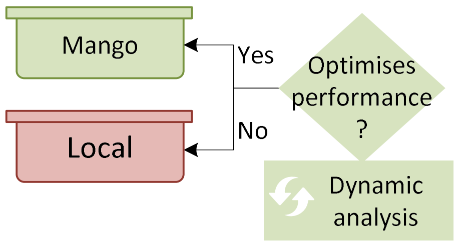


Figure 4.6 Quality Verification

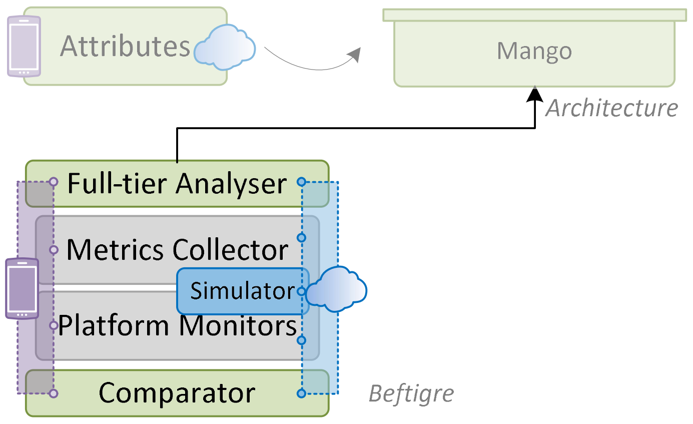


Figure 4.7 Behaviour-driven Full-tier Green Evaluation

4.6 Behaviour-driven Full-tier Green Evaluation

The evaluation approach used in Mango is called Beftigre (details presented in Chapter 6). Beftigre (abbreviation, for *behaviour-driven full-tier green evaluation*) provides an evaluation for MCA by taking into account the full-tier qualities attributes of the architecture, as shown in Figure 4.7. The challenges and difficulties of the mobile-centric architecture scenario approach fall into the category of (green[13]) software testing/evaluation [23], [34]. In response to the identified challenges (see Chapter 3), Beftigre is proposed, which adopts the behaviour-driven technique to address challenges 1 and 2. Furthermore, the thesis treats challenge 3 as a testing granularity problem, consequently resolved by fine-grained testing of mobile tier and cloud tier (i.e. full-tier).

The behaviour-driven (BDD) concept is achieved at the Comparator component as annotations: with the purpose of presenting the factors surrounding MCA test in a comparable and communicable manner. Consequently BDD terms such as Given, When and Then are used. Platform monitors are useful for monitoring the platform for target metrics whereas the Metrics Collector computes the gathered data. For instance, at the mobile tier, power and performance monitors and collectors are used for gathering metrics

[13] Green is the term used for software optimisation and testing based on the energy-efficiency and performance metrics [10], [18]. Since offloading schemes target energy-efficiency of mobile devices, similarly scenario based comparison is within the green software testing category.

for energy and performance computation. And for the cloud-tier resource monitors and collectors are used for CPU and memory metrics. Thus full-tier qualities (or attributes) are analysed at the evaluation phase. Details of the Beftigre approach and how the identified research gaps have been addressed are presented in Chapter 6 and Section 7.7.

4.7 Summary

Table 4.4 Summary of Mango Architecture

Tiers	ACTS Design Pattern	Caller-Callee Model
Mobile	Aspect	Caller
	Context	-
	Task	Callee
Cloud	Service (with Shared Context)	Callee

Class Aspect	***Collaborators***
Responsibility • Intercepts Caller calls to Callee. • Routes execution to Task. • Returns results and execution to Caller.	• Task

Class Context	***Collaborators***
Responsibility • Provides logic for persisting context to storage. • Implements logic for adapting context.	• Task

Class Task	***Collaborators***
Responsibility • Receives Callee parameters from Aspect. • Implements the optimisation logic on the Callee. • Dispatches Callee execution to mobile or cloud tier. • Adapts context	• Aspect • Context • Service

Class Service	***Collaborators***
Responsibility • Receives Callee parameters from Task. • Implements the optimisation logic on the Callee. • Dispatches Callee execution to mobile or cloud tier. • Provides shared attributes to alternate Service.	• Task • Service

Figure 4.8 Class Responsibility of ACTS

This chapter presented the Mango Architecture as a model-driven architecture which is defined by a Caller-Callee model and an ACTS design pattern (as summarised by Table 4.4). The architecture aims at integrating software quality attributes (SQAs) at the mobile and cloud tier. The SQAs explored by the architecture are performance and energy-efficiency; for the mobile tier, and resource efficiency and software availability; for the cloud tier. Although four SQAs has been proposed, more SQAs can be implemented following the Mango architectural style. The Caller-Callee model is used to integrate the specified SQAs with identified offloadable component (Callee) at mobile and cloud tier.

The greyed out cells in Table 4.4 also shows the connection between the Caller-Callee Model nodes and ACTS components. ACTS stands for Aspect-Context-Task-Services pattern. Figure 4.8 presents a summary of the responsibilities of ACTS components using the Class Responsibility Collaborator (CRC) model [110]. In Figure 4.8 the *Class* is the ACTS component, the *Responsibility* is something that the class knows or does, and the *Collaborators* are other classes that the class interacts with to fulfil its responsibilities.

The purpose of the design pattern is to implement the Caller-Callee model; thus actualising the Mango architecture in a code base for mobile and cloud tiers. The decision making in Mango architecture is based on execution contexts (at mobile and cloud tier) and assisted by a Time Threshold at both tiers (Algorithms 4.2 and 4.3). The Time threshold ensures that the architecture is robust enough to absorb any stresses on the mobile performance due to unpredictable adverse environmental conditions.

Other benefits of Mango architecture are as follows;

- *Reduced development complexity*. The implementation process – involving identification of CI tasks and implementation of their optimisation code, is simplified through modelling. The *Mosaic* framework assists in modelling.
- *Unobtrusive optimisation for legacy systems.* The approach is also suitable for optimisation of legacy systems as the Mango components

are loosely coupled to (or decoupled from) the base system though *Aspects*. As shown in the case study, no significant changes are required to be made to the existing legacy code in order to optimise the application for mobile-cloud offloading.

- *Controlled over-head, improved efficiency*. The approach makes use of adaptive time based decision making to ensure improved efficiency at runtime (as opposed to multiple environmental factors).
- *Flexible full-tier quality integration*. The approach supports seamless integration of SQAs for both mobile and cloud tiers, as a way to drive efficiency at both tiers. E.g. mobile performance and energy efficiency, cloud resource efficiency and availability.
- *Extensive reusability support*. The approach highly promotes reusability through a templating process. The *Mosaic* framework implements ACTS as templates which can be adapted to suit any specific domain requirement.
- *Taxonomy robustness*. The approach is efficient with different application taxonomies in such a way that it can adapt to accommodate non-computation intensive applications with insignificant overhead. This benefit has been verified with data intensive applications – in Chapter 7.

Chapter 5. Mosaic Modelling Approach

A Modeller and Analyser for MCA

5.1 Introduction

This chapter introduces the Mosaic framework, which is a *model-based selective approach for identification of Callees* (i.e. offloadable or computation intensive tasks in mobile applications). The framework is composed of three key parts – Selective analyser, Caller-Callee modeller and Quality Verifier as shown in Figure 5.1. The aforementioned Mosaic features fulfil the CRAC phases (presented in Chapter 4), as presented in the following bullet points.

This chapter is grouped based on the three Mosaic components;

- Selective Analyser: is used for identifying offloadable tasks by applying rules to static analysis process. The output is a call-graph (.mcg). This feature fulfils Phases 1 and 2 of CRAC process (see Chapter 4).
- Caller-Callee Modeller: is used to model Caller-Callee nodes with SQAs useful for MCA optimisation. And also used for generating the ACTS pattern code following the Caller-Callee model. This feature fulfils Phase 3 of CRAC process (see Chapter 4) – which is phase 1 and 2 of Mango approach.

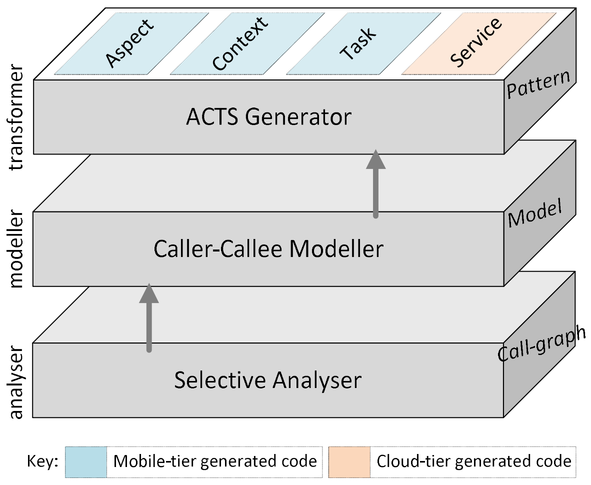


Figure 5.1 MOSAIC Framework

- Quality Verifier: is a form of architecture validator, which is used to ensure that the Mango architecture will yield performance benefits. This feature fulfils Phase 3 of Mango approach (see Chapter 4).

The three aforementioned features of Mosaic are provided as an Android library, and the modelling feature is also provided as a GUI tool.

A benefit of the Mosaic (alongside others presented in Chapter 4) over its counterparts [4], [8], is that it supports seamless modifiability of code (through the modeller tool) while in development (i.e. forward and backward-engineering of code) as opposed to existing counterparts which are not as robust. Also Mosaic possesses a better performance in identification of Callees. The core objective of Mosaic is to provide the MDE set of tools for realising the Mango approach from design to verification (phase 1 to 3 of Figure 4.2).

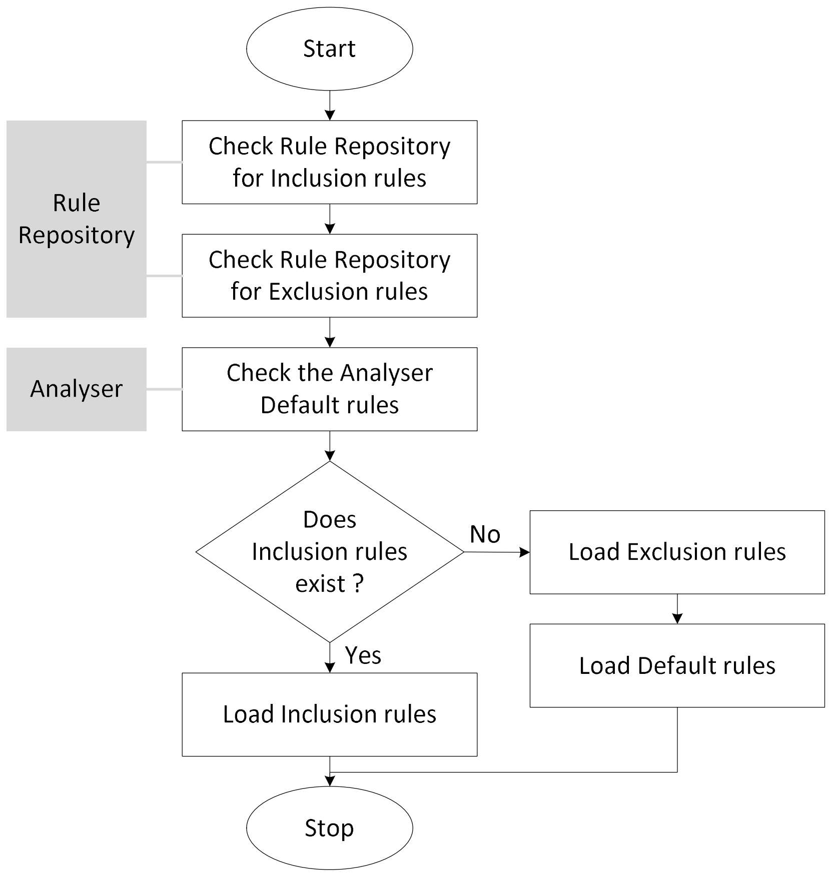

Start
Rule Repository
Check Rule Repository for Inclusion rules
Check Rule Repository for Exclusion rules
Analyser
Check the Analyser Default rules
Does Inclusion rules exist ?
No
Load Exclusion rules
Load Default rules
Yes
Load Inclusion rules
Stop

Figure 5.2 Selective Analysis Approach

5.2 Selective Analyser

To identify offloadable tasks, Mosaic first adopts the static analysis approach, adopted elsewhere [8] – which is based on analysing the classes of an application to produce a call-graph. Rules are applied to the static analysis process to enhance the identification of offloadable components – in this research this approach is referred to as a selective analysis approach. Thus; the purpose of the Selective Analyser is to achieve finer granularity in static analysis through the use of rules for identification of offloadable tasks. Since static analysis is incapable of identifying computation intensive components at finer-granularity, the selective approach is proposed to achieve finer granularity by applying rules (Algorithm 5.1) to the static analysis process. The analysis process produces a final call-graph which specifies the offload candidates in form of Caller-Callee mapping; however the most important component is the Callee – since the pointcut of AOP Aspect is used to intercept any occurrences of the Callee. Selective analysis (Figure 5.2) is based on three kinds of rules; inclusion, exclusion and default rules.

Algorithm 5.1 Selective Analysis within Mosaic

Let: r represent the rule repository

```
 1: if r.hasInclusion() then
 2:     use inclusions as call-graph
 3: else
 4:     x ← compile classes into jar
 5:     while x has call do
 6:         if nonR(call) && nonExclusion(call) && nonLang(call) &&
               nonGenerated(call) && nonConstructor(call)
            then
 7:             add call to call-graph
 8:         end if
 9:         next call
10:     end while
11: end if
12: save call-graph to file
```

1: android.app.Activity findViewById
2: android.widget.BaseAdapter *
3: in package.Class:callee

Figure 5.3 Rules Repository illustrating Inclusion and Exclusion rules

5.2.1 Inclusion Rules

From Algorithm 5.1, the Selective Analyser first checks the existence of any inclusion rules (line 1); and specifies offload candidates based on these rules if they exist (line 2). Inclusion rules are rules added to the repository for the purpose of specifying Callees/offloadable candidates allowed in the program. They are beneficial for explicitly specifying distinct offloadable candidates for evaluation and can be specified by appending the word '*in*' before the full Callee specification – i.e. package name, class name and method name, as shown by line 3 of Figure 5.3.

As mentioned earlier, the key achievement of the selective approach is the identification of offloadable tasks at a finer granularity. With the selective analyser, rules can be added to the analysis process until finer results are achieved. Also, the results of the dynamic analysis can be used to formulate rules (as inclusion rules) to achieve finer granularity of call-graph; thus allowing for flexibility in the software development process.

5.2.2 Exclusion Rules

Exclusion rules are rules added to the repository for the purpose of specifying call properties which are to be excluded from the program as offloadable candidates. During static analysis, project classes are scanned and the calls made within the class are derived. Each call consists of various properties as presented in Table 5.1. The call properties used for an exclusion rule are "sa callee" or "sa *" as shown in lines 1 and 2 respectively of Figure 5.3.

Table 5.1 Call properties used in selective static analysis

Symbol	Description
caller	The Caller, which is a method
a	Class of caller
sa	Super class of caller
callee	The Callee, which is a method
b	Class of callee

“sa callee” excludes a call if the super class of the caller and its callee are specified in the exclusion rule. For example, from line 1 of Figure 5.3, if a call has its caller superclass as Android Activity, and the called method (i.e. the callee) is findViewById, then the callee is not an offloadable candidate. This is because findViewById is an Android’s Activity native method – i.e. tied to the view of the mobile device, and therefore cannot be offloaded. Therefore the exclusion rules are useful for excluding device/sensor-centric call properties. “sa *” is similar to “sa callee”, but the * is used as a wildcard character to exclude all methods (callees) associated with the specified sa.

The reverse of the exclusion rules must be met by a call in order to be considered as an offloadable candidate, i.e. nonExclusion(call), as shown in line 6 of Algorithm 5.1.

5.2.3 Default Rules

Default rules, like exclusion rules, specify call properties to be excluded from the program as offloadable candidates. The difference between exclusion and default rules is that exclusion rules are specified within the repository, and therefore with custom exclusion rules can be added to tailor the analyser for a program. Default rules, however, are standard exclusion rules which cannot be modified as they are provided by the system. These rules are useful to exclude calls which are generally mobile device or platform specific. Like the exclusion rules, the reverse of the default rules must be met by a call in order to be considered as an offloadable candidate (see line 6 of Algorithm 5.1). The constituents of the default rules are presented below;

- nonR(call)

The rule is used to exclude a call if the class of the Caller is R. R is an Android generated resource class – used for assets such as widgets and layouts. It is mobile device specific, and thus cannot be offloaded to the cloud.

- nonLang(call)

The rule is used to exclude a call if the class of the Callee is a Java or Android language class. This is determined by checking if the package name of the class (of the Callee) begins with ‘android’ or ‘java’ before the first period (i.e. android. or java.). The java.lang and android platform classes are defined by the JVM and DVM runtimes and thus cannot be offloaded.

- nonGenerated(call)

The rule is used to exclude a call-graph item if the Caller or Callee is a constructor. This is determined by checking if the Caller or Callee is <init>. <init> in Java signifies that a method is a constructor. A constructor alone cannot be offloaded as they are required for object creation, moreover, it is impractical to offload a constructor without their dependencies.

- nonConstructor(call)

The rule is used to exclude a call-graph item if the Caller, the Callee, the class of the Caller, or the class of the Callee are generated by the Java platform. This is determined by checking if the class or method names contains the dollar $ sign. In Java, $ is used to annotate the names of inner classes (e.g. $class_name) or anonymous inner classes (e.g. $number); and their methods (e.g. $number). By convention, these forms of classes are used to implement platform specific methods e.g. ActionListener and actionPerformed used in swing programming. Thus they are not appropriate offload candidates.

5.3 Caller-Callee Modeller

Caller-Callee Modeller is used to model Caller-Callee nodes with SQAs useful for MCA optimisation. The Caller-Callee model is introduced in Chapter 4. This section presents the modeller to actualise/validate the model introduced in Chapter 4. The Caller-Callee modeller is provided in two forms;

- A modelling tool: The modelling tool (or Modeller, for short) is a GUI – built on the JGraphX[14] swing API [111]; used to independently model offloadable components like any independent MDE tool (e.g.

[14] JGraphX User Manual is useful for extending the modelling layer and is located at https://jgraph.github.io/mxgraph/docs/manual_javavis.html

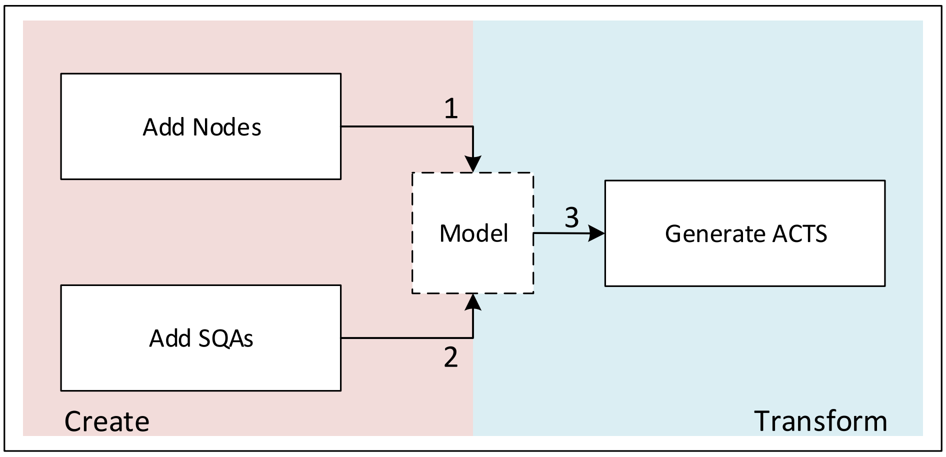


Figure 5.4 Modeller process

MySQL Workbench). The tool generates two forms of output; first the model diagram (.mod file), and the transformed model (ACTS classes). The tool is useful for the design process of new systems as it separates the model diagram from the transformed model. Appendix C shows a screenshot of the Modeller and sample model diagram (.mod) file.

- A library feature: In this case, the modeller is integrated with the Mosaic library (.jar) file which can be loaded into the IDE during development. The output of the modeller in the library is the transformed model (ACTS classes). The library is useful for legacy systems (and for continuous integration), as it is loaded within the development environment.

The modeller uses Caller and Callee as nodes (obtained from earlier presented call-graph) to create the model – while specifying the SQAs for the model, as shown in steps 1 and 2 of the Modeller process (Figure 5.4). The modeller process is completed by generating ACTS classes (i.e. step 3) which is an implementation of the model diagram.

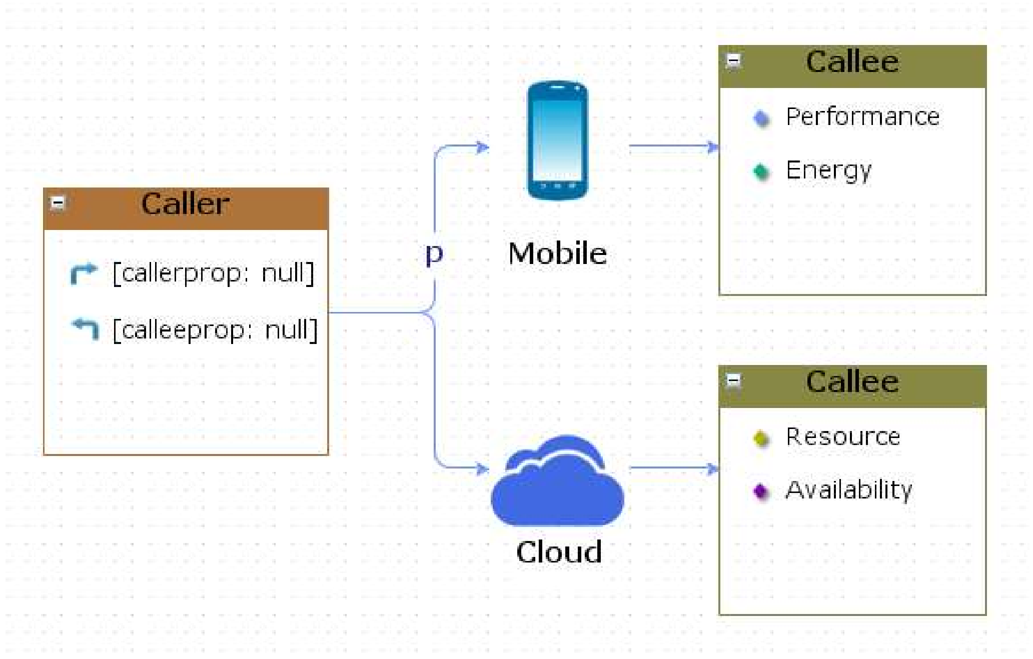


Figure 5.5 A valid Caller-Callee model diagram from Modeller

5.3.1 Model Creation

As shown in Figure 5.5, the Modeller defines four nodes (as presented in rule r, in Algorithm 5.2) and four attributes (as presented in rule sr, in Algorithm 5.2). The nodes are; Mobile, Cloud, Caller and Callee node. The attributes are; mobile performance (p), mobile energy (e), cloud resource (r), and software availability (a).

The Caller-Callee model is a graph of nodes (or vertices), connectors (or edges), and attributes (i.e. SQAs). Algorithm 5.2 presents the process for validating nodes – Lines 1-8, connectors – Lines 9-17, and the model as a whole (nodes, connectors and attributes) – Lines 18-24.

5.3.1.1 Node Validation

Node validation is used to validate the model when a node is added to the model, during the model creation. The node validation algorithm (Lines 1-8) is implemented as an event listener for a drag and drop event on the node. The condition in Line 3 ensures that for any model; there is only one instance of all nodes except Callee node, for which the condition ensures there are only two instances of it. The reason for two instances of a Callee is that it is required to be implemented on the mobile and cloud tiers.

5.3.1.2 Connector Validation

Connector validation is used to validate the model when a connector is added to the model, during the model creation. The connector validation algorithm (Lines 9-17) is implemented as an event listener for a drag and drop event on the connector. The modeller defines Connectioln rule (*r*) used to validate

Algorithm 5.2 *Model validation* algorithm

```
Require: Model in view as m
         Connection rule as r ← { (Caller, Mobile, 1, 1), (Mobile, Callee, 1, 1),
                                  (Caller, Cloud, 1, 1), (Cloud, Callee, 1, 1) }
         SQA rule as sr       ← { (Mobile, Callee, "p, e"), (Cloud, Callee, "r, a") }
         m.size(a)            ← return the number of a nodes in m
         r.isValid(a, b)      ← return true if a links to b in r
         m.validate(rule)     ← validates connections in m against rule
//(A,B,1,1)   in r  means A links to B in 1 to 1 bi-directional relationship
//(A,B,"x,y") in sr means B in A can only have x and/or y SQA
Let:     a    represent action performed
         e    represent drag event

 1: if e is on node then
 2:     n ← get the node dropped in model
 3:     if m.size(n) == 0 || ( n is Callee && m.size(n) < 2 ) then
 4:         add node
 5:     else
 6:         ignore node
 7:     end if
 8: end if

 9: if e is on connector then
10:     s ← get the source node of the connector
11:     t ← get the target node of the connector
12:     if r.isValid(s, t) then
13:         connect s to t
14:     else
15:         ignore connection
16:     end if
17: end if

18: if a is validate action then
19:     if m.validate(r) && m.validate(sr) then
20:         'The model is valid'
21:     else
22:         'The model is invalid'
23:     end if
24: end if
```

connectors. The connection rule (r, presented in Algorithm 5.2) is used to define the relationship between nodes, thus specifying the kind of connections (source/target) allowed for a node. The condition in Line 12 ensures that the source and target nodes of a connector comply to r.

5.3.1.3 Model Validation

Model validation is used to validate the model as a whole, after the model creation is completed, prior to model transformation. The model validation algorithm (Lines 18-24) is implemented as an action, thus can be executed any time during or after the model creation. It is however required for verifying that the model complies with the defined rules. Two set of rules are defined for model validation, they are; Connection rules (r) and SQA rules (sr). While the Connection rules define the connections between nodes as mentioned earlier, the SQA rules define the software qualities allowed for a connection.

As shown in sr in Algorithm 5.2, performance and energy SQAs are applicable to connections from Mobile to Callee node as they are mobile tier SQAs. Furthermore, resource and availability SQAs are applicable to connections from Cloud to Callee node as they are cloud tier SQAs. These SQAs have been detailed in Chapter 4. Line 19 validates the model by validating the nodes, connectors and attributes.

- Validating the connection (nodes and connectors)

$m.\text{validate}(r)$ is used to validate the connections (i.e. connectors and nodes) in the model m against the connection rule r.

Connectors are validated by verifying that all the connectors in the model comply with the connection rule. E.g. Caller must be connected to Mobile, Mobile must be connected to Callee, etc. as defined by r.

Whereas nodes are validated by verifying that all the rules in r were implemented by a connection. E.g. connections; Caller to Mobile, Mobile to Callee, Caller to Cloud and Cloud to Callee must exist in a 1 to 1 relationship in the model. This would result in one Mobile node, one Cloud node, one Caller node and two Callee nodes (one for the mobile tier and one for the cloud tier).

- Validating the attributes

$m.\text{validate}(sr)$ is used to validate attributes of the connections (i.e. connectors and nodes) in the model m against the SQA rule sr.

The attributes of connectors in the modeller specify priority attributes used for execution (presented in Chapter 4). Priority attributes apply to the mobile qualities; thus, $m.\text{validate}(sr)$ verifies that the priority attributes on the connectors comply to sr. E.g. the attribute of the connector must be either ‘p’ or ‘e’.

The attributes of the nodes specify all SQAs to be implemented in a tier (mobile or cloud). $m.\text{validate}(sr)$ also verifies that the attributes on the nodes comply to sr. The nodes of interest are the Callee nodes. Thus $m.\text{validate}(sr)$ verifies that the Callee node with Mobile parent has ‘p’ or/and ‘e’ and the Callee node with Cloud parent has ‘r’ or/and ‘a’ as SQAs.

A valid Caller-Callee model (e.g. Figure 5.5) is utilised by the modeller to engineer MCA application based on the ACTS design pattern (i.e. model generation for Mango architecture). By adopting a standardised modeller (built on JGraphX), a created Caller-Callee model diagram can be easily modified and reused in different applications. Thus the mosaic framework supports modifiability and reuse of artefacts during development.

5.3.2 Model Transformation

The modeller also fulfils the model transformation process in Mosaic framework which uses *templates* to transform a model into an application code-base compliant to the mango architectural style. Templates are .tmp files implementing the ACTS design pattern described in Chapter 4. Furthermore, templates constitute valid *code bodies* with *placeholders* – which are

Table 5.2 Template targets

ACTS Templates (*.tmp)	Description	Tiers
Aspect	AspectJ Aspect	Mobile tier
Context	Plain Class	
Task	Android Task	
Service	Plain Class	Cloud tier

substituted with appropriate values during transformation and *tags* – which are integrated as SQAs.

5.3.2.1 *Code bodies*

Table 5.2 presents the code bodies (or targets) of the templates. The Aspect template is based on AspectJ Aspect class – and it makes use of the annotation style AspectJ as shown in Figure 5.6. The Task template is based on Android Tasks; hence a subclass of the Android Task class. Context and Service templates are both plain Java classes. All generated code have .java extensions, in other words, they are Java classes.

5.3.2.2 *Placeholders for Callee Properties*

Placeholders are specified with square brackets within the .tmp file as shown in the Aspect template presented in Figure 5.6. Appendix D presents the complete ACTS templates used in the modeller process. The placeholders in templates are of two types: i) meta-model placeholders; which are

```
package mango;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;

@org.aspectj.lang.annotation.Aspect
public class Aspect {

    @Pointcut("call(* [Callee](..)) && args([ArgumentIDs])")
    public static void offloadMethod([Arguments]) {
    }

    @Around("offloadMethod([ArgumentIDs]) && !within(Aspect) && !within(Task)")
    public [Return] aroundOffloadMethodCall(ProceedingJoinPoint jp, [Arguments]) throws Throwable {
        return new Task().execute(new Object[]{[ArgumentIDs]}).get();
    }
}
```

Figure 5.6 Aspect Template

placeholders derived from call-graph meta-model. They are [Callee], [Arguments], [ArgumentIDs], [CastedArguments] and [Return] placeholders. And ii) custom placeholders; which are placeholders independent of the meta-model but added to expedite development. They are [Host] and [Port] placeholders.

[Callee] placeholders are substituted with the actual class name during transformation. Although the Caller is presented in the model, in the actual implementation, the Caller class is not required, as Aspect pointcut is used to intercept the program at any point where the Callee is called. [Callee] placeholders are used within Aspect and Task templates.

[Arguments] placeholders specify the argument types and identifiers of the Callee. [Arguments] placeholders are used in Aspect template. Within the Aspect class, they are applied at the method declarations of the pointcut and advice, as shown in Lines 11 and 15 of Figure 5.6 respectively. These are useful for obtaining the arguments of the Callee when the call is intercepted.

[ArgumentIDs] placeholders are substituted with identifiers of the arguments. While the [Arguments] placeholder consists of argument types and identifiers, the [ArgumentIDs] only specifies the identifiers. These are useful in the Aspect class for passing the arguments to the Task to launch the execution, as shown in Line 16 of Figure 5.6. They are also used in the pointcut and advice code of the Aspect class for specifying the argument identifiers of the Callee, as shown in Lines 10 and 14 of Figure 5.6 respectively.

[CastedArguments] placeholders are [ArgumentIDs] which are casted with the argument types from [Arguments]. [CastedArguments] are used as arguments for [Callee] in order to make a call to the offloadable task. [CastedArguments] are used in the Task and Service classes of ACTS when calling the offloadable task.

[Return] placeholders are substituted with the return type of the Callee. [Return] placeholders are used in Aspect template as shown in Line 15 of Figure 5.6. They are also used in Task template.

[Host] placeholders are used to specify the host IP address of the cloud. They are used in the Task template.

[Port] placeholders are used to specify the port number which the cloud server is listening to. They are used in the Task and Service templates.

A demonstration of the transformation of ACTS by Mosaic framework is presented in the Case studies (Chapter 7).

5.3.2.3 Tags for Quality Attributes

Four quality attribute (SQA) types are provided by the Mango architecture as presented in the Model's SQA rule (sr in Algorithm 5.2). These are; performance and energy-efficiency for the mobile tier; resource-efficiency and software availability for the cloud tier.

Within the Modeller (as shown in Figure 5.5), the attributes for tiers are specified on the mobile tier (i.e. Mobile → Callee node) and cloud tier (i.e. Cloud → Callee node). And the priority attribute (e.g. p) is specified on the connector. Quality attributes are mapped to sections of the templates using tags. In order words, tag sections are used to wrap the implementation of a quality attribute, for a given template.

During model transformation, the quality attributes specified on the tiers are mapped to sections of the template as described below. sr in a tag denotes

```
<sr:p>
    private static final int OVERHEAD = 0;
</sr:p>

<sr:e>
    private static final int OVERHEAD = /*Value ms*/;
</sr:e>

<sr:pe>
    private static char pa = /*get priority attribute from UI*/;

    private static int overhead(){
        if(pa=='e'){
            return /*Value ms*/;
        }
        return 0;
    }
</sr:pe>
```

Figure 5.7 Mobile tier tags at Task Template

SQA rule, p denotes performance, e denotes energy-efficiency, r denotes resource-efficiency and a denotes availability. Notice that the sr tag attributes (i.e. p, e, a, r) match those defined in the Modeller in Algorithm 5.2. The Mosaic framework recognises pre-defined tags and associates them to a model diagram (.mod). In creating quality attribute sections in templates the sr tags must be used in an open and close format, similar to HTML tags convention.

Mobile tier qualities (shown in Figure 5.7) are implemented within the Task template. The mobile tier qualities currently implemented in Mango are performance (sr:p, lines 1-3) and energy-efficiency (sr:e, lines 5-7). These are presented in using the sr tags, notice that qualities can be combined at the mobile tier (e.g. sr:pe for performance and energy-efficiency combined, lines

```
<sr:a>
    public static void dispatcher(Object[] params) {
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                result = [Callee]([CastedArguments]); //Callee on this server
            }
        });
        thread.start();
        try {
            thread.join(TIME_THRESHOLD);
            if (thread.isAlive()) thread.interrupt();
        } catch (InterruptedException ex) {  }
    }
</sr:a>

<sr:r>
    public static void dispatcher(Object[] params) {
        if (availableCPU() < CPU_THRESHOLD) {
            result = /*reference to the Callee on an alternate server*/
        } else {
            result = [Callee]([CastedArguments]); //Callee on this server
        }
    }
</sr:r>

<sr:ar>
    public static void dispatcher(Object[] params) {
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                if (availableCPU() < CPU_THRESHOLD) {
                    result = /*reference to the Callee on an alternate server*/
                } else {
                    result = [Callee]([CastedArguments]); //Callee on this server
                }
            }
        });
        thread.start();
        try {
            thread.join(TIME_THRESHOLD);
            if (thread.isAlive()) thread.interrupt();
        } catch (InterruptedException ex) {  }
    }
</sr:ar>
```

Figure 5.8   Cloud tier tags at Service Template

9-18). Using the format presented in Figure 5.7, more software quality attributes can be integrated into the mobile tier of Mango architecture.

Furthermore, Cloud tier qualities (shown in Figure 5.8) are implemented within the Service template. The cloud tier qualities currently implemented in Mango are availability (sr:a, lines 1-15) and resource-efficiency (sr:r, lines 17-25). These are presented using the sr tags. Notice that qualities can be combined at the mobile tier (e.g. sr:ar for performance and energy-efficiency combined, lines 27-45). Using the format presented in Figure 5.8, more software quality attributes can be integrated into the cloud tier of Mango architecture.

5.4 Quality Verifier

The Quality Verifier is used to validate the Mango architecture. In Mosaic framework a Profiler Aspect class is the implementation of the Quality Verifier

Algorithm 5.3 Profiler Aspect

```
Require: callee from call-graph,
        scenario from Profiler DB
//By default scenario is set to local

 1: before callee:
 2:     t1 = start time
 3: end

 4: after callee:
 5:     t2 = finish time
 6:     t  = t2 - t1
 7:     if scenario == 'local' then
 8:         lt ← t
 9:         scenario ← 'mobile'
10:     else if scenario == 'mobile' then
11:         print 'mobile overhead is ' + (t - lt)
12:         scenario ← 'cloud'
13:     else if scenario == 'cloud' then
14:         cs ← lt - t
15:         print 'cloud saving is ' + cs
16:         scenario ← 'decider'
17:     else if scenario == 'decider' then
18:         if cs > 0 then
19:             print 'decider saving is ' + (lt - t)
20:         end if
21:         reset Context DB
22:         scenario ← 'mobile'
23:     end if
24: end
```

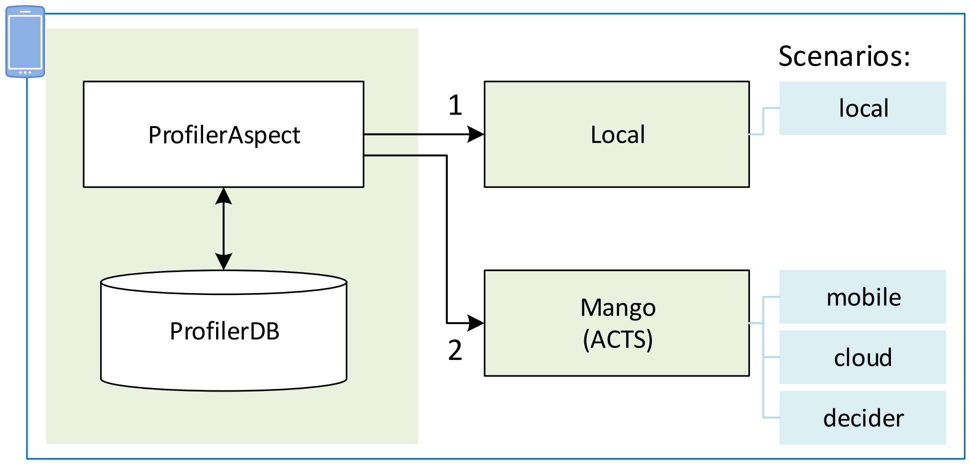


Figure 5.9 Profiler Aspect for Architecture Verification

which handles the architecture validation. The Profiler Aspect (Algorithm 5.3) is an AOP Aspect class used to evaluate an identified offloadable candidate to determine if it could yield benefits when offloaded. The Profiler Aspect is composed of a *before* and *after* pointcut (lines 1-3 and 4-24) which marks the points before and after the Callee execution using timestamps (lines 2 and 5) – so as to calculate the elapsed time of execution. The use of AOP for component evaluation is explored in greater detail in [84].

The Profiler Aspect is firstly generated by Mosaic for an offload candidate (or Callee) and is used for the evaluation of Callee performance in local execution and `Mango` execution scenarios (Figure 5.9).

The Profiler Aspect (Algorithm 5.3) adopts a similar flow of execution as the decision maker within the Task component (Algorithm 4.1). In other words, the *scenarios* within the Profiler Aspect are congruent to the modes in the decision-maker and stored within the Profiler DB for the purpose of administering evaluation for all execution modes. The default scenario is the *local* scenario, as it is executed before Mango scenarios.

5.4.1 Measuring Local Execution of Callee

After a Callee is identified and the Profiler Aspect generated, the mobile app is then executed to measure the elapsed time for the *local* execution scenario. The elapsed time for *local* is stored in profiler DB (which is an Android shared preferences storage) – line 8. And the scenario is then set to *mobile*, for the purpose of evaluating the first Mango mode (which is *mobile* by default – Algorithm 4.1)

5.4.2 Measuring Mango Execution of Callee

After the generation of ACTS components, the application can be executed with the Profiler Aspect to measure the Mango implementation of the offloadable candidate (*based on best possible environmental scenario*[15]). Repeating execution of the Mango optimised application will evaluate the *mobile*, *cloud* and *decider* modes of Mango – thus capturing all modes.

Lines 21 and 22 reset the Context DB (i.e. the mode becomes set to default *mobile*), while setting the *scenario* back to *mobile*. This is to allow the repetition of the evaluation for the *best possible environmental scenario*. The purpose of using the best possible Mango execution scenario is to determine if the offload candidate will most certainly yield benefits – i.e. given any arbitrary environmental condition from best to adverse.

5.4.3 Comparing Execution Scenarios

The mobile overhead, the cloud saving and the decider saving are estimated by the Profiler Aspect for various execution scenarios.

The *mobile overhead* (line 11) is a comparison between the execution time of Mango's mobile scenario and that of the local scenario. This is used to estimate the possible overhead contributed by Mango to the execution of the Callee when it is not offloaded.

The *cloud saving* (lines 14-15) is a comparison between the execution time of the local scenario and that of Mango's cloud scenario. If the *cloud saving* is

[15] Best possible environmental scenario means that the `mango` execution can be repeated to obtain lowest elapsed times for `mango` execution – stored in profiler repository.

a positive value, then the offload candidate will yield benefits, otherwise it will not.

The *decider saving* (lines 18-20) is obtained if the cloud saving is positive. And it is used to estimate the savings when the decision maker is executed prior to offload.

Chapter 7 presents an evaluation of Mosaic using Case Studies (with Section 7.4 demonstrating, in particular, the effectiveness of the Profiler Aspect).

5.5 Mosaic Library Usage

The Mosaic library and modeller are the key tools that make up the framework. The modeller as presented in section 5.3 is used for creating or modifying the Caller-Callee model diagram. The mosaic library, however, implements all the features of the framework – which includes selective analyser, modeller and

build.gradle (Project: Sample)

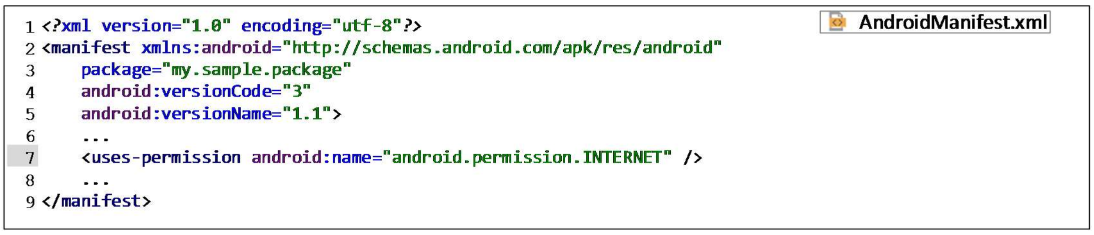
```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.0'
        classpath 'com.uphyca.gradle:gradle-android-aspectj-plugin:0.9.+'
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}
```

Figure 5.10 Project Build Gradle for Mosaic

build.gradle (Module: app)

```
apply plugin: 'com.android.application'
apply plugin: 'android-aspectj'

...

dependencies {
    compile 'org.aspectj:aspectjrt:1.8.+'
    compile fileTree(include: ['*.jar'], dir: 'libs')
}

javaexec {
    classpath files('libs/mosaic.jar')
    main 'mosaic.Main'
    args file('..')                                 //project home
    args android.getSdkDirectory().getAbsolutePath()  //android SDK directory
    args android.compileSdkVersion                  //compile SDK version (and support Version)
    args android.defaultConfig.applicationId        //core package(s)
    args 'C:\\AndroidProjects\\rules.mrl'           //rules repository or 0
    args 0                                          //jars used or 0
    args '[p,e],[a,r],[p]'                          //mobile SQAs, cloud SQAs, priority SQA
}
```

Figure 5.11 App Build Gradle for Mosaic



```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="my.sample.package"
4     android:versionCode="3"
5     android:versionName="1.1">
6     ...
7     <uses-permission android:name="android.permission.INTERNET" />
8     ...
9 </manifest>
```

Figure 5.12 Android Permission for Mango

quality verifier; apart from the diagramming interface of the modeller. Therefore, as the library is not GUI-based, it provides an interface (presented in section 5.5.2) for specifying SQAs in order to transform model for ACTS classes. The library has been tested using Android studio, and the formats presented below are associated with Android studio.

5.5.1 Library Dependencies

The Mosaic library is a jar file which can be loaded into the mobile application being developed, via the Gradle script of the application. To setup the mosaic library in the project;

- Add the annotation style Android plugin as a classpath dependency of the project as shown in Line 8 of 10. The Aspect code generated by the library is dependent on AspectJ annotation-style library. Furthermore; apply the AspectJ plugin within the app build.gradle and add the library as compile dependency as shown in Lines 2 and 7 respectively of Figure 5.11.
- Add the mosaic jar as a compile dependency in the app build.gradle as shown in Line 8 of Figure 5.11.
- Setup the arguments for the mosaic library within the app build.gradle as shown in Lines 11-21 of Figure 5.11.
- An application must specify the INTERNET Android permission in the manifest file in order to use the Mango architecture; as shown in line 7 of Figure 5.12. The INTERNET permission is used for connecting to the cloud via the Task class (of ACTS pattern). Note that the Context class (of ACTS pattern) uses Shared Preferences for storing context

information and therefore does not require permission via the Manifest file.

5.5.2 Library Arguments

The Mosaic library supports seven arguments as presented in Figure 5.11, Lines 14-20.

The first argument is the project home (Line 14). This is the directory that holds all the project files. It can be referenced from Gradle using `file('..')` or by providing the absolute path e.g. `'C:\\AndroidProjects\\Sample'` where `Sample` is the home folder created by Android studio to hold the `Sample` project.

The second argument is the Android SDK directory (Line 15). This is the directory where the Android SDK was installed to. It can be referenced from Gradle using `android.getSdkDirectory().getAbsolutePath()` or by providing the absolute path e.g. `'C:\\Users\\Chinenyeze\\AppData\\Local\\Android\\sdk'`.

The third argument is the compile SDK version used by the project (Line 16) and the support version if applicable. If the project uses a support SDK, it is appended to the compile SDK argument, separating the two with a colon. If a project uses support SDK it is indicated in the dependencies of the app Gradle.build. E.g. `compile 'com.android.support:support-v4:23.0.0'` would be part of the dependencies in Figure 5.11, to use a version 4 support SDK. The compile SDK version can be referenced from Gradle using `android.compileSdkVersion` or by providing the string explicitly, e.g. `'android-23'`. The support SDK version can be provided by appending a colon and the support SDK version to the compile SDK version. E.g. `'android-23:4'` would refer to compile SDK version 23 and support SDK version 4. Using Gradle variable format, the sample argument would look like `android.compileSdkVersion+':4'`. A compile SDK is required for the third argument, the support SDK is only required if applied to a project.

The fourth argument is the core package(s) of the application (Line 17). The specified package(s) must be the packages of interest for analysis. The default package of the application can be referenced from Gradle using `android.defaultConfig.applicationId` or explicitly e.g. `'com.sample.mypackage'`. To reference more than one package, the argument can be provided as a comma-separated string of packages, e.g. `'com.sample.one, com.sample.two'`.

The fifth argument is the rules repository (Line 18). This is an optional argument. If not provided the analyser would use the default exclusion rules (see section 5.2) to generate a call-graph. If provided, the argument must be the absolute path to the rules repository, e.g. `'C:\\AndroidProjects\\rules.mrl'`. The granularity of default exclusion rules in identifying offload candidates is evaluated in section 5.6.

The sixth argument is the jars used (Line 19). This is an optional argument. However if any external or third-party jar(s) were referenced by the project, they are provided in this argument, e.g. `'MyJar.jar'`. The jar(s) are required by the analyser to analyse the project. If more than one jar apply they are separated by a comma, e.g. `'JarOne.jar, JarTwo.jar'`. The jar names are required and not the absolute path, as the analyser references the jar from the lib directory of the project.

The seventh argument is the specification of SQAs for the model (Line 20). This argument is required for the model transformation. SQAs are specified using character symbols (as presented in SQA rule, sr in Algorithm 5.2). As shown in line 20, the first set of SQAs are for the mobile tier, followed by the cloud tier SQA and subsequently the priority attribute.

The fifth and sixth arguments can be replaced with a 0 (as used in Line 19) in order to skip any of the arguments.

5.5.3 Code Refactoring

The framework generates the code-base to be used for development. However, a few refactoring processes are required (as demonstrated in the Case Studies – Chapter 7), as the code-base is only a scaffold based on the generic template.

- Mobile tier requires Android Activity

Aspect requires an Android activity to launch/initialise the Task and Context class. An activity can be exposed by any Android Activity class (e.g. the Activity class on which the Caller is implemented) and then used within the Aspect class.

- Mobile tier requires socket connection and permission

The Task class at the mobile tier uses Java socket to connect to the cloud, in order to execute the Callee remotely, the host and IP address has to be supplied to the socket. Also, the parameters sent to the cloud and the result must be cross-checked to be appropriate to the Callee. Furthermore, the Android internet permission must be provided in the Android manifest file, to allow for the remote communication.

- Mobile tier requires Service in Manifest

The Service class of the mobile tier are Android services, and consequently; they are required to be specified in the Android manifest file.

- Cloud tier requires Callee implementation

The analyser only provides a method of the Callee for the cloud tier template. Consequently, the Callee implementation is required for the Callee method within the Task class of the cloud tier.

5.6 Performance Evaluation

The performance evaluation was performed for Mosaic library on Windows 10 x64 PC, with Intel i7 2.20GHz CPU and 8GB memory, using four Android applications as case studies – Linpack, MatCalc, MathDroid and NQueen. Furthermore, the average of 20 test samples in each app was used to investigate the performance of the library by comparing its *Mosaic* build time

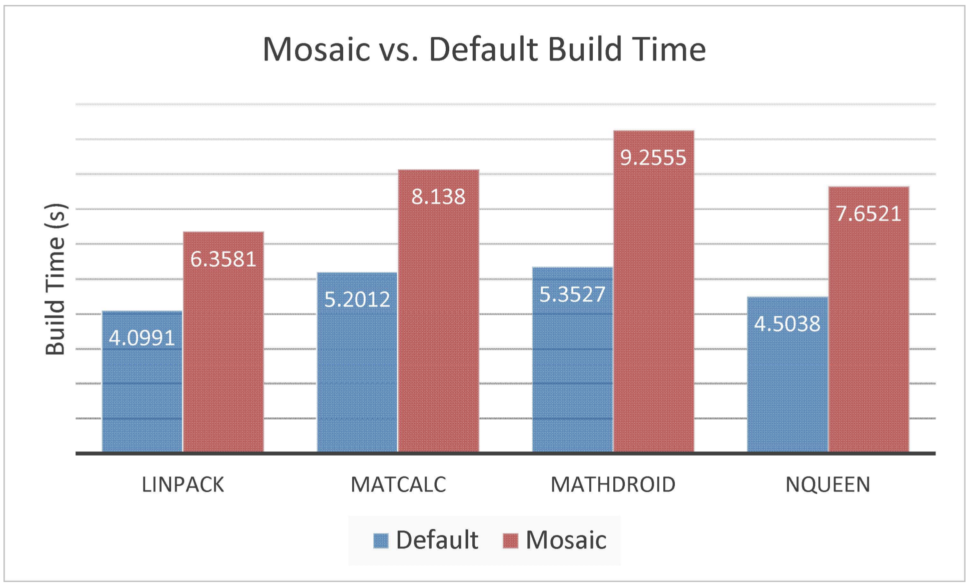


Figure 5.13 Mosaic vs. Default Build Time

with that of *Default*. The build time[16] is used as a basis for evaluation as this is when the Mosaic library is executed. The build time is measured using Android Studio's build functionality.

- *Default* scenario is the build time without the Mosaic library.
- *Mosaic* scenario is the build time which uses the Mosaic library, and also performs the selective analysis and generation process.

Table 5.3 Mosaic file extensions

Apps	**Dependent Libraries**		**Call-graph Items**		
	Jar	**Size (KB)**	**Total**	**Default Rule**	**Custom Rule**
Linpack	-	-	249	2	1
MatCalc	Jama-1.0.2.jar	33	1280	30	1
MathDroid	Calc.jar	233	2666	31	1
NQueen	achartengine1.0.0.jar	99	2077	24	1

Key:		
	Total	Total call-graph items
	Default Rule	Final call-graph items identified using Mosaic default rule
	Custom Rule	Final call-graph offloadable item identified using Mosaic custom rule

[16] Build time is the time to (re)build the modules and libraries of a project. Mosaic library is executed at build time.

Based on the results in Figure 5.13, Linpack, MatCalc, MathDroid and NQueen shows a 55.11%, 56.46%, 72.91% and 69.9% increase respectively resulted by the Mosaic analysis at build time. Consequently showing an overhead of 55-73% increase in build time.

Although the results (as illustrated by Figure 5.13) shows that Mosaic incurs some overhead (55-73% overhead) to the build time of an application, some of the applications (e.g. MathDroid and NQueen) portray higher overhead than the others (e.g. Linpack and MatCalc). The increased overhead in Mosaic scenario is affected by three key factors (summarised in Table 5.3):

- **Total call-graph items.** The more the number of call-graph items the higher the build time. As shown in Figure 5.13 and Table 5.3, Linpack has the most minimal total number of call-graph items (249), whereas MathDroid has the highest (2666). Consequently, the Mosaic overhead in Linpack is the least (55.11%), and that of MathDroid is the highest (72.91%) of the case studies.
- **Existence of Jar dependencies.** Mosaic library loads dependencies into the classpath when analysing an application, (without the dependencies in the classpath, the analysis will be unsuccessful as classes which reference the dependent jars will break the analysis process). Since the jars are loaded into the classpath, their classes are also subjects of the analysis process. Consequently, an application which has no dependency will have lesser build time compared to an application with dependency. E.g. Linpack app which has no jar dependencies (and therefore lesser number of call-graph items), has much lesser build time in comparison to other apps which has jars (as shown in Table 5.3).
- **Size of Dependency Jar files.** Jars can be of different sizes, a jar with larger size means that more classes are implemented in the jar – thus increasing the total call-graph items. And consequently, requiring more time for analysis.

Table 5.4 Offloading Candidates

App	Offloading Candidates (Callees)
Linpack	`rs.pedjaapps.Linpack.Linpack` class
MatCalc	`times(Matrix B)` method of `Matrix` class
MathDroid	`computeAnswer(String query)` method of `Mathdroid` class
NQueen	`nQueenCount(int input)` method of `NQueen` class

Despite the 55-73% increase in build time, the total build times are less than 10s (MathDroid being the maximum, at 9.2555s). Similar frameworks, which however is useful for only analysis (for identification of offloadable tasks) such as Elicit [8], records an average analysis time of 30-40s, which is in fact 233-344% increase on the Mosaic elapsed (build) time. Considering that the mosaic build time includes the full features of the framework using default model, and not only analysis process, the Mosaic framework saves development time, with an absolute accuracy when the custom rule is used (see sections 5.2.1 and 5.2.2).

What is meant by absolute accuracy? This is in comparison with the existing counterpart analysis tool – Elicit. Hassan et al. [8] uses similar applications presented in Figure 5.13 (i.e. MatCalc, MathDroid and NQueen) to evaluate the performance of Elicit, which results to 30-40s on average. Furthermore, the effectiveness of Elicit for identification of offloadable items is investigated; and the items identified as offloading candidates for MatCalc, MathDroid and NQueen are 1, 5, and 2 in number. Using the Mosaic framework, rules are applied to identify the same and specific offloading candidates, as shown in Table 5.4. Thus templates can be generated for specific identified candidates.

Note that; an application build time can be sped up by commenting out the Mosaic execution command (i.e. javaexec in Gradle in Figure 5.11) after offloading candidate (Callee) have been identified, and templates generated.

5.7 Summary

This chapter presented the Mosaic Framework as an MDE tool for Mango architecture to address the challenges (see Related Work, Chapter 3 for

Table 5.5 Mosaic file extensions

Mosaic file extensions		Description
Mosaic rule repository	.mrl	Used for selective analysis
Mosaic call-graph	.mcg	Output of selective analysis Used for model creation
Caller-Callee Model	.mod	Output of model creation Used for model transformation
Mosaic template	.tmp	Used for model transformation
ACTS classes	.java	Output of model transformation

details) of the existing MCA optimisation approaches – most especially addressing the gap of development and performance inefficiency (i.e. the key aims of Mosaic). The framework was developed to drive the Mango architecture. As a model-based framework, Mosaic generates application code, by use of a *model* from Modeller, for identified offloadable components (identified by the Selective Analyser). Table 5.5 summarises the files used within the framework.

The model (.mod) is created using the Caller-Callee Modeller (or can be obtained from the Transformer; for default model). It is the file that specifies the SQAs used for the mobile and cloud tiers and the defined contexts used within the application. In Mosaic, the model makes it easy to reuse SQAs in a MCA application, by abstracting the SQA logic as models, decoupled from the actual Caller-Callee definitions within the call-graph.

The ACTS templates (.tmp) implement the ACTS design pattern presented in Chapter 4, and are used as the code-bodies for scaffolding the Mango derived MCA application. The model transformation process uses templates to generate the ACTS classes (which are Java classes) implementing the Mango architecture. A benefit of the Mango architecture is that; as a model-driven approach, no significant changes are required to be made in order to adapt legacy systems for mobile-cloud optimisation. Thus, it is development efficient. Also Mosaic is performance efficient, in the sense that it validates the architectures of the application prior to deployment to ensure that the use of

Mango architecture within an application will most certainly yield benefits – this is achieved by the Quality Verifier.

Finally, an evaluation of the Mosaic framework was conducted, to investigate its overhead in development. Compared to existing counterpart the library is more efficient – incurring lesser development/build time, compared to the added 233-344% increased overhead of the counterpart.

Chapter 6. Beftigre Evaluation Approach

An Approach for MCA Evaluation and Comparison

6.1 Introduction

This section presents a framework – known as Beftigre[17], for evaluating mobile cloud applications. Beftigre stands for *behaviour-driven full-tier green evaluation*. The novelty of the approach is to use formalised software engineering concepts/methodology to assist in testing and comparing between offloading schemes applied in MCAs. Although Beftigre is presented in the Mango approach, it can also be used for evaluating existing MCA and their offloading schemes. Unlike existing evaluation approach adopted in the literature, Beftigre can present a full-tier consistent result for the MCA offloading scheme being evaluated.

Methodology:

Behaviour-driven: Behaviour-driven development (BDD) is a design approach to aid collaboration between non-technical contributors (such as business analysts, or users) and software engineers. Consequently, BDD gears towards more verifiable and collaborative test process by being able to compare expected behaviours with actual results, following standard simplified scenarios – constructed by simple language clauses, GIVEN, WHEN and THEN [108]. Beftigre adopts the BDD concept and simple clause approach, to simplify the comparison and evaluation of offloading schemes, and thus; simplifying software design decisions.

Full-tier: the approach adopts the concept of fine-grained software testing to present the implications of an offloading scheme on the mobile tier as well as on the cloud tier. By evaluating the system as a whole the approach can detect whether an offloading scheme is aware of both mobile and cloud resource

[17] Beftigre is pronounced /biːf ˈtaɪ.gər/ as in beef-tiger.

consumption. The full-tier objective of the approach is also assisted by the BDD concept.

Glossary of Terms:

- *Actual*: This is the test results of the application under test.
- *Expected*: This is the test results of an application with which is the basis of comparison against an application under test. These values are provided in Beftigre annotations (@Given, @When and @Then).
- *Evaluation vs. Comparison*: Evaluation involves performing a test without comparison (i.e. expected values), while Comparison involves the use of expected values (in other words, there is an expectation of the outcome).

Appendix E to H presents the implementation specific details on the Beftigre evaluation approach. Also the source code implementation of the framework and further technical guide is provided in the Beftigre documentation website[18].

6.2 The Beftigre Approach

The challenges and difficulties of the mobile-centric architecture scenario approach fall into the category of (green[19]) software testing/evaluation [23], [34]. In response to the identified challenges (see Section 3.3.1 of Related Work), the thesis proposes an evaluation approach known as Beftigre, which adopts the behaviour-driven technique to address challenge 1 and 2. Furthermore, the thesis treats challenge 3 as a testing granularity problem, consequently resolved by fine-grained testing of the mobile tier and the cloud tier (i.e. full-tier). In particular, the approach makes the following novel contributions:

[18] Beftigre documentation: http://beftigre-mca.appspot.com

[19] Green is the term used for software optimisation and testing based on the energy-efficiency and performance metrics [10], [18]. Since offloading schemes target energy-efficiency of mobile devices, similarly scenario based comparison is within the green software testing category.

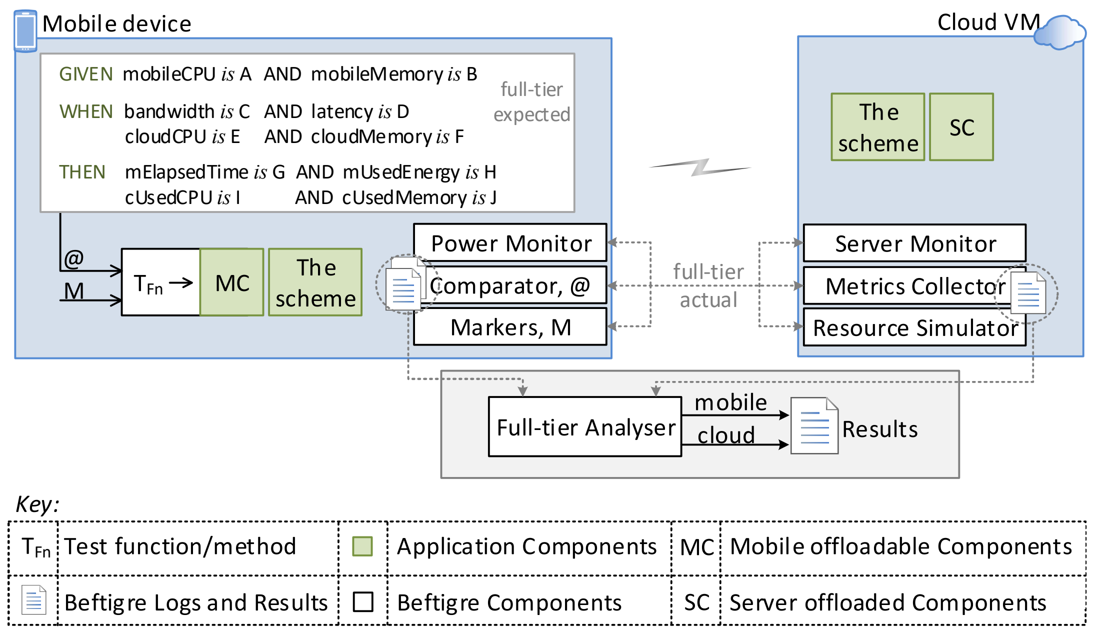

Mobile device
GIVEN mobileCPU is A AND mobileMemory is B
WHEN bandwidth is C AND latency is D
cloudCPU is E AND cloudMemory is F
THEN mElapsedTime is G AND mUsedEnergy is H
cUsedCPU is I AND cUsedMemory is J
full-tier expected
@
M
TFn
MC
The scheme
Power Monitor
Comparator, @
Markers, M
full-tier actual
Cloud VM
The scheme
SC
Server Monitor
Metrics Collector
Resource Simulator
Full-tier Analyser
mobile
cloud
Results
Key:
TFn Test function/method
Application Components
MC Mobile offloadable Components
Beftigre Logs and Results
Beftigre Components
SC Server offloaded Components

Figure 6.1 Beftigre system architecture

1) Behaviour-driven Comparison – as single scenario

Beftigre adopts BDD concept. BDD simplifies software testing by using clauses which can easily be communicated across development team to construct a scenario for testing [112]. Similarly, Beftigre's *Comparator* (Figure 6.1) uses these clauses in form of annotations to specify the expected (THEN) MCA system behaviour, given a set of conditions (GIVEN and WHEN). The annotations can be applied to a test function (TFn; which is a method to test the mobile component – as shown in Figure 6.1), and the information supplied through the annotations is then used (by Full-tier Analyser) to evaluate or compare against the actual test results. Thus, the behaviour-driven comparison process acts as a single scenario replacing varying architecture scenarios, and alleviating inconsistency of architecture scenarios; therefore a solution to challenge 1 and 2. Furthermore, the annotations are used to implement full-tier comparison (to address challenge 3) by using @Given and @When annotations to specify mobile and server *expected* preconditions respectively, for comparison on Then clauses.

Markers are objects for counter-based instrumentation which has attributes that can be written to file for processing. The purpose of Markers is to integrate instrumentation counters with annotation information (of comparator component) for efficient single scenario comparison. The start and finish counters in Markers component are vital to calculating *actual* values of THEN clause (mobile energy and time, server CPU and memory used) which are used to compare against *expected* values of @Then annotation. Thus, Marker interface contributes to the behaviour-driven solution to challenge 2 and contributes a foundation for full-tier analysis (i.e. based on THEN clauses).

2) Full-tier Analysis – results from mobile and cloud perspective

Beftigre evaluates a MCA from mobile (power usage and performance) and cloud (resource usage) perspective to produce full-tier actual test results. The analysis is accomplished by *Full-tier Analyser* (details presented later in Algorithm 1 and 2). The aim is that by full-tier analysis (as opposed to mobile-centric testing) an offloading technique can be finely evaluated in terms of its impact on the system as a whole. [34], shows that a fine-grained approach to energy measurement (using counters) can reveal specific energy usage points. Similarly, to identify specific energy points, Beftigre adopts fine-grained measuring distributed across the mobile tier (using *Markers*) and cloud tier (using *Metrics Collector*). Consequently, by revealing the mobile to cloud resource consumption implications of a MCA application, the full-tier analysis objective addresses challenge 3.

3) Unified Monitoring – for facilitation of full-tier results

The required data for full-tier analysis is collected by *Power Monitor* and *Markers* for mobile energy and performance respectively; by *Metrics Collector* for cloud resource usage; and by *Resource Simulator* for determining cloud resource availability and network. By unified monitoring of mobile and cloud tiers, the aforementioned components contribute to the full-tier evaluation. Specifically, these components (Figure 2) contributes to the research objectives as follows;

Power Monitor is achieved by PowerTutor model via an API for seamless integration with the BDD objective of Beftigre. The power monitor generates

PowerLog, which (in conjunction with counters from Marker component) is used to compute *actual* mobile energy usage which is used to compare against the expected used energy attribute of @Then annotation. This is a step which contributes to the mobile-tier of the full-tier evaluation, using behaviour-driven concept.

Server Monitor provides the logic for monitoring and serving the cloud tier metrics useful for the behaviour driven full-tier evaluation. It uses *PerfMon Server Agent* to compute metrics for *actual* cloud-tier values of Then clause (i.e. percentage CPU and memory usage), and socket server programs to compute metrics for *actual* values of Where clause (i.e. available percentage CPU and memory).

Resource Simulator makes it possible to evaluate the MCA scheme based on different set environmental conditions. Furthermore, it provides the capability for introducing controlled rigour (i.e. experimental replication) to the evaluation process (as shown later in section 5). In the implementation section, the simulator is provisioned with the server monitor.

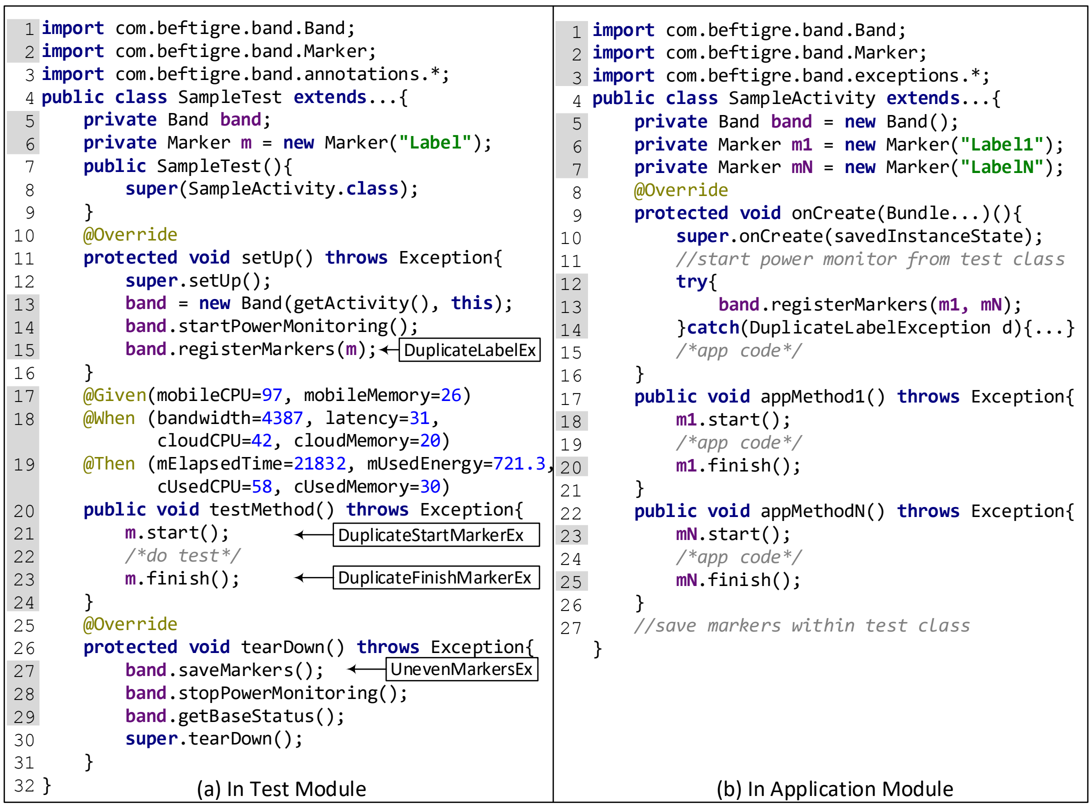
Metrics Collector provides the logic (client programs) for saving the cloud-tier metrics received from the server monitor component. The metrics collector persists the *actual* values for When clause and cloud-tier values of Then clause. By generating evaluation data for cloud-tier, the metrics collector contributes to the full-tier solution to challenge 3.

The aforementioned components focus on monitoring: power and performance for the mobile tier, and resource usage for cloud tier, because they are the popularly investigated green metrics for MCA domain [4], [7], [46], [113], [114]. Mobile energy is increasingly gaining research interest due to the resource-constrained nature of mobile devices and the increasing demand for rich mobile applications. Resource usage (specifically CPU and memory) is a commonly investigated metric when monitoring workload impact on the cloud [46], [115]. In the current Beftigre approach, the scope of the energy measurement is at the application level, i.e. the overall energy consumption of the mobile execution. The overall mobile energy is measured using the power tutor model and applies to all network types (including WLAN and 3G).

6.3 Design Details

6.3.1 Behaviour-driven Comparison

Behaviour-driven comparison is achieved by the Comparator interface. The comparator is implemented as a method named getBaseStatus() which calculates *actual* mobile CPU and memory availability, consequently; contributing to the single scenario and behaviour-driven objective, by providing *actual* values for Given clause. It is made as the last API call (as seen on Line 29 Figure 6.2a), to ensure the process does not add any overhead to the power and performance readings; furthermore, it is designed as an android alarm monitor service, to ensure the process completes even after the test is terminated (i.e. tearDown() method on line 30).



```
 1 import com.beftigre.band.Band;
 2 import com.beftigre.band.Marker;
 3 import com.beftigre.band.annotations.*;
 4 public class SampleTest extends...{
 5     private Band band;
 6     private Marker m = new Marker("Label");
 7     public SampleTest(){
 8         super(SampleActivity.class);
 9     }
10     @Override
11     protected void setUp() throws Exception{
12         super.setUp();
13         band = new Band(getActivity(), this);
14         band.startPowerMonitoring();
15         band.registerMarkers(m);
16     }
17     @Given(mobileCPU=97, mobileMemory=26)
18     @When (bandwidth=4387, latency=31,
              cloudCPU=42, cloudMemory=20)
19     @Then (mElapsedTime=21832, mUsedEnergy=721.3,
              cUsedCPU=58, cUsedMemory=30)
20     public void testMethod() throws Exception{
21         m.start();
22         /*do test*/
23         m.finish();
24     }
25     @Override
26     protected void tearDown() throws Exception{
27         band.saveMarkers();
28         band.stopPowerMonitoring();
29         band.getBaseStatus();
30         super.tearDown();
31     }
32 }
```

(a) In Test Module

```
 1 import com.beftigre.band.Band;
 2 import com.beftigre.band.Marker;
 3 import com.beftigre.band.exceptions.*;
 4 public class SampleActivity extends...{
 5     private Band band = new Band();
 6     private Marker m1 = new Marker("Label1");
 7     private Marker mN = new Marker("LabelN");
 8     @Override
 9     protected void onCreate(Bundle...)(){
10         super.onCreate(savedInstanceState);
11         //start power monitor from test class
12         try{
13             band.registerMarkers(m1, mN);
14         }catch(DuplicateLabelException d){...}
15         /*app code*/
16     }
17     public void appMethod1() throws Exception{
18         m1.start();
19         /*app code*/
20         m1.finish();
21     }
22     public void appMethodN() throws Exception{
23         mN.start();
24         /*app code*/
25         mN.finish();
26     }
27     //save markers within test class
   }
```

(b) In Application Module

Figure 6.2 BAND Template

The annotations: @Given, @When, and @Then are the basis on which offloading schemes are evaluated or compared. @Given annotation specifies the *expected* percentage CPU and memory availability of the mobile device, both of which are integer typed. Recall, the *actual* value for Given clause is obtained from getBaseStatus(). @When annotation specifies the *expected* bandwidth (bps), latency (ms), and server percentage CPU and memory availability, all of which are integer typed. The *actual* value for the When clause is obtained from the server monitors. @Then annotation specifies the *expected* mobile elapsed time (ms) and used energy (mJ), and cloud percentage used CPU and memory, during the period of the test. All parameters are integer typed except used energy which is double typed.

@Given is a precondition for the mobile end, while @When is the precondition for the server end. @Then annotation is the postcondition with elements specifying values to be asserted or compared against. Both the pre and post conditions are based on the full-tier concept (i.e. mobile and cloud involved).

6.3.2 Platform Monitoring and Control

Full-tier Platform Monitoring is achieved by power monitor and server monitor interfaces for the mobile and cloud tiers respectively.

6.3.2.1 Power Monitor

Power monitoring in Beftigre is achieved using the PowerTutor [50] model. Although the PowerTutor monitor is widely adopted in the research for mobile power monitoring, it is worth noting that the PowerTutor seems to produce accurate energy measurement for specific brands and models of mobile devices and rough estimates for others [50]. However, as exact measurements may not be necessary for relative comparisons, this thesis adopts the widely used PowerTutor model, leveraging the core logic of the monitor for Beftigre API. Furthermore, two methods are exposed to start and stop the monitor. In Beftigre API, the PowerTutor[20] based monitor is not a UI-based application but a service which runs at the background to monitor the application power usage. startPowerMonitoring() launches PowerTutor, while stopPowerMonitoring() stops and saves power data to PowerLog.

The purpose of adapting the monitor as an API is for seamless integration with the mobile test package; consequently, allowing for ease of control of the monitoring process from the code. Furthermore, as a background process, the monitor does not interfere with the application being tested. As shown in lines 14 and 28 of Figure 6.2a, the monitor is started and stopped right before and after the call to register and save marker objects within setup and teardown methods of the Android test framework, respectively. This is to ensure that the monitor captures all test execution process.

6.3.2.2 Server Monitor and Simulator

The server monitor interface is used to orchestrate three monitoring processes and two resource simulation processes on the server. For server monitoring: *PerfMon Server Agent* monitors the percentage CPU and memory usage. *CPUMemoryAvail* computes percentage CPU and memory availability using

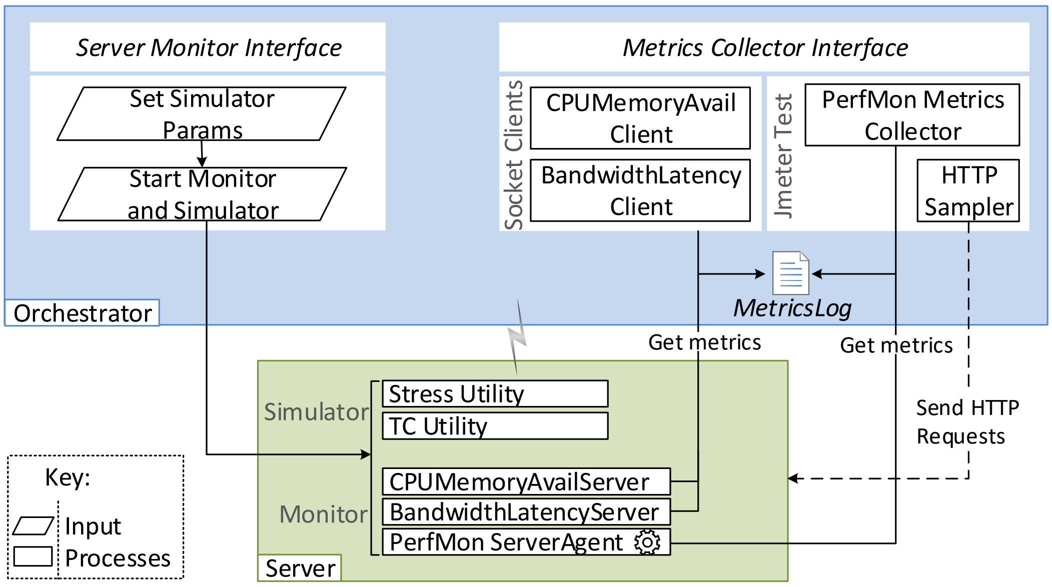


Figure 6.3 Logical functions of Server Monitor and Metrics Collector

[20] PowerTutor; https://github.com/msg555/PowerTutor [01-Jul-2016].

SIGAR API[21]. And *BandwidthLatency* is used for obtaining bandwidth and latency. The last two monitors are java socket server programs. For resource simulation: the *Traffic Control Utility* is used to simulate different network conditions (using the parameters: bandwidth and latency)[22], the *Stress Utility* is used to simulate CPU and memory load (using the parameters: CPU load and memory load). TC and Stress are both Linux utilities, and along with SIGAR API, they are popularly adopted for server resource monitoring and simulations. To start the monitoring the aforementioned simulation parameters are passed to the Server Monitor Interface (as shown in Figure 6.3).

6.3.3 Metrics Collection

Full-tier Metrics Collection is achieved by Marker and Metrics Collector interfaces for the mobile and cloud tiers respectively.

6.3.3.1 Marker

Markers are objects for counter-based instrumentation (line 6, Figure 6.2a) which has attributes that can be written to file for processing. A marker object takes a *label* attribute as a parameter which is used to associate counters with test functions. After markers have been created, registerMarkers() is used to validate and assign a unique identifier to all marker objects to be used in the test, while saveMarkers() is used to save and write the attributes of registered Marker objects to MarkerLog. A marker object (Mn, n is an integer typed unique identifier) consists of the following attributes: Mn_start, Mn_finish, Mn_label and Mn_anno. Mn_start and Mn_finish are references to the timestamps in ms used to identify the execution block for the test section. Mn_label is the reference to the label assigned during marker creation. And

[21] SIGAR API; https://support.hyperic.com/display/SIGAR [01-Jul-2016].
[22] Slow: simulates low bandwidth, high-latency; https://gist.github.com/obscurerichard/3740206 [01-Jul-2016].

Mn_anno is a reference to the annotations assigned to the test function (i.e. lines 17 to 19, Figure 6.2a).

Beftigre carries out four types of exception handling associated with Marker objects: DuplicateLabelException is thrown if two or more markers were registered with the same label. Labels are used to create a unique identifier for markers, no two markers can have the same label. DuplicateStartMarkerException and DuplicateFinishMarkerException are thrown from the Marker class. The former is thrown when the API identifies that start() was called more than once on a registered marker while the latter is thrown when finish() is called more than once on a registered marker. UnevenMarkerException is caused if a marker set was incomplete. For example when a marker is started by calling the start() method, and not finished. All registered markers must have a complete set. A marker is said to have a complete set if it calls a start() and a finish() method.

With the Beftigre API, one test method can be created within a test project to test the overall application – this includes only a marker object and annotations (see Figure 6.2a). Multiple markers can be applied within the application project; however, annotations are not applicable in this scope (see Figure 6.2b) because the annotations of the main test method (in Figure 6.2a) covers any markers used within the application project (in Figure 6.2b). Furthermore, service-based API features (such as start and stop power monitoring, and base status service) are only declared once from the test projects, as the call from test project captures the application module execution.

6.3.3.2 Metrics Collector

Following the start of server monitors and simulators, is the launch of metrics collectors. The Metrics Collector Interface is composed of three metrics collector processes within the orchestrator (as shown in Figure 6.3). *CPUMemoryAvailClient* records the percentage CPU and memory availability obtained from *CPUMemoryAvail* monitor. *BandwidthLatencyClient* records bandwidth and latency, which is measured by sending packets to and from the *BandwidthLatency* monitor. *CPUMemoryAvailClient* and *BandwidthLatencyClient* collectors execute only once (i.e. when metrics collectors are launched) and then terminates. *PerfMon Metrics Collector* runs continually for a scheduled time, recording server CPU and memory usage at intervals using Apache JMeter binaries. Furthermore, *PerfMon Metrics Collector* is launched as a listener, based on HTTP request sampling, within JMeter test plan. The test plan implements a loop controller to ensure that metrics recording process is continuous until a scheduled duration is elapsed. The duration is any speculated time, in seconds, which covers the evaluation process.

Network and resource availability metrics from Socket clients are first logged into MetricsLog, followed by the Resource usage metrics from *PerfMon Metrics Collector*. The purpose of using *PerfMon Metrics Collector* for continuous recording of resource usage at intervals is so that the average percentage resource usage (of the MCA scheme being evaluated) can be computed after the test is done. See Algorithm 6.1 for further details.

6.3.4 Full-tier Analysis and Control

Full-tier analysis and control is achieved by the Full-tier Analyser interface. Completing the test on the mobile and server tier generates three logs: MarkerLog and PowerLog from the mobile; and MetricsLog from the server. The full-tier analyser interface (analyser for short) analyses data stored in the logs both from mobile and cloud tier to produce full-tier results. The analyser also references the behaviour-driven annotation from the test for comparison.

Algorithm 6.1 *Evaluate* function, to produce actual values of Then clause

```
Require: $T_S$ and $T_F$, $P_S$ and $P_F$, $C_S$ and $C_F$, $M_S$ and $M_F$, which represent start and finish
range for mobile time, mobile power, cloud CPU usage and cloud memory usage values
from Map.

1: a_mElapsedTime ← $T_F - T_S$
2: a_mUsedEnergy ← (sum( $P_S : P_F$ ) / count( $P_S : P_F$ )) × a_mElapsedTime
3: a_cUsedCPU     ← sum( $C_S : C_F$ ) / count( $C_S : C_F$ )
3: a_cUsedMemory ← sum( $M_S : M_F$ ) / count( $M_S : M_F$ )
```

Algorithm 6.2 *Assert* function, to produce assertion of comparison

```
Require:  actual values of Then clause from Evaluate.
inRange(a, b)          : return true; if a is in ± 5% range of b
lessOrMore(a)          : return a . "% more"; if a is positive else a . "% less"
isCongruent(a, b, c, d) : return true; if any three of a, b, c, d is within ± 1%
/* e_... are expected values, a_... are actual values */

 1: Set variables for all the expected values from annotation attributes:
        e_mobileCPU, e_mobileMemory,
        e_cloudCPU, e_cloudMemory, e_bandwidth, e_latency,
        e_mElapsedTime, e_mUsedEnergy, e_cUsedCPU, e_cUsedMemory
 2: Set variables for the actual values of Given and When clauses:
        a_mobileCPU, a_mobileMemory,
        a_cloudCPU, a_cloudMemory, a_bandwidth, a_latency
 3: if (inRange(a_mobileCPU, e_mobileCPU) & inRange(a_mobileMemory, e_mobileMemory)
      & inRange(a_cloudCPU, e_cloudCPU) & inRange(a_cloudMemory, e_cloudMemory)
      & inRange(a_bandwidth, e_bandwidth) & inRange(a_latency, e_latency))
    then
 4:     assert_mElapsedTime ← (a_mElapsedTime – e_mElapsedTime) × 100 / e_mElapsedTime
 5:     assert_mUsedEnergy  ← (a_mUsedEnergy – e_mUsedEnergy)  × 100 / e_mUsedEnergy
 6:     assert_cUsedCPU     ← (a_cUsedCPU – cUsedCPU) × 100 / e_cUsedCPU
 7:     assert_cUsedMemory  ← (a_cUsedMemory – e_cUsedMemory) × 100 / e_cUsedMemory
 8:     result ← lessOrMore(assert_mElapsedTime) . lessOrMore(assert_mUsedEnergy) .
                 lessOrMore(assert_cUsedCPU) . lessOrMore(assert_cUsedMemory)
 9:     if (isCongruent(assert_mElapsedTime, assert_mUsedEnergy,
            assert_cUsedCPU, assert_cUsedMemory))
        then
10:         "The compared systems are similar"
11:     end if
12: end if
```

The analyser contributes the final solution to the challenges in Section 3.3.1 by consolidating the contributions of all the Beftigre components.

Algorithms 6.1 and 6.2 presents the *Evaluate* and *Assert* functions of the full-tier analyser respectively, used to compute the results of the MCA test.

Evaluate presents the algorithm useful for evaluating an offloading scheme (self-evaluation) which gives a full-tier result (mobile performance and energy, with cloud CPU and memory used) that is, the *actual* values of Then clause. *Assert* presents the algorithm useful for comparing between offloading schemes. *Assert* extends the *Evaluate* function in order to determine the relationship between the schemes using the behaviour driven annotations, by asserting which scheme is more efficient based on the given and where conditions or whether the expected and actual values are from the same/similar system/offloading scheme. The functions which make up the analyser are presented below in the order of execution;

- *ExtractExpected* extracts the expected value of all attributes of annotations from MarkerLog.
- *ExtractActual* extracts the actual (or measured) values of Given and When clause, i.e. mobile CPU and memory availability from PowerLog; and cloud CPU and memory availability, bandwidth and latency from MetricsLog (used in line 2 of Algorithm 6.2).
- *Map* obtains the start timestamp (TS) and finish timestamp (TF) from MarkerLog, and matches them to that of the PowerLog and MetricsLog in order to obtain the exact mobile power (PS to PF), cloud CPU (CS to CF) and cloud memory (MS to MF) used by the MCA during the evaluation.
- *Evaluate* (Algorithm 6.1) firstly computes the elapsed time (ms), used energy (mJ), average CPU usage (%) and average memory usage (%) using the data from *Map* function.
- *Assert* (Algorithm 6.2) compares Then clause *actual* values from *Evaluate* function with the @Then annotation *expected* values from *ExtractExpected* function to assert a result following some condition. *Assert* is achieved in two stages below;

  Stage 1 Assertion:

  Result – The assertion result (lines 4-8, Algorithm 6.2) gives the percentage increase or decrease[23] between expected scheme and actual scheme at a full-tier scale (i.e. mobile elapsed time, mobile used energy, cloud CPU usage and cloud memory usage).

[23] Percentage increase (in actual value compared to expected value) means increased or more resource demand, while percentage decrease means decreased or less resource demand.

Condition – The assertion is performed if the Given and When values (from *Evaluate* and annotation) are within ± 5% range[24] (meaning schemes are comparable).

Final Assertion:

Result – asserts that the expected and actual schemes are of the same or similar system (lines 9-11, Algorithm 6.2).

Condition – the assertion is true if any three values of Then clauses of *Stage 1 Assertion* are within ± 1%. In other words if after comparing *actual* and *expected* of Then clause, and any three of its attributes (i.e. any three of mobile time, mobile energy, cloud CPU and cloud memory) has percentage increase or decrease within the range of ± 1%, then the *actual* and *expected* schemes are similar or the same.

After the above analysis process is completed, a CSV file containing the computed values/summary of analysis is generated for reporting purpose.

6.4 Performance Evaluation

The performance evaluation was performed for BAND API on Windows 10 x64 PC, with Intel i7 2.20GHz CPU and 8GB memory. Furthermore, the mean value of 30 test samples (on local execution) was used to investigate the performance (overhead) of the API on mobile testing by comparing the build, setup and execution time of *Default* test setup against *Beftigre* setup (Figure 6.4). The *Default* setup, is the conventional android test setup with Robotium API for UI test, while *Beftigre* setup adds the BAND API to the default setup.

Build time; measured using Android Studio's build functionality; is the time to (re)build the modules and libraries of the project. Setup and execution time are measured using Java timestamp utility. Setup time is the time it takes to

[24] Existing experimental research e.g. [117], uses 5% range as acceptable range of comparison between expected and actual power readings (for power meters vs power models, respectively). ±5% range is therefore applied to green metrics in this thesis.

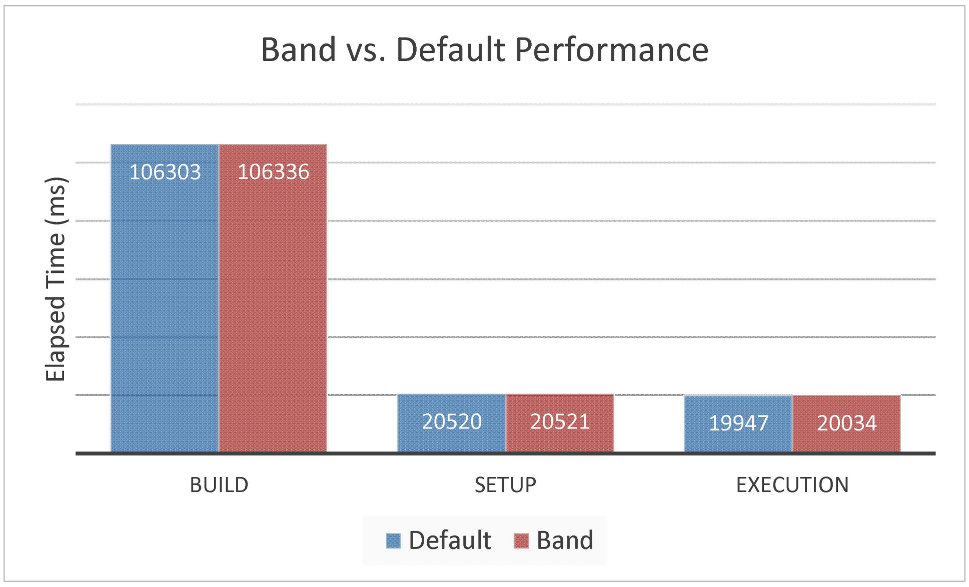


Figure 6.4 BAND API Performance Evaluation

initialise all test library objects used (i.e. Lines 11-16 of Figure 3a). Execution time is the time from setup to test completion.

From Figure 6.4, less than 0.3% increase is observed in the execution time, which implies no overhead is caused by the test API to the application build time. This is because the libraries which contribute to the build-time overhead are those used within the application module (and not the test module), such as Mosaic library – presented in Chapter 5. Also, as the power monitor runs as a different android service (and as a different process) to the test process, its execution has no significant effect on the test; this is similar to executing PowerTutor app external to the application under test. Consequently, the results of the evaluation show that BAND API has no significant overhead to the android test process. And also since the API integrates with the test process/project, the API does not interfere with the actual execution of the application under test.

6.5 Summary

This chapter presented the Beftigre evaluation approach as a solution to the challenges of mobile-centric architecture scenarios approach to MCA evaluation – challenges presented in the Methodology (Chapter 3) in details. The objective of the approach is i) full-tier evaluation: which is to achieve

Table 6.1 Summary of Beftigre components (of Band and Befor APIs)

Clauses	***Metrics***	***Source Beftigre Component/Interface***		
		Expected	***Actual***	***Both***
Given	• Mobile % CPU availability • Mobile % Memory availability	Comparator's Annotations	Comparator's BaseService	Full-tier Analyser
When	• Bandwidth (bps) • Latency (s) • Cloud % CPU availability • Cloud % Memory availability	Comparator's Annotations	Socket Clients, with Socket Servers.	
Then	• Mobile Elapsed Time (ms) • Mobile Used Energy (mJ)	Comparator's Annotations	Marker Power Monitor	Full-tier Analyser
	• Cloud % CPU usage • Cloud % Memory usage		Perfmon Metrics Collector, with Server Agent	

evaluation at a fine granularity for MCA by taking the metrics of both mobile and cloud tiers into consideration, ii) comparability: which makes it possible to evaluate MCA comparing between counterpart techniques, and iii) reproducibility: which makes it possible to repeat tests for a given MCA and arrive at the same conclusion. This is achieved by controlling the environmental factors of MCA. Control is administered through the server simulator and full-tier analyser.

Table 6.1 presents a summary of the Beftigre framework and the coordination between the two APIs. From Table 6.1 the preconditions (Given and When) and post conditions (Then) of the test process, are full-tier, in other words, they span through the mobile and cloud tier. Furthermore, at each tier, logs are generated to provide data for computing metrics that reveal the implications of a test process at each tier. The evaluation and demonstration of effectiveness for Beftigre approach has been achieved using real world applications and presented in the Case Studies (Chapter 7).

Chapter 7. Case Studies

Using Real-life Applications to Critically Analyse *Mango*

7.1 Introduction

This chapter uses four different android applications as case studies to evaluate the proposed Mango approach. Table 7.1 describes the applications and their functionalities. These applications are chosen based on their unique characteristics/taxonomy: Linpack[25] and NQueen[26] are computation intensive, MatCalc[27] and MathDroid[28] are data intensive. The use of a case study with a sample size of four taxonomy unique applications in the thesis is because it is the popularly adopted sample size and sampling technique for MCA experiments in the literature, e.g. [7], [8], [52]. Furthermore, these applications have also been evaluated by previous studies [7], [8], [52]. Details of the selection criteria are presented in Appendix B.

The objectives of the experiments in this section are as follows:

- To evaluate the effectiveness of the Mosaic framework in identifying Callees based on Quality Verification (phase 3 of Mango approach).

Table 7.1 Characteristics of the case studies

Application	**Offloading candidates**	**CI**	**DI**	**Description (Android apps)**
Linpack	*run()* method of *Linpack* class	✓		Linear algebra benchmark app
MatCalc	*times(Matrix B)* method of *Matrix* class		✓	Matrix calculator app
MathDroid	*computeAnswer(String query)* method of *Mathdroid* class		✓	Calculator app
NQueen	*nQueenCount(int input)* method of *NQueen* class	✓		NQueen computation app

Key: CI – computation intensive, DI – data intensive.

[25] Linpack: https://github.com/pedja1/Linpack
[26] NQueen: https://github.com/acelan/NQueen
[27] MatCalc: https://github.com/kc1212/matcalc
[28] MathDroid: https://code.google.com/archive/p/enh/source

Recall that quality verification determines whether an identified offloadable candidates (Callees) will most certainly yield benefits when adopting the Mango architectural approach. (See Section 7.5 for evaluation).

- To evaluate the proposed Mango architecture in terms of the earlier stated benefits: suboptimal awareness, variability (context) awareness, and full-tier awareness for SQAs. Recall that SQAs considered in Mango are: performance and energy-efficiency for the mobile, and resource-efficiency and availability for the cloud. (See Section 7.6 for evaluation).
- To evaluate the proposed Beftigre approach in terms of the earlier stated benefits: full-tier effectiveness, robustness of test, reproducibility of test. (See Section 7.7 for evaluation).

A benefit of Mango architecture is that; as a model-driven approach, no significant changes are required to be made in order to adapt legacy systems for mobile-cloud optimisation (*the model-based framework, Mosaic aids with the adaptation*). In this chapter, the required changes made to the base code – in order to make it suitable for Mango architecture – have been presented in the 'Legacy Adaptation' section (7.4).

7.2 Experimental Settings

7.2.1 Experimental Variables

Table 7.2 presents the experimental variables used with the case studies.

Table 7.2 Experimental Variables

Dependent	Independent	Control (factors)
- *Green metrics:*	- *Architecture scenarios:*	Mobile CPU availability
Mobile Performance	Local	Mobile memory availability
Mobile Energy usage	Optimal	Cloud CPU availability
Cloud CPU usage	Offloading scheme/Architecture	Cloud memory availability
Cloud Memory usage		Bandwidth
- *Other quality attributes:*		Latency
Application availability		Size of transferred data

Table 7.3 Selection of Independent Variables

Summary	**Techniques (and distinct scenarios used)**		
	POMAC	**TB-CP**	**DPartner**
Local	OnDevice	Smart phone only	Phone Offload(Local)
Server	OnServer	Offload(All)	Offload(All)
Optimal	Optimal	Offload(only Assessed)	Offload(only Monitored) diff. RTT
Offloading-scheme	POMAC	Offload(+adapted) Offload w/Threshold	On-Demand Offload

The dependent variables are the events being studied, and expected to change whenever the independent variable is altered. As shown in Table 7.2, the events studied are green metrics for mobile and cloud tiers. Also, quality attributes are investigated.

The independent variables: for the experiment are Local, Optimal and Offloading architecture scenarios. Table 7.3 presents the criteria for selecting the independent variables – which is based on recurrent scenarios in the literature (i.e. the summary column of Table 7.3). Beftigre framework is a MCA evaluation (and comparison) approach contributed/proposed by this research as an improvement on the architecture scenario evaluation approach.

Following an investigation into the scenarios used in the evaluation of the selected offloading techniques, a summary is presented in Table 7.3. The summary column uses a common term to present the recurrent scenarios, as follows:

- Local: the original app runs entirely on the phone [4], [7], [52], [116]. [52] also specifies Offload-Local which is a scenario where the optimised application runs entirely on the phone. In the experiment, Local is used since it is recurrent of the two.
- Server: all identified offloadable components are executed on the cloud [4], [7], [52].

- Optimal: only offloadable components that are assessed as computation intensive (or data intensive) are executed on the cloud [4], [7], [52].
- Offloading scheme: is the scenario comprising the proposed offloading scheme [4], [7], [52], [116], e.g. POMAC, TB-CP, DPartner, (and in this research; MANGO).

The control variables: also has an effect on the dependent variables, so they are also monitored in the system. For the experiment, these include bandwidth, latency and size of data being transmitted. Although the control variables are naturally challenging to control, the Beftigre framework presents a technique to effectively monitor these variables for a comparison process. The technique is behaviour-driven and uses a full-tier evaluation technique – details presented in Chapter 6. Note that: some or all (depending on the scheme) of the control variables are adopted in offloading schemes to predict optimal behaviour or make offload decisions.

7.2.2 Metrics, Tools & Platform

As a green software research, the key metrics being measured in this research are Green metrics for mobile and cloud tier. These are energy usage and elapsed time (performance) for mobile, and resource usage (i.e. CPU and memory usage) for cloud (see Table 7.4). The case studies evaluate the approach based on three green metrics – energy usage, resource usage and performance.

Table 7.4 Metrics, Tools & Platforms for Case Studies

Green metrics	Measuring Tools	Domain/Platform
Energy usage	Power Tutor model – computes mobile power usage.	Mobile: Android Samsung galaxy S3
Performance	Java timestamp utility – computes total execution time.	
Resource usage	PerfMon Server Agent & Metrics Collector – computes % CPU and memory usage.	Cloud: Amazon EC2 Ubuntu instance

The mobile device used is Samsung Galaxy S3 Neo running Android 4.4.2 (KitKat) on Quad-core 1.4 GHz, with 1.5GB memory. While the cloud configuration is an Amazon EC2 m3 instance running Linux Ubuntu 14.04 64 bit, with Intel Ivy Bridge 2.5GHz CPU and 3.75GB memory. Furthermore, Java JDK 1.8 was used for implementing the case studies – mobile and cloud tier. The minimum SDK version was set as 15 for the android applications, with compile SDK from version 22 upwards. The selected case studies have been tested on the new Android Studio 2.0, thus; the contributions and results of this research are up-to-date with current development tools and consequently relevant for current software practice.

Power Tutor was used for measuring the power usage of the mobile application. It is chosen because it is the most popular power model adopted in the literature [4], [7], [52] – and has also been used in evaluating the selected offloading schemes. Energy-efficiency (EE) metric [34] was gotten as a derived metric by computing energy usage based on the power logs of power tutor. Similarly, resource efficiency was determined from the CPU and memory usage logs from the server. These logs are obtained through PerfMon ServerAgent and PerfMon Metrics Collector (more on the evaluation tools presented in Chapter 6). Furthermore, the measuring tools presented in Table 7.4 are all open sourced and also components of the Beftigre framework presented in Chapter 6.

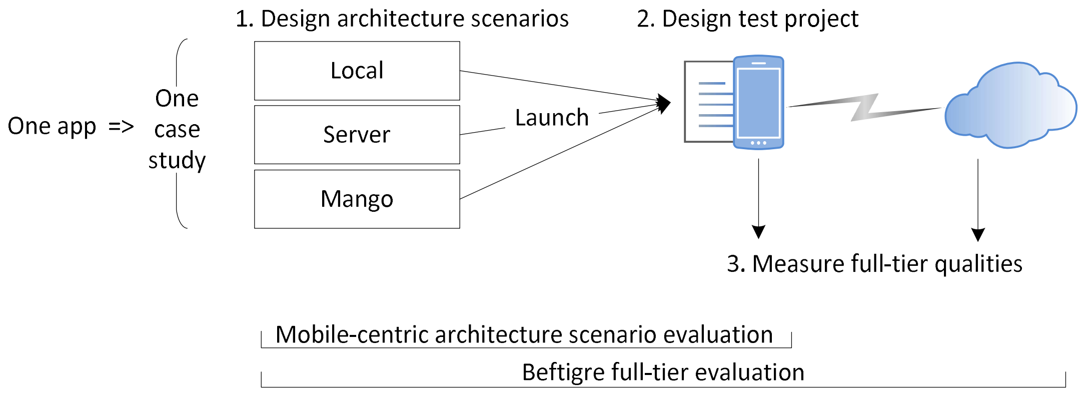

1. Design architecture scenarios
2. Design test project
One app =>
One case study
Local
Server
Mango
Launch
3. Measure full-tier qualities
Mobile-centric architecture scenario evaluation
Beftigre full-tier evaluation

Figure 7.1 Experimental Process

7.2.3 Experimental Process

The experiment on each case study (Table 7.1), combined the architecture scenario evaluation with Beftigre full-tier evaluation for full-tier qualities, involved three key tasks/processes. As shown in Figure 7.1 these are; 1) designing scenarios, 2) designing and launching tests, and 3) measuring full-tier qualities; i.e. metrics for both the mobile and cloud tier.

Firstly, for each case study three architecture scenarios (i.e. the independent variables – Local, Server & Mango) are designed. Based on these architecture scenarios in combination with the applications used in the literature, Mango approach is compared against counterpart techniques. Secondly, an android test project is designed for each case study, which is irrespective of the architecture scenario. In other words, the same test is used for all architecture scenarios of a case study app. And thirdly, green metrics are measured for the application under test. For the mobile-centric architecture scenario evaluation approach, the metrics measured are mobile energy used and mobile elapsed time. In the Beftigre evaluation option, only the offloading scheme is evaluated, other architecture scenarios are not necessary. The test is then annotated with Beftigre annotations for comparison or evaluation, then the green metrics are collected based on a full-tier evaluation (i.e. mobile and cloud tier). The mobile tier metrics being elapsed time and used energy, and the cloud tier metrics being used CPU and memory, as presented in Table 7.2.

Two adaptive optimisation techniques have been selected to evaluate the Beftigre evaluation approach. The techniques of comparison are POMAC (a machine learning approach) [7], [8] and TB-CP (a threshold-based checkpointing approach) [4].

7.3 Test Classes

In order to accurately measure the effect of Mango architectural approach to MCA optimisation on the application, the test is written to capture the exact execution of the offloadable (CI) task – i.e. the Callee. This is achieved by implementing the finish marker (of the Band API) immediately after the Callee is completed.

To execute the finish marker immediately after the Callee is completed entails capturing a *predictable* or *set* Callee output – using Robotium[29] solo.waitForText(String text) rather than solo.waitForActivity(String activity). The difference is that the former returns true immediately a passed text is found on display, whereas the latter returns true after all processes in a passed activity is completed – thus incurring more waiting time. (Appendix I presents the Test Classes for the case studies and screenshots of test output).

- **A Predictable Callee Output**

The predictable Callee output is the result of the Callee that is known before execution and thus can be passed to solo.waitForText(String text) prior to the test execution. The test assertions for MatCalc, MathDroid and NQueen case studies are based on the results of the Callee, as they are predictable outputs (see Appendix I.2, I.3 and I.4 respectively, for the classes).

The mathematical results for Callee of MatCalc and MathDroid (given a set of inputs) can be predicted/obtained without execution. For example;

MatCalc app has its Callee as the matrix multiplication task. Therefore given two matrix A and B then the result A.B can be predicted as shown below.

If Matrix A is $\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix}$ and Matrix B is $\begin{matrix} 0.5 \\ 2 \\ 8 \end{matrix}$ Then A.B = $\begin{matrix} 28.5 \\ 60 \\ 19.5 \end{matrix}$

Thus the test assertion is implemented based on the predicted results (i.e. A.B above), as; solo.waitForText("28.5\n60\n19.5").

MathDroid app has its Callee as the number multiplication task. Therefore given two numbers 3 and 7, the product is 21. Thus the test assertion is implemented based on the product as; solo.waitForText("21").

[29] Robotium [119] is an Android UI testing API.

NQueen app has its Callee as the counter functionality of game – which gives the total number of possible queens in an x square game. Given an input (x) of 14, the total number of possible queens are 365596. Thus the test assertion is implemented based on the expected result as; solo.waitForText("365596").

- **A Set Callee Output**

The set Callee Output is a notification string used to indicate when the Callee execution is completed. This is achieved by implementing an Android toast with the notification string, after the Callee implementation, as shown below;

```
Toast.makeText(getApplicationContext(), "Callee completed.", Toast.LENGTH_SHORT).show();
```

This approach is used for the case studies comprising Callee outputs that cannot be clearly predicted prior to execution, e.g. Linpack (see Appendix I.1 for the test class). Within the test class, the set Callee output is referenced as the notification string, as; solo.waitForText("Callee completed.").

7.4 Legacy Adaptation

As shown in Table 7.5, code scaffolding was achieved in two ways; Mosaic automated refactoring, and manual refactoring. The Mosaic automated refactoring is the generation of the ACTS classes by the Mosaic framework –

Table 7.5 Legacy Adaptation

Place holders /refactor	**Linpack**	**MatCalc**	**MathDroid**	**NQueen**
Mosaic automated refactoring				
[Callee]	`…MainActivity .runLinpack`	`…MainActivity .times`	`…Mathdroid .computeAnswer`	`…NQueen .nQueenCount`
[Arguments]	`Class clazz`	`Matrix A, Matrix b`	`String query`	`int input`
[References]	`clazz`	`A, b`	`query`	`input`
[Return]	`Result`	`Matrix`	`Node`	`Integer`
Manual refactoring /implementations				
Activity	`MainActivity .activity`	`MainActivity .activity`	`Mathdroid .activity`	`NQueen.activity`
Serialisation	Required	- (In Legacy)	- (In Legacy)	-
Cloud tier	Required	Required	Required	Required

which replaces all placeholders (in templates) with actual values (from the call-graph model). The manual refactoring entails further changes and implementation that was required to fully adapt the application to the Mango architecture (or as a MCA application – e.g. Cloud tier implementation).

Furthermore, the source code screenshots used for explanation of the legacy adaptation process are obtained only for Linpack application, for the purpose of demonstration. However, the snippets of the ACTS classes for the four case studies are presented in Appendix J.

7.4.1 Mosaic automated refactoring

The highlighted segments in Figure 7.3 and Figure 7.2 shows the adaptation on the Aspect and Task templates made by Mosaic, to generate the Aspect and Task classes, respectively. To demonstrate Mosaic automated refactoring the code screenshot have only been presented for Aspect and Task classes as they show all Mosaic placeholders.

The Mosaic call-graph, mcg (Table 7.6) is the model for the generation of the actual values of the Mosaic placeholders. This is the identified offloadable callee for MCA optimisation. As noted in the Mosaic framework (Chapter 5), the mcg (Table 7.6) specifies the superclass of the Caller, the Caller, the Callee, its return type and its argument types – which are relevant for model transformation into ACTS classes.

Table 7.6 Mosaic Call-graph (mcg)

For Linpack

`android.app.Activity  rs.pedjaapps.Linpack.MainActivity:run`
`rs.pedjaapps.Linpack.MainActivity:runLinpack  rs.pedjaapps.Linpack.Result  java.lang.Class`

For MatCalc

`android.app.Activity  com.android.matcalc.MainActivity:customTimes`
`Jama.Matrix:times  Jama.Matrix  Jama.Matrix`

For MathDroid

`android.app.Activity  org.jessies.mathdroid.Mathdroid:exe`
`org.jessies.mathdroid.Mathdroid:computeAnswer  org.jessies.calc.Node  java.lang.String`

For NQueen

`android.app.Activity  com.mango.queens.NQueen:computeNQueen`
`com.mango.queens.NQueen:nQueenCount  int  int`

```
 1 package mango;
 2
 3 import ...
 4
 5 @org.aspectj.lang.annotation.Aspect
 6 public class Aspect {
 7
 8     @Pointcut("call(* rs.pedjaapps.Linpack.MainActivity.runLinpack(..)) && args(arg_0)")
 9     public static void offloadMethod(Class arg_0) {
10     }
11
12     @Around("offloadMethod(arg_0) && !within(Aspect) && !within(Task)")
13     public Result aroundOffloadMethodCall(ProceedingJoinPoint jp, Class arg_0) throws Throwable {
14         return new Task().execute(new Object[]{arg_0}).get();
15     }
16 }
```

Figure 7.3 Aspect Class for Linpack

As shown in Table 7.5 and Figure 7.3 (Line 8) Mosaic replaces the [Callee] placeholder with appropriate rs.pedjaapps.Linpack.MainActivity.runLinpack from the mcg (Table 7.6). Similarly the [Arguments], [ArgumentIDs] and [Return] placeholders are transformed to appropriate values as shown in Figure 7.3, Aspect class (Lines 11 and 15 for [Arguments] as `Class arg_0`, Lines 10, 14 and 16 for [ArgumentIDs] as `arg_0` and Line 15 for [Return] as `Result`). This transformation is also achieved for other ACTS classes.

For example, for the Task class the [Callee] and [CastedArguments] placeholders for referencing the offloadable method has been transformed

```
 1 package mango;
 2
 3 import ...
 4 import rs.pedjaapps.Linpack.MainActivity;
 5 import rs.pedjaapps.Linpack.Result;
 6
 7 public class Task extends AsyncTask<Object, Integer, Result> {
 8     private static Result result      = null;
 9     ...
10     private static final int TIMEOUT  = 5000;
11     private static final int OVERHEAD = 2542; //25% OVERHEAD
12
13     @Override
14     protected Result doInBackground(final Object[] params) {
15         ...
16     }
17
18     private void runOnMobile(Object[] params) {
19         result = MainActivity.runLinpack((Class)params[0]);
20     }
21
22     private void runOnCloud(Object[] params) {
23         try {
24             Socket socket = new Socket("46.137.91.122", 3);
25             ...
26             result = (Result) inputStream.readObject(); //read
27             ...
28         } ...
29     }
30     ...
31 }
```

Figure 7.2 Task Class for Linpack

appropriately as shown in Line 19 of Figure 7.2, with [CastedArguments] replaced by `(Class)params[0]`. Furthermore, the [Return] placeholder which holds the return type of the Callee is replaced by `Result` object as shown in Lines 7, 8, 14 and 26 of Figure 7.2 Task class.

Also as shown in Table 7.5, the Mosaic transformation was also performed for MatCalc, MathDroid and NQueen applications. Table 7.5 shows that, like Linpack (used for demonstration), Mosaic also transformed placeholders for these applications into appropriate ACTS classes with actual values derived from mcg (Table 7.6).

7.4.2 Exposing and Referencing an Activity

The Context class (of ACTS) requires an activity to create shared preferences used in the decision-making at the mobile tier. The activity to be referenced by the Context class is therefore needed to be exposed from an android activity class – by initialising an Android activity as a public static variable. The initialisation of the exposed activity static variable is performed within the onCreate(…) method of the Android activity, to ensure that the activity (static variable) is initialised immediately the Android activity is created. To ensure that the variable is always set when the application is launched the main/launcher activity of the application is used to initialise the static activity variable used by Context class (i.e. the launcher activity is the activity that starts the app).

```java
package mango;
import ...

public class MainActivity extends Activity implements Runnable {
    ...
    public static Activity activity;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        activity = MainActivity.this;

        ...
    }
}
```

Figure 7.4 Launcher Activity of Linpack

```
1 package mango;
2 import ...;
3 import rs.pedjaapps.Linpack.MainActivity;
4
5 public class Context {
6     private static Activity activity = MainActivity.activity;
7     ...
8 }
```

Figure 7.5 Context Class for Linpack

Exposing and referencing the activity are achieved manually, as shown by the highlighted sections of Figure 7.4 and Figure 7.5. Exposing the activity requires two lines of code as shown in Figure 7.4 – which involves declaring the static variable of Activity type (Line 6) and initialising the variable with the launcher activity (Line 13). Whereas referencing the activity only requires two lines of code as shown in Figure 7.5 – which involves importing the activity (Line 3) and calling the activity (Line 6).

Furthermore, as shown in Table 7.5, activities are also manually exposed and referenced for the MatCalc, MathDroid and NQueen applications; with their respective launcher activities as MainActivity, Mathdroid and NQueen classes respectively.

7.4.3 Serialising Objects

By convention, as with distributed systems, objects passed across the network, have to be serializable. In Java, this is achieved by implementing java.io.Serializable interface (to allow communication across the network). As shown in Table 7.5, objects are required for the Callees of Linpack, MatCalc and MathDroid applications.

For the Linpack application, to execute the Callee (i.e. runLinpack method), java.lang.Class is required as an argument – which is a generic Java class. However, the execution of the Callee returns a Result object – which means that the result class is required to be serialised in order to execute the Linpack Callee remotely. The legacy Result class does not implement Serializable interface, therefore it was refactored by implementing java Serializable interface to allow for the object to be transmitted across the network.

Table 7.7 Cloud tier Callee dependencies

Apps	Cloud tier Callee dependencies	
	Libraries	**Classes**
Linpack	-	rs.pedjaapps.Linpack.Linpack.java (Callee) rs.pedjaapps.Linpack.Result.java
MatCalc	Jama-1.0.2.jar (Callee)	com.android.matcalc.MatrixParser.java
MathDroid	Calc.jar	org.jessies.mathdroid.Mathdroid.java (Callee)
NQueen	-	com.queens.nQueenLib.java (Callee)

Key: (Callee) signifies the location of the implementation of the Callee method.

For the MatCalc application, to execute the Callee (i.e. times method), Matrix objects are required as arguments, moreover, the Callee also returns a Matrix object. Thus the Matrix class has to be serializable in order to allow for remote execution of MatCalc's identified Callee. As indicated in Table 7.5, the legacy MatCalc application already defines the Matrix class as serializable, thus refactoring was not necessary. Similarly, the Node class of legacy Mathdroid application was already defined as serializable and was not refactored.

7.4.4 Implementing the Cloud tier

Unlike the mobile tier which already consists of the implementation of the Callee, the cloud tier always requires the Callee to be implemented (as shown in Table 7.5). The implementation of Callee on the cloud tier is not achieved by Mosaic; as there may be libraries (Table 7.7) required by the Callee during execution. Thus Callees are implemented manually on the cloud tier and integrated or called within the generated Service class of ACTS pattern.

Table 7.7 shows the libraries and classes required for implementing the Callee on the cloud tier. For MatCalc, the identified offloadable Callee (i.e. `times` method from Table 7.1) is within the library, whereas that of other case study apps are within classes. All the aforementioned dependencies presented in Table 7.7 are deployed in the cloud alongside their respective Service class (of ACTS pattern), which together make up the cloud tier of the Mango MCA architecture. Mosaic placeholders found within the Service template are transformed by Mosaic while transforming other ACTS components – as presented earlier. Moreover, the CPU threshold and Time threshold of the Service class can be set manually.

For the case studies; CPU threshold was set as 30% – an arbitrary value to signify the minimum CPU availability of the cloud to process any request. Time threshold was set to correspond to the approximate tolerable local execution time to the nearest 1000ms. For example, the average local execution time of Linpack is 4733ms, the Time threshold at the cloud tier is set to 5000ms. The screenshot for the code base of the Service classes for the case studies are presented in Appendix J.

7.5 Critical Analysis of Mosaic Approach

Table 7.1 presents the initially identified offloadable candidates for each of the applications. The initially identified offloadable candidates are based on the selective analysis presented in Algorithm 5.1. To enhance the initial identification, inclusion rules were applied for Callees based on the identified offloadable candidates in the literature [8], [52] – thus Table 7.1 outcome.

To further determine the efficacy to yield offload benefits, Algorithm 5.3 (Profiler Aspect) is applied to the initially identified Callees of Table 7.1. The result of Profiler Aspect is presented in Figure 7.6, based on 10 experiments.

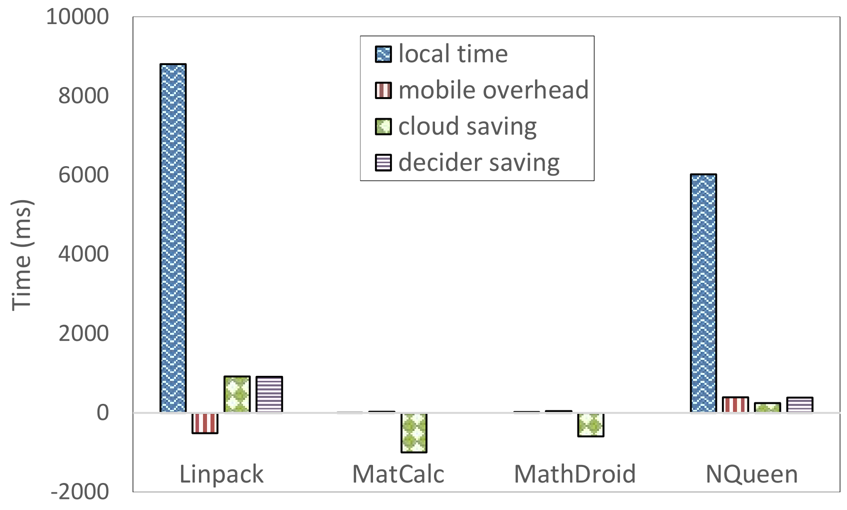


Figure 7.6 Mosaic (Profiler Aspect) evaluation of offload candidates

Mobile overhead: Recall that *mobile overhead* compares `Mango` mobile scenario to the local execution scenario. The results show that `Mango` does not contribute significant overhead when executing the Callee on the mobile, as shown in MatCalc, and Mathdroid. The mobile overhead for Linpack yields a negative value due to differences in environmental states, naturally, this is not supposed to be so. However, environmental states are unpredictable, thus comprising outliers. Such outliers show that `Mango` mobile scenario in Linpack comprised of more favourable environmental factors compared to the local scenario. The reverse being the case for NQueen.

Cloud saving: Recall that *cloud saving* compares `Mango` cloud scenario with the local execution scenario, and negative value of *cloud saving* means no offload benefits, and therefore not recommended for MCA optimisation. From the results of Fig. 8 the Profiler Aspect evaluation shows that offloading the offload candidates for MatCalc and MathDroid does not yield benefits, with MatCalc having a significant lower bound – significantly inefficient. Hassan et al. [8] also evaluates MathDroid and MatCalc based on the identified offloadable candidates in Table 7.1, and these applications are found to not yield offload benefits after optimisation. The literature asserts that this is due to the application being data intensive. Moreover the evaluation in the related work[8] does not focus on the Callee, but rather the entire optimised application – thus the results of the evaluation (in [8]) seem to portray a closeness between local execution and MCA optimisation (and a possibility for savings) – i.e. for MatCalc and MathDroid, however, this is not the case as shown by the negative cloud saving of the aforementioned apps in Fig. 8. Using Profiler Aspect provides a finer granularity of evaluation by using pointcuts to point to the exact call to the Callee and using *before* and *after* advices to measure the exact execution time of the Callee of concern.

In the `Mango` evaluation section, the `Mango` architecture was applied to MatCalc and MathDroid, to demonstrate that the evaluation of the application as a whole is significantly different from the Callee evaluation (fulfilled by Aspect Profiler). This sheds light on the reason why the local (or on-device) execution of app is interpreted as being close to the MCA optimised scenario by related work [8].

7.6 Critical Analysis of Mango Architecture

In this section, the proposed architecture is evaluated by comparing three scenarios; Local, Server and `Mango` scenarios. The local scenario executes the Callee on the mobile device, the server scenario executes the Callee remotely in the cloud, while the `Mango` scenario executes using the proposed architecture. For the server and `Mango` scenarios, 10 experiments are conducted by simulating fog settings (which has higher bandwidth and low latency) and cloud settings (which is based on adverse network conditions). In *Fog* setting (denoted by **F**, in the graph) the following was used: 100Mbps bandwidth, 20ms latency, 2 CPU and 2 memory worker threads. In *Cloud* setting (denoted by **C**, in the graph) the following was used: 50kbps bandwidth, 1s latency, 6 CPU and 4 memory worker threads. Note: the worker threads are used (by *Stress Utility*) to stress the server CPU and memory.

The results of the experiments (and `Mango` benefits/efficiencies) are discussed based on observed behaviours – classified in three sections; suboptimal awareness, variability awareness and full-tier awareness.

7.6.1 Suboptimal awareness

Recall from the Profiler Aspect evaluation, that the MatCalc and MathDroid applications do not yield offloading benefits. Similarly, as shown in Figure 7.7 the server scenario (which executes the Callee on the server) is significantly inefficient (both performance and energy wise) compared to local execution. `Mango` architecture captures the aforementioned concern during decision-making and does not offload subsequent executions of the Callee – thus avoiding the overhead of offloading without actual benefits. Consequently, compared to the server scenario, `Mango` is more efficient. However, in comparison to the local scenario, some overhead – although not significant (5.33%) – is incurred by the decision-making process. Thus reemphasizing

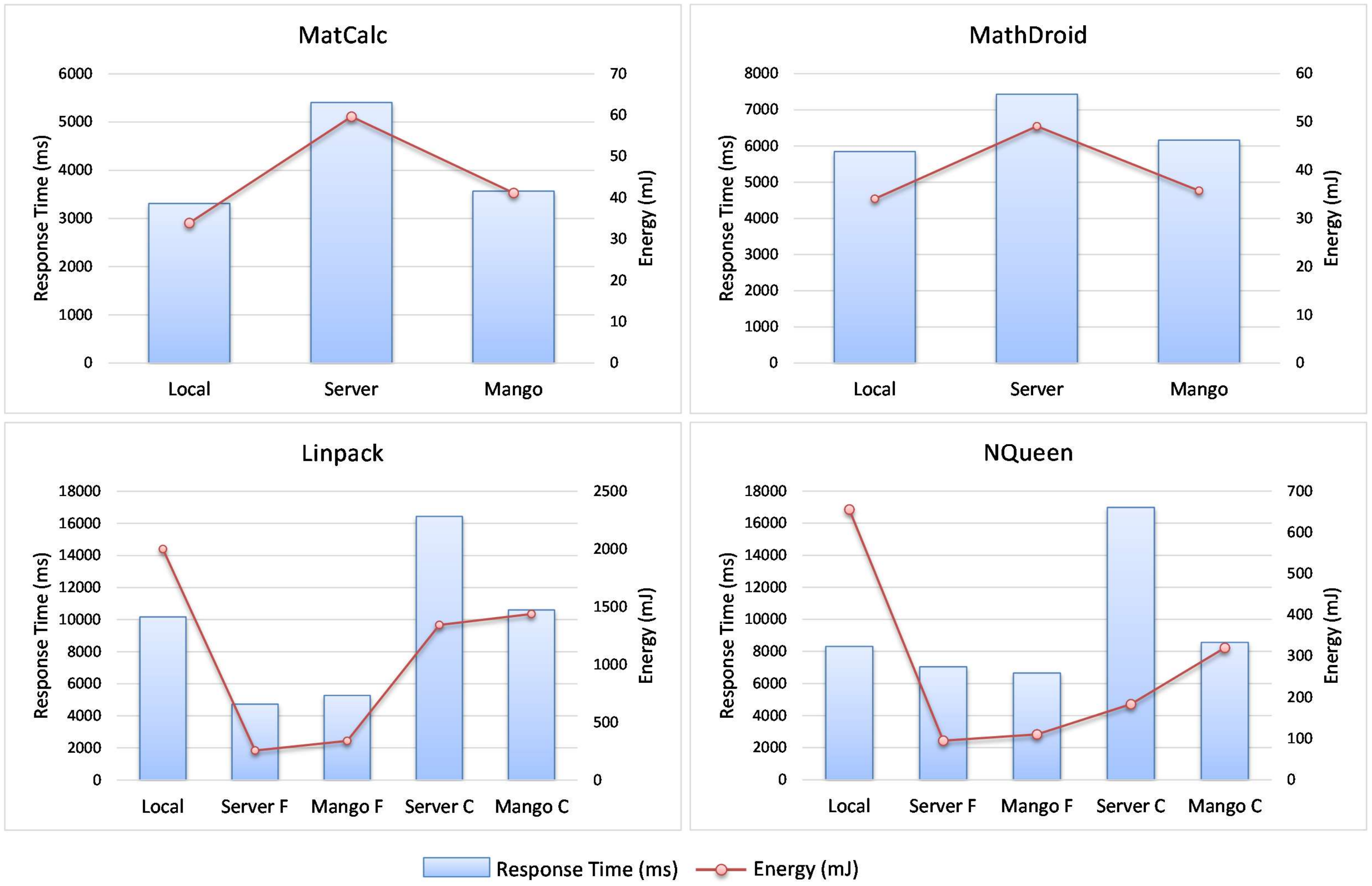


Figure 7.7 Performance and Energy Results of the Mobile tier

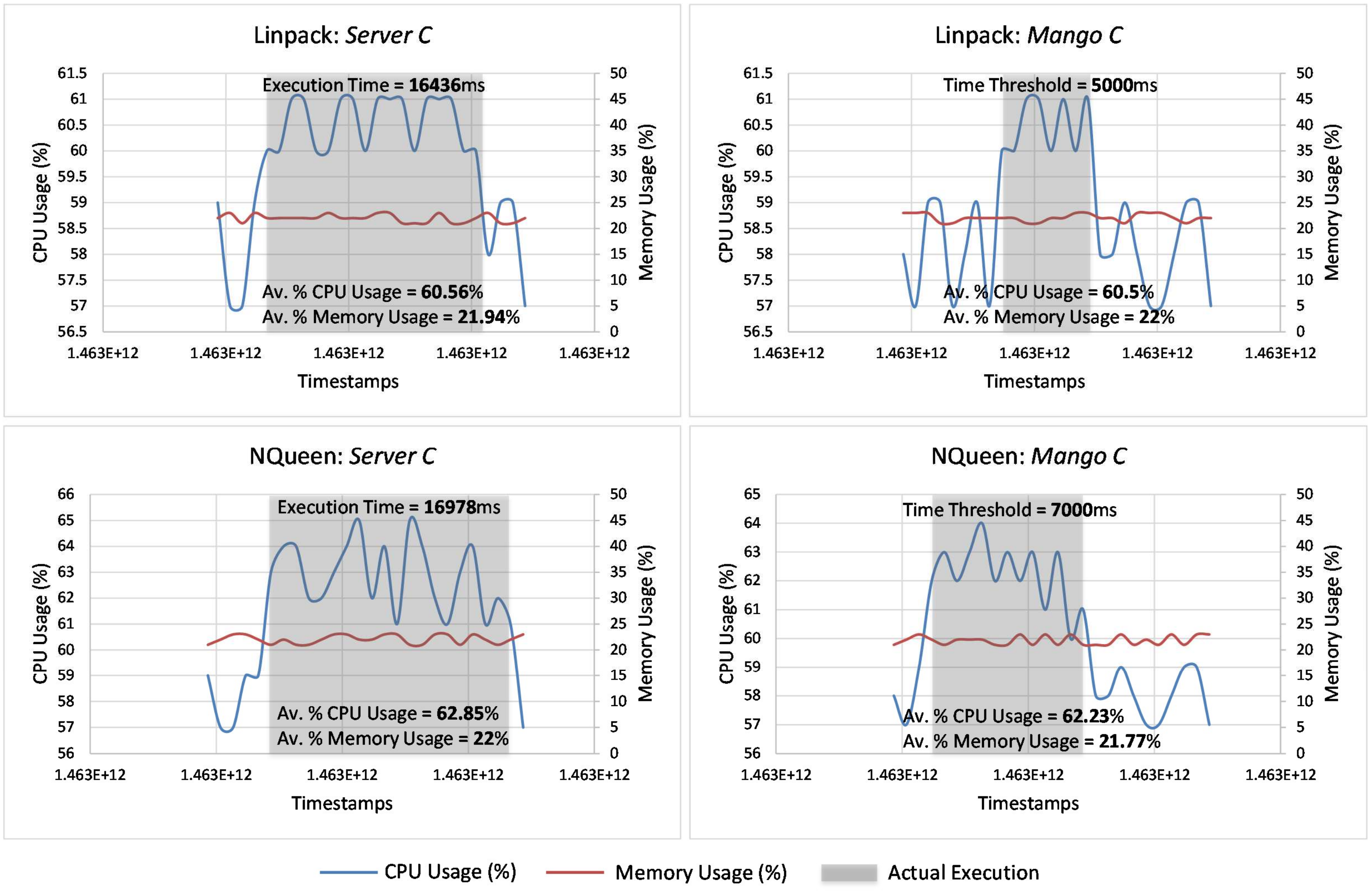


Figure 7.8 Cloud tier Results for Resource Efficiency and Availability

the importance of the fine-grained evaluation by Profiler Aspect (Quality Verifier) at the earlier stages of identification of offloading candidates – a proof of hypothesis H1.

Hypothesis H1 in Chapter 3 presented that *Offloading any task which compromises the condition where combined overhead of all MCA components is always lesser than local, will always compromise performance, even if the remote execution time of offloadable component is less than that of local.* Suboptimal awareness refers to the capability of the Mango approach in avoiding such situations where offloading does not yield benefits. The Mosaic framework (of Mango approach) provides a Selective Analyser which selects offloadable components by applying the rules in the rules repository. The rules repository is composed of both exclusion and inclusion rules that are applied during the process of identification of offloadable tasks. Such suboptimal tasks are added to the rules repository for exclusion during identification process. This means that the final transformed MCA will exclude such tasks not yielding offloadable benefits (in other words, completely avoiding the 5.33% overhead mentioned above), and thus avoiding the overhead resulting in improper identification of offloadable tasks (a solution to Problem I of Chapter 3).

7.6.2 Variability awareness

Variability awareness refers to the capability of the Mango architecture to adapt (or make decisions) in varying environmental conditions with minimal overhead, whether normal or adverse conditions, in order to achieve software target qualities. Figure 7.7 presents the result for favourable conditions as F (i.e. fog settings), and for adverse conditions (i.e. adverse cloud settings).

For Linpack and NQueen offloading, (*Server F*) yields performance and energy benefits, this is also achieved by *Mango* (*F*), see Figure 7.7. However, as shown by the case studies, always offloading in adverse conditions (*Server C*) may compromise performance, `Mango` (*Mango C*) is aware of such variability in environmental conditions – achieving at least 30% performance improvement in adverse conditions compared to the always offloading scenario. And also achieving energy and performance improvements (from 10-30% performance improvement on normal cloud conditions, and from 72% energy savings for normal cloud conditions – all compared to local execution scenario). Conclusively, `Mango` approach is aware of both favourable and adverse environmental conditions to achieve mobile energy and performance efficiency. Thus, time based context-aware decision making in mango is used to achieve accuracy in decision making, without the overhead of multiple parameter monitoring (solution to Problem II of Chapter 3).

7.6.3 Full-tier awareness

Figure 7.8 presents the resource usage results for adverse cloud conditions (i.e. *Server C* and *Mango C*). Furthermore, the results of Figure 7.8 are focused on Linpack and NQueen, as it has been established in earlier sections (6.1 and 6.2.1) that MatCalc and MathDroid does not meet the effective offloading criteria and that as a consequence; offloading to the cloud would result in a mobile performance overhead, as well as unnecessary cloud resource usage.

To investigate the cloud resource usage (Figure 7.8), the CPU and memory are monitored. The highlighted section of the graphs (in Figure 7.8) shows the execution of the Callee on the cloud. Recall from the `Mango` algorithm that *Time Threshold* is applied in the cloud and mobile tier (Algorithms 4.2 and 4.3 respectively) as a way to ensure that the application's performance is not compromised by adverse environmental conditions of offloading. Consequently, in adverse cloud conditions, a better performance is achieved at the mobile tier by using the proposed architecture (*Mango C*) compared to always offloading (*Server C*), as shown in Figure 7.7, and mentioned earlier in 6.2.2. Simultaneously, at the server, the *Time Threshold* achieves resource savings during adverse conditions, with `Mango` (*Mango C*) compared to always offloading (*Server C*) as shown in Figure 7.8. For example with Linpack, the elapsed cloud tier execution time for the Server scenario is approximately 16.4s, however, this is cut down to 5s with the *Time Threshold* of Mango, and for both scenarios the resource usage is approximately 60%

and 22% for CPU and memory respectively. Thus; the CPU was in more busy states for the *Server C* than *Mango C* for an extra 11s time period; similarly with the Memory. Thus `Mango` (*Mango C*) is more resource efficient for cloud in adverse conditions. Note: the *Time Threshold* in `Mango` was set by using the approximate average response time of the Server in normal condition.

Conclusively, `Mango` approach is aware of both the mobile tier and the cloud tier in achieving software qualities (for improving the efficiency of the application) – a justification for hypothesis H2.

7.7 Critical Analysis of Beftigre Approach

To evaluate Beftigre approach, it is compared against the conventional Non-BDD approach using the case study applications. Thus the focal scenarios are;

- *Beftigre*: the proposed behaviour-driven full-tier approach, and
- *Non-BDD*: the mobile-centric architecture scenario approach (using Local, Server and Scheme scenarios) – see Chapter 3 for details.

Sample Compared Schemes:

For each approach (Beftigre and Non-BDD) two offloading schemes are compared:

- *Scheme1* is based on threshold-based policy, similar to [4].

In threshold-based offloading scheme [4], a method is offloaded only when its parameter data size is greater than a predefined threshold value. The scheme in [4] is implemented based on runtime checkpointing which incurs a transmission overhead due to varying offload data size, hence the use of data size for thresholding. The experiment omits runtime checkpointing, thus making the offload data size fixed for all the applications used in the experiment. Consequently, a predefined threshold is used based on the network rather than data size. Therefore, the *criteria for offloading* in Scheme1 is when bandwidth is greater than 500bps and latency is greater than 150ms.

The bandwidth and latency values are obtained by sending packets to and from the server. The premise behind the effectiveness of static thresholds – such as Scheme1 – is that the local execution time of the application for the threshold used is always greater than the remote execution time; therefore, using the threshold would amount for time or energy savings [4]. Within the *Simulation Parameters* section, the WLAN offload favourable condition simulates an environment favourable for the static threshold of Scheme1.

- *Scheme2* is based on perceptron algorithm, similar to [7].

Scheme2 uses multiple criteria for offloading – based on learned data (as opposed to a predefined static threshold). *Offload criteria* are bandwidth, latency, server and mobile CPU and memory availabilities. The adapted perceptron algorithm used in the experiment is open sourced[30]. To extract learning data for Scheme2, a special version of the application is implemented, which has the offloadable component execute remotely (LearnRemote) and locally (LearnLocal). LearnRemote and LearnLocal are instrumented with random simulation parameters (including WLAN, Outlier and 3G) to generate the following metrics; mobile CPU and memory available, server CPU and memory available, bandwidth, latency, and elapsed time. The generated data is then used to build the training dataset classified for *remote* or *local* execution. A data subset is classified as a *remote* data if the remote execution time is greater than local execution time, otherwise, it is classified as *local* data.

Simulation Parameters:

Stress and TC utility (also utilised by Beftigre's Resource Simulator) are used to provide parameters which simulate different environmental conditions to test the schemes. The simulation parameters presented below are used to maintain the same level of rigour for both Beftigre and Non-BDD approaches, and also used to discuss the results (see Table 7.8 and Table 7.9).

[30] Perceptron Algorithm in Java; https://github.com/nsadawi/perceptron [01-Jul-2016].

- *WLAN*: consists of 30mbps bandwidth, 20ms latency, 2 CPU worker loads and 2 memory worker loads. With these parameters, local execution time is greater than remote execution time, which is appropriate for offload.
- *Outlier*: consists of 20mbps bandwidth, 200ms latency, 2 CPU worker loads and 2 memory worker loads. These parameters are used to verify the offloading schemes.
- *3G*: consists of 500kbps bandwidth, 200ms latency, 5 CPU worker loads and 5 memory worker loads. With these parameters, local execution time is less than remote execution time.

The settings used in WLAN, Outlier and 3G are found to be commonly used in experiments [7], [8]. The results of the experiment have been presented in the tables (Table 7.8 and Table 7.9) and figures (Figure 7.9, Figure 7.10 and Figure 7.11), and discussed in the following three sub-sections. Following the full-tier and behaviour-driven objective of Beftigre, the approach has been evaluated against the earlier described mobile-centric architecture scenario approach (Non-BDD).

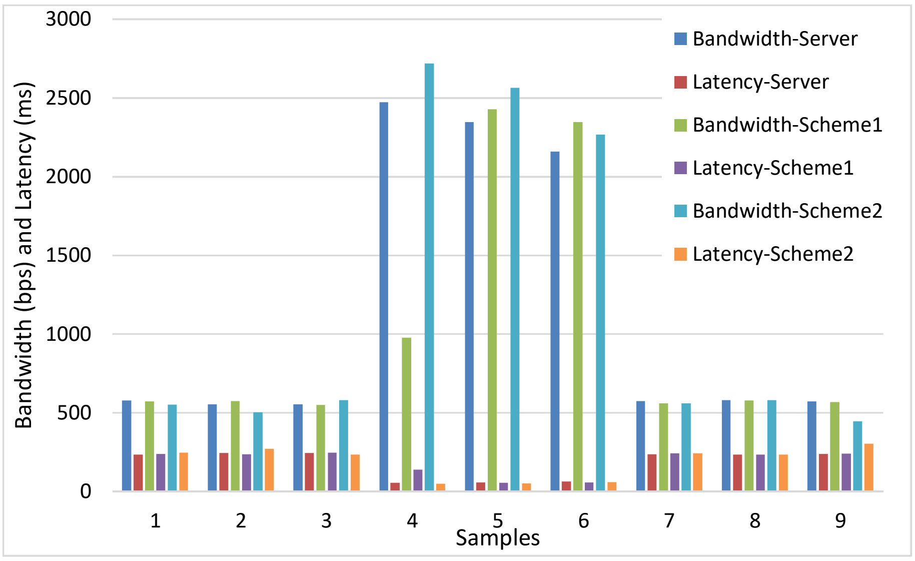


Figure 7.9 Bandwidth and Latency

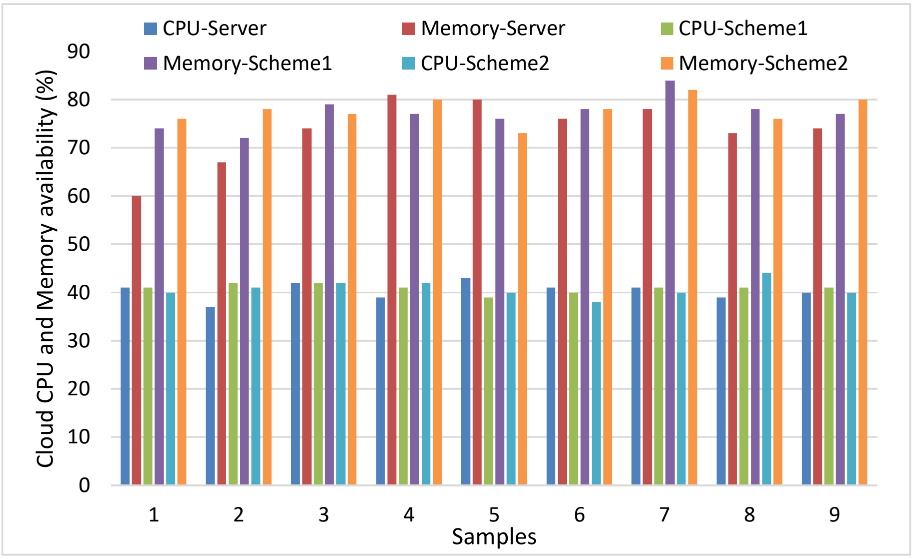


Figure 7.10 Cloud CPU and Memory availability

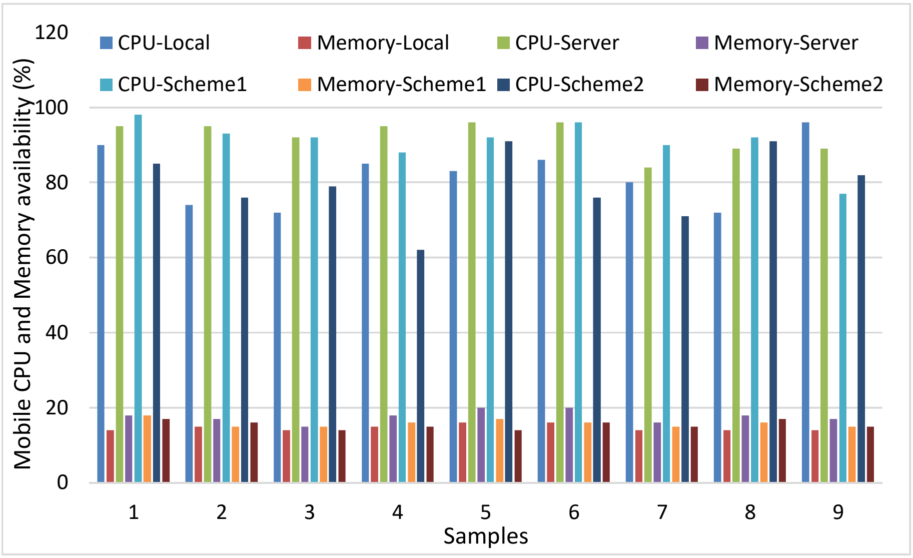


Figure 7.11 Mobile CPU and Memory availability

7.7.1 Inconsistency challenge to Non-BDD

The Non-BDD approach (Table 7.8) presents the elapsed time and used energy of the mobile device during the period of the experiment. Since the same level of rigour was applied on all experiment, sample population of 9 (3 samples from each simulation parameters – 3G, WLAN, and Outlier) was used. Table 7.8 further specifies three architecture scenarios: Local, Server and the scheme being evaluated. Local and server results are the same for both schemes and are consequently used as a basis for evaluating the schemes (using % difference), and subsequently used for comparison in the

Table 7.8 Non-BDD Evaluation and Comparison

Architecture Scenarios	**Scheme1** (3G, WLAN, Outlier)		**Scheme2** (3G and WLAN, Outlier)	
	Elapsed Time (ms)	**Used Energy (mJ)**	**Elapsed Time (ms)**	**Used Energy (mJ)**
Linpack				
Local	21952.50	3168.23	21952.50	3168.23
Server	21948.67	2539.87	21948.67	2539.87
Schemes:	21953.33	2206.08	21945.33	3102.39
Local % diff.	-0.0038	35.8055	0.0327	2.1
Server % diff.	-0.0212	14.0663	0.0152	-19.9395
MatCalc				
Local	6313.22	492.48	6313.22	492.48
Server	8340.03	711.17	8340.03	711.17
Schemes:	6403.33	517.07	7009.98	583.10
Local % diff.	-1.42	-4.87	-10.46	-16.85
Server % diff.	26.27	31.61	17.33	19.79
NQueen				
Local	17123.02	2702.13	17123.02	2702.13
Server	16980.88	2009.89	16980.88	2009.89
Schemes:	15707.00	2609.05	15689.52	1898.99
Local % diff.	8.63	3.51	8.74	34.91
Server % diff.	7.79	25.94	7.91	5.67
Note: *Local % diff.* and *Server % diff.* is the percentage difference of the scheme in comparison to Local and Server scenarios respectively. A negative value is used to signify a loss in energy savings or performance.				

Non-BDD Evaluation and Comparison above is based on mean values of samples similarly adopted by existing works [4], [7].

Non-BDD scenario. However, following the Beftigre approach to also investigate other environmental conditions, the results show inconsistencies and complex correlation between scenarios. For example, the Local scenario (which is affected by less environmental inconsistencies – only mobile CPU and memory availability – Figure 7.11), begins with its first sample as 90% and 14%, whereas Scheme1 and Scheme2 are 98% and 18%, 85% and 17% for CPU and memory availability respectively. This inconsistency follows through the samples, and also occurs in the Server scenario. Even more, the Server scenario inconsistencies are more profound as it also involves network and server resources – Figure 7.9 and Figure 7.10.

Conclusively, Non-BDD is not very effective for the comparison of schemes as each sample that make up the mean of the experiments are affected by different conditions – which are unrelated to the scheme. With Non-BDD, however, generalised conclusions can be made, such as; scheme1 is more energy efficient compared to scheme2 on the basis of local and server scenarios - this would be based on the assumption that a more rigorous

Table 7.9 Beftigre Evaluation and Comparison/Assertion

Label	**@Given: mobile**		**@When: cloud/network**				**@Then: mobile**		**@Then: cloud**		**Final Assert**
	CPU	**Mem.**	**CPU**	**Mem**	**Bandw.**	**Lat.**	**Time**	**Energy**	**CPU**	**Mem.**	
Linpack Evaluation											
Scheme1	93	18	41	67	560	243	22208	3008	58	28	-
Scheme2	81	41	41	62	557	247	21887	3181.08	60	32	-
Linpack Compare: Scheme2 expected on Scheme1 actual											
Expected	81	41	41	62	557	247	21887	3181.08	60	32	-
Actual	92	17	41	68	559	241	22304	3029.44	58	29	-
Assert	-	-	-	-	-	-	1.91% more	4.77% less	3.33% less	9.38% less	Different
Linpack Compare: Scheme1 expected on Scheme1 actual											
Expected	93	18	41	67	560	243	22208	3008	58	28	-
Actual	92	16	41	68	563	240	22233	3031	58	29	-
Assert	-	-	-	-	-	-	0.11% more	0.76% more	0%	3.57% more	Similar system
MatCalc Evaluation											
Scheme1	90	23	38	56	505	211	7033	613.41	55	26	-
Scheme2	92	22	38	56	512	224	6442	537.07	55	26	-
MatCalc Compare: Scheme2 expected on Scheme1 actual											
Expected	92	22	38	56	512	224	6442	537.07	55	26	-
Actual	94	21	38	58	507	209	7075	625.12	55	26	-
Assert	-	-	-	-	-	-	9.37% more	15.15% more	0%	0%	Different
MatCalc Compare: Scheme1 expected on Scheme1 actual											
Expected	90	23	38	56	505	211	7033	613.41	55	24	-
Actual	91	22	38	56	501	217	7085	609.91	55	24	-
Assert	-	-	-	-	-	-	0.75 more	0.57% less	0%	0%	Similar system
NQueen Evaluation											
Scheme1	84	25	43	67	552	246	16344	2810.11	59	29	-
Scheme2	89	25	42	65	540	244	15702	1908.09	59	30	-
NQueen Compare: Scheme2 expected on Scheme1 actual											
Expected	86	25	42	65	540	244	15702	1908.09	59	30	-
Actual	85	25	42	68	553	241	16289	2797.42	59	30	-
Assert	-	-	-	-	-	-	3.67% more	37.80% more	0%	0%	Different
NQueen Compare: Scheme1 expected on Scheme1 actual											
Expected	84	25	43	67	552	246	16344	2810.11	59	29	-
Actual	85	25	43	68	553	243	16302	2785.90	59	30	-
Assert	-	-	-	-	-	-	0.26% less	0.87% less	0%	3.39% more	Similar system

Full-tier Results (i.e. Then clauses) are presented above based on a comparison between Scheme1 vs. Scheme2 – Outlier simulation parameters used.

Key: *Given*: mobile CPU and memory availability (%), *When*: cloud CPU and memory availability (%), bandwidth (bps) and latency (ms), *Then*: mobile elapsed time (ms), mobile used energy (mJ), cloud used CPU (%) and cloud used memory (%).

experiment would span different environmental conditions, as 35.8mJ to 2.1J ratio is a significant saving on Scheme1. The significant saving is due to the simulation parameters used which seemed to favour Scheme1 as the robustness of Scheme2 in adverse conditions seemed to be compromised by its energy intensive decision making – especially if the adverse conditions are not extreme – further explained in the next point. Having extracted a conclusive result using Non-BDD approach, it is difficult to tell the behaviour of the scheme, for instance, given a specific (or category of) environmental condition.

Also notice that for MatCalc application which is data-intensive, Non-BDD (Table 7.9) shows Scheme1 to be 31.61% more energy-efficient compared to Server whereas Scheme2 is only 19.79% more efficient. Beftigre approach (Table 7.9) shows the reverse to be the case. This is understandable because Scheme2 is based on trained data and is aware that the application is data-intensive, and therefore stops subsequent offload. The misconception of results associated with the Non-BDD approach is due to the randomisation of samples – in which case there is more presence of favourable conditions (such as *WLAN*) than unfavourable (such as *3G*). Also with NQueen application, the same inconsistency issue reoccurs in Non-BDD (Table 7.9) where Scheme1 and Scheme2 are presented with very close results for elapsed time, i.e. 8.63% and 8.74% respectively, which is however clarified by Beftigre approach (Table 7.9). Also, there is no consistency across applications with the Non-BDD approach (Table 7.9), for example; Scheme 1 is more energy-efficient in Linpack application (using Local scenario) compared to Scheme 2, but in NQueen the reverse is the case. Beftigre, however, maintains consistency across applications.

7.7.2 Beftigre Full-tier Effectiveness

Table 7.9 presents the results of a sample evaluation and comparison using the Beftigre approach. First, the schemes are self-evaluated using the outlier simulation parameter (20mbps bandwidth, 200ms latency, 2 CPU loads and 2 memory loads), these are provided through the Orchestrator's Server Monitor Interface. Evaluation of Scheme1 gives 22208ms and 3008mJ, and Scheme2; 21887ms and 3181mJ for mobile tier green metrics (elapsed time and used energy). Furthermore, the cloud CPU and memory usage during the process are obtained – as 58% and 28% for Scheme1, 60% and 32% for Scheme2. From Table 7.9 evaluation, a prediction can be made; from the CPU and memory usage of both schemes; that Scheme1 was executed locally and did not offload whereas Scheme2 did.

Recall that in Scheme1, the criteria for offload is set to be as bandwidth >= 500bps and the latency <= 150ms, as a result, using the outlier, Scheme1 was not offloaded – thus consuming more time and energy. This means that Scheme1 is not robust enough to be aware of conditions beyond the parameters of its criteria constraints. Conversely, Scheme2 which learns from trained data is not only aware of adverse, environmental conditions but also incurs training overhead – which in this case is more energy consequential than time. The overhead is caused by the decision-making process; communication to and from the server to calculate cloud CPU and memory availability, as well as mobile resource availability and network states, prior to deciding to offload. Inspecting the full-tier analysis, it can be deduced that, based the outlier parameters Scheme2 is both inefficient in mobile and cloud tier. However, since it is aware of the environment, its benefits would be more appreciated in adverse conditions (which may include mobile devices with very low computing capacities).

7.7.3 Robustness of Beftigre Assertion

As well as deducing the behaviour of a scheme using the full-tier analysis. By adopting BDD concepts, Beftigre makes it easy to communicate expected goal of offload schemes within software teams; from business analyst to software engineers. For example, Table 7.9 also shows the comparison of the schemes based on the earlier evaluated simulation parameters. By re-executing Scheme1 changes in the actual experimental values can be observed, however, these changes are within ± 1% of the percentage increase or decrease between the expected of Scheme1 and actual of Scheme1 (to satisfy Line 9 of Algorithm 6.2). The reverse is the case when Scheme1 is compared against Scheme2, the percentage increase/decrease are beyond ± 1%.

Consequently, by developing a test plan (using simulation parameters) earlier, and a sample application, schemes can be better evaluated, compared and communicated between the development team – thus adopting the wider software engineering objective of BDD. Furthermore by adopting Beftigre, one can avoid inconsistencies, and the difficulty of variability of architecture scenarios through the use of guided annotations, while providing a finer granularity of results through full-tier analysis/evaluation.

7.7.4 Reproducibility Effectiveness of Beftigre

Table 7.10 uses the statistical method (of standard deviation) to verify the effectiveness of Beftigre for repeatability or reproducibility of its test results. The verification is achieved using: mean (on five samples for each Scheme run with *Outlier* parameters), standard deviation and 5% range criteria. Recall that in Beftigre approach, 5% range criterion is the basis on which two schemes are compared. In other words: two schemes are comparable if the preconditions (@Given and @When) of the expected scheme are within ± 5% range of the actual preconditions – as shown by *inRange(a, b)* method of Algorithm 2. The 5% range criterion is popularly explored in research [117] and in a similar context to Beftigre (i.e. for comparison). The purpose for which

Table 7.10 Replication capability of Beftigre Evaluation

	@Given: mobile		**@When: cloud/network**			
	CPU	**Mem.**	**CPU**	**Mem.**	**Bandw.**	**Lat.**
Linpack						
Scheme1 Mean	92.54	18.50	41.30	67.30	560.12	243.03
Scheme1 Deviation	1.13	0.84	0.65	1.30	11.25	5.43
Scheme1 5% range	4.63	0.93	2.03	3.37	28.01	12.15
Scheme2 Mean	84.25	42.01	41.50	63.40	558.31	247.80
Scheme2 Deviation	1.25	1.03	0.55	1.44	9.96	4.04
Scheme2 5% range	4.21	2.10	2.08	3.17	27.92	12.39
MatCalc						
Scheme1 Mean	92.91	23.20	38.06	56.85	506.45	217.00
Scheme1 Deviation	1.21	0.77	0.71	0.82	5.00	2.01
Scheme1 5% range	4.65	1.16	1.90	2.84	25.32	10.85
Scheme2 Mean	91.44	22.31	38.42	56.10	517.66	220.60
Scheme2 Deviation	2.22	0.90	0.74	1.31	4.02	6.03
Scheme2 5% range	4.57	1.12	1.92	2.81	25.88	11.03
NQueen						
Scheme1 Mean	84.60	24.84	43.44	68.70	550.98	245.31
Scheme1 Deviation	1.01	1.09	0.58	2.77	9.03	3.93
Scheme1 5% range	4.23	1.24	2.17	3.44	27.55	12.27
Scheme2 Mean	86.08	25.06	42.02	65.22	548.40	249.16
Scheme2 Deviation	2.30	1.02	0.99	2.40	7.72	3.20
Scheme2 5% range	4.30	1.25	2.10	3.26	27.42	12.91

N.B. The replication data above was verified using the Outlier parameters.

the range criteria was applied in Beftigre approach is in consideration of the unpredictable and varying nature of the MCA environment. The difficulty to predict MCA environments is a challenge which in practice contributes to the inconsistencies of existing MCA evaluation approach (as discussed in Section 3.3.1). To achieve a solution with a better consistency of results, Beftigre applies environmental control and scoping. Control is orchestrated by the Resource Simulator while the range criteria (of Full tier analyser) provides scope for comparison. Together the aforementioned features achieve reproducibility as shown in Table 7.10.

For example, from Table 7.10 varying samples have been executed using the *Outlier* parameter settings, and the deviation gives the varying range of the samples – demonstrating natural inconsistencies in MCA. However, despite the inconsistencies there are mostly overlaps in the values of the environmental metrics (i.e. CPU, memory, etc.) for the schemes (Scheme1 and Scheme2). Furthermore, the deviations are also within the ± 5% range criteria (not more or less), thus validating the scoping effectiveness of the Full-tier analyser (achieved by *inRange*) for comparison of schemes. Therefore the extent to which replication can be achieved in our evaluation approach is by using range criteria and applying an element of control to the environment.

7.8 Summary

By applying four case studies (of varying taxonomies – data and compute intensities) in the evaluation of Mango architecture this chapter has demonstrated that the architecture is effective in; achieving full-tier efficiency (i.e. savings for mobile and cloud tier), awareness of varying environmental conditions (i.e. adapting to normal and adverse conditions), and suboptimal awareness (i.e. awareness of situations where offloading does not yield benefits). Furthermore, Mosaic framework was also evaluated and proves to be effective in the identification of offloadable candidates at a finer granularity – through the use of AOP technique. By fine-grained evaluation of offload candidates, Mosaic is capable of determining offloadable candidates that can yield benefits at an earlier stage of development. Furthermore, the framework is useful for scaffolding MCA code from templates implementing the proposed ACTS design pattern of Mango, so as to automate the refactoring process in legacy systems. Furthermore, by comparing the Beftigre evaluation approach against existing evaluation approaches, it has been proven to be effective in the full-tier evaluation of MCAs. Also, the Beftigre evaluation approach presents better accuracy and granularity in the comparison between systems through the use of annotations, making it possible to use appropriate software engineering and testing concepts such as BDD (assertions) in the evaluation of MCAs.

Chapter 8. Conclusions

8.1 Introduction

The research undertaken for this thesis has enabled the development of a novel approach for green MCA software systems. The approach involves a model-driven architecture, a model-driven framework which realises the architecture, and a testing framework suitable for the architectural objectives (summarised in Figure 8.1). Together they serve to provide an efficient Mobile Cloud Application (MCA) development and testing process useful for developers; moreover providing improved mobile and cloud optimisation (or savings) useful for users and service providers. These research outcomes involve both traditional and latest theory support (such as surrounding; AOP, MDE and MCA offloading), and are backed by up-to-date technologies (such as Android Gradle compatibility for proposed frameworks, JGraphX for modelling based on XMI, Gradle Android AspectJ plugin for AOP).

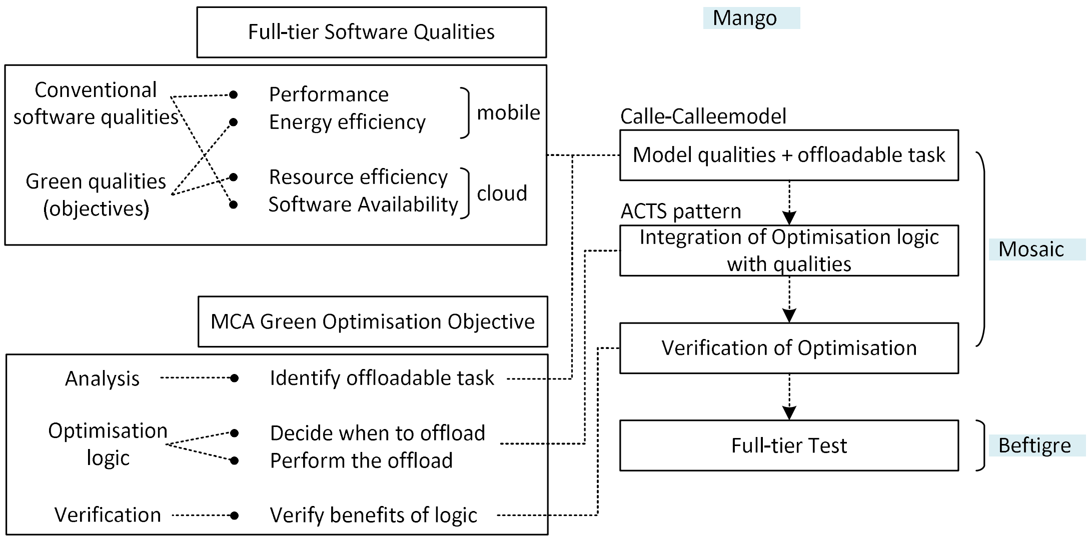

Full-tier Software Qualities
Conventional software qualities
Green qualities (objectives)
Performance
Energy efficiency
mobile
Resource efficiency
Software Availability
cloud
MCA Green Optimisation Objective
Analysis
Optimisation logic
Verification
Identify offloadable task
Decide when to offload
Perform the offload
Verify benefits of logic
Mango
Calle-Calleemodel
Model qualities + offloadable task
ACTS pattern
Integration of Optimisation logic with qualities
Verification of Optimisation
Full-tier Test
Mosaic
Beftigre

Figure 8.1 Conclusions of the Thesis

This chapter discusses the above research outcomes in terms of how well they achieve the research objectives defined previously and fulfil the different individual requirements involved. Next, the conclusions are reached and the contributions are presented. Finally, the future research directions are outlined.

8.2 Conclusions and Contributions

Despite the variety of efforts made towards mobile application optimisation using the cloud as surrogate – i.e. MCAs, the challenges of these approaches in delivering efficient solutions applicable to the development process, continues to be a major factor hindering their adoption in MCA development. Most specifically these existing approaches are challenged by optimisation overhead, development inefficiency, overall inefficiency in qualities and inadequate testing. While literature on MCA were reviewed, the study was also based on literature pertaining to MDE for mobile development; as MDE is a prominent technique in achieving development efficiency among several other benefits. The research showed that existing mobile MDE approaches have not explored the MCA domain and moreover they cannot be directly applied to the MCA domain due to the multi-tier (mobile and cloud) nature of MCAs. Consequently, the motivation to apply MDE in the MCA domain to eliminate the challenges in existing MCA optimisation approaches – mostly caused due to the use of custom runtimes in MCAs. Also existing MCA testing/evaluation approach is not robust to support the evaluation of mobile tier as well as cloud tier of MCA; consequently a need for a full-tier evaluation approach. Thus the concerns of MCA are generally associated with the SDLC of MCA; in terms of development and testing.

The thesis aims towards an efficient MCA development approach which takes into consideration the mobile and cloud tier during optimisation – and consequently improves full-tier qualities (mobile energy and performance; cloud resource and availability). The approach applies the MDE concept in MCA order to facilitate development efficiency. Furthermore; an efficient testing approach is achieved for effective full-tier evaluation of MCAs. The key contributions are summarised below;

8.2.1 Contribution I: Mango Approach

This thesis presented a novel architecture and process for MCA development named Mango. It owns the following main features; 1) it introduces a process for MCA analysis which is based on identification of offloadable tasks, 2) it employs MDE; thus proposing meta-modelling in MCA called the Caller-Callee model which handles full-tier specification of qualities, and 3) it reveals an optimisation approach based on a design pattern for MCA called ACTS; which facilitates reuse. In Mango, optimisation logic is implemented as an ACTS design pattern.

The benefit of the Mango architecture is in achieving a model-driven approach to MCA development which drives development efficiency – realised by the meta-modelling concept. Also by integrating full-tier qualities, i.e. qualities for both mobile and cloud tiers, into the MCA, the architecture achieves better overall efficiency for the MCA (realised as better mobile performance and energy usage and better cloud resource usage and awareness of software availability).

8.2.2 Contribution II: Context-aware Green Architecture

The Mango architecture was proposed in the thesis as a context-aware architecture. Optimisation in the architecture is based on awareness of two kinds of context, which are user context and environmental context. Consequently, due to the finer granularity of context-awareness the architecture achieves mobile performance and energy savings even in adverse environment conditions – as shown in the experiments (section 7.6.2). Also as an architecture which targets full-tier qualities, the consequence of context-aware decision making in Mango also yields resource savings in the cloud tier. This has also been demonstrated in the experiments (section 7.6.2).

8.2.3 Contribution III: Mosaic Approach

This thesis presented a novel MDE framework named Mosaic for realising the proposed Mango architecture objective. A key novelty of Mosaic is its seamless integration with the mobile development environment as an API – via the Gradle console. Its novelty contribution is based on the following main features; 1) Selective Analyser for identification of offloadable tasks, 2) an API interface for specifying quality attributes for underlying meta-model, 3) a transformation engine which uses meta-model and ACTS templates to generate application code. Mosaic optimisation logic is defined in the templates implementing ACTS pattern; thus facilitating reuse. And 4) Profiler Aspect; a profiling system for architecture evaluation to ensure that an optimisation process for an identified offloadable task will most certainly yield benefits – thus avoiding optimisation overhead. Mosaic is model-driven, and the benefit is to achieve a platform-independent design of the Mango solution for MCA. Using the Model, quality attributes can be specified for the mobile and cloud tier of the application while modelling offloadable components. The modeller is a graph-based modelling tool which generates information in XMI format. This makes it highly interoperable; as transformation code can be written for any platform (based on ACTS pattern) to consume the model.

8.2.4 Contribution IV: Beftigre Evaluation Approach

This thesis presented Beftigre approach for evaluation of MCA. An important achievement of Beftigre is in the full-tier evaluation capability and behaviour-driven concept. Full-tier evaluation makes it possible to evaluate the MCA at a finer granularity which takes into consideration metrics from both mobile and cloud tiers. Behaviour-driven evaluation makes it possible to provide a consistent and reliable comparison between other approaches or counterpart techniques. (See Chapter 6 for details). Experiments have shown in comparison to the approach adopted in the literature, that Beftigre is more reliable providing results at a fine granularity and reproducibility.

8.3 Future Work

Considering future research directions on mobile development, the future work will target extending the proposed frameworks for multiple platforms as an extended MDE feature. For example; although Mango is model-driven, the Mosaic transformation tool which realises the architecture has been explored in the context of Android platform. Therefore for future development, Mosaic should be extended to support other popularly used mobile platforms (such as Windows Phone and iPhone) – this can be done by implementing the ACTS design pattern for the respective platforms and transformation code to transform meta-model based on the pattern. The aforementioned cross-platform transformation feature will further enhance the adoption of Mango approach in MCAs.

Also, the current MCA meta-model proposed by this thesis was focused on MCA offloadable tasks, future work can integrate the MCA meta-model with the generic meta-model for a mobile application. This can be achieved by extending the proposed MCA meta-model of this thesis to incorporate any of the existing reviewed mobile modelling frameworks (in the literature review chapter). Similarly, the APIs which have been implemented based on the proposed Beftigre test framework can be extended to target more platforms. Unlike Mango and Mosaic, Beftigre is not model-driven but rather language-specific. Consequently, implementing Beftigre for different platforms will involve re-implementing the entire system. This would also involve implementing for different OS platforms in the cloud as well as mobile platforms. Therefore research will have to properly investigate ways to achieve this, and any third party libraries that may assist (for example; libraries to assist in monitoring a Windows cloud environment, and power monitoring for windows phone).

References

[1] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, “A survey of mobile cloud computing: architecture, applications, and approaches,” *Wirel. Commun. Mob. Comput.*, vol. 13, no. 18, pp. 1587–1611, 2013.

[2] S. Bhattacharya, K. Gopinath, K. Rajamani, and M. Gupta, “Software Bloat and Wasted Joules : Is Modularity a Hurdle to Green Software?,” *Computer (Long. Beach. Calif).*, vol. 44, no. 9, pp. 97–101, 2011.

[3] E. Capra, C. Francalanci, and S. A. Slaughter, “Is software ‘green’? Application development environments and energy efficiency in open source applications,” *Inf. Softw. Technol.*, vol. 54, no. 1, pp. 60–71, Jan. 2012.

[4] Y.-W. Kwon and E. Tilevich, “Energy-Efficient and Fault-Tolerant Distributed Mobile Execution,” in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, 2012, pp. 586–595.

[5] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys ’10*, 2010, vol. 17, pp. 49–62.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: elastic execution between mobile device and cloud,” in *Proceedings of the sixth conference on Computer systems - EuroSys ’11*, 2011, pp. 301–314.

[7] M. A. Hassan, K. Bhattarai, Q. Wei, and S. Chen, “POMAC : Properly Offloading Mobile Applications to Clouds,” in *Proceedings of the 6th USENIX conference on Hot Topics in Cloud Computing*, 2014, pp. 1–6.

[8] M. A. Hassan, Q. Wei, and S. Chen, “Elicit : Efficiently Identify Computation-intensive Tasks in Mobile Applications for Offloading,” in *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2015, pp. 12–22.

[9] B. Trask, D. Paniscotti, A. Roman, and V. Bhanot, “Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications,” in *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA ’06*, 2006, pp. 846–853.

[10] G. Lami, L. Buglione, and F. Fabbrini, “Derivation of Green Metrics for Software,” in *Software Process Improvement and Capability Determination*, T. Woronowicz, T. Rout, R. V. O’Connor, and A. Dorling, Eds. Springer Berlin Heidelberg, 2013, pp. 13–24.

[11] C. Calero and M. Piattini, “Introduction to Green in Software Engineering,” in *Green in Software Engineering*, Cham: Springer International Publishing, 2015, pp. 3–27.

[12] M. Dick, S. Naumann, and N. Kuhn, “A Model and Selected Instances of Green and Sustainable Software,” in *What Kind of Information Society? Governance, Virtuality, Surveillance, Sustainability, Resilience*, vol. 328, J. Berleur, M. D. Hercheui, and L. M. Hilty, Eds. Springer Berlin Heidelberg, 2010, pp. 248–259.

[13] S. Naumann, M. Dick, E. Kern, and T. Johann, “The GREENSOFT Model: A reference model for green and sustainable software and its engineering,” *Sustain. Comput. Informatics Syst.*, vol. 1, no. 4, pp. 294–304, Dec. 2011.

[14] D. Rogers and U. Homann, “Application Patterns for Green IT,” *The Architecture Journal - Green Computing*, 2009. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd393307.aspx. [Accessed: 18-Jan-2016].

[15] S. Murugesan, “Harnessing Green IT: Principles and Practices,” *IT Prof.*, vol. 10, no. 1, pp. 24–33, 2008.

[16] C. Reimsbach-kounatze, “Towards Green ICT Strategies - Assessing Policies and Programmes on ICT and The Environment,” no. 155. OECD Working Party on the Information Economy, 2009.

[17] B. Steigerwald and A. Agrawal, “Developing Green Software | Intel® Developer Zone,” 2011. [Online]. Available: http://software.intel.com/en-us/articles/developing-green-software. [Accessed: 18-Jan-2016].

[18] P. Bozzelli, Q. Gu, and P. Lago, “A systematic literature review on green software metrics,” VU University Amsterdam, The Netherlands, 2013.

[19] J. Taina and S. Mäkinen, “Green Software Quality Factors,” in *Green in Software Engineering*, Cham: Springer International Publishing, 2015, pp. 129–154.

[20] I. Goiri, R. Beauchea, K. Le, T. D. Nguyen, M. E. Haque, J. Guitart, J. Torres, and R. Bianchini, “GreenSlot: Scheduling energy consumption in green datacenters,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’11*, 2011, pp. 1–11.

[21] N. Vasić, P. Bhurat, D. Novaković, M. Canini, S. Shekhar, and D. Kostić, “Identifying and Using Energy-Critical Paths,” in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies on - CoNEXT ’11*, 2011, pp. 1–12.

[22] L. Benini and G. De Micheli, “System-level power optimization: techniques and tools,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 2, pp. 115–192, 2000.

[23] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Boston: Addison-Wesley Professional, 2003.

[24] M. A. Khan, C. Hankendi, A. K. Coskun, and M. C. Herbordt, “Software optimization for performance, energy, and thermal distribution: Initial case studies,” in *2011 International Green Computing Conference and Workshops*, 2011, pp. 1–6.

[25] M. Denti and J. K. Nurminen, “Performance and energy-efficiency of scala on mobile devices,” in *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*, 2013, pp. 50–55.

[26] C.-W. You and H. Chu, “Replicated client-server execution to overcome unpredictability in mobile environment,” in *2004 4th Workshop on Applications and Services in Wireless Networks, 2004. ASWN 2004.*, 2004, pp. 21–29.

[27] M.-D. Cano and G. Domenech-Asensi, “A secure energy-efficient m-banking application for mobile devices,” *J. Syst. Softw.*, vol. 84, no. 11, pp. 1899–1909, 2011.

[28] I. Sommerville, *Software Engineering*, 9th ed. Pearson, 2011.

[29] J. Williams and L. Curtis, “Green: The New Computing Coat of Arms?,” *IT Prof.*, vol. 10, no. 1, pp. 12–16, 2008.

[30] C.-H. Hsu, S.-C. Chen, C.-C. Lee, H.-Y. Chang, K.-C. Lai, K.-C. Li, and C. Rong, “Energy-Aware Task Consolidation Technique for Cloud Computing,” in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*

(CloudCom), 2011, pp. 115–121.

[31] B. Zhong, M. Feng, and C.-H. Lung, "A Green Computing Based Architecture Comparison and Analysis," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, 2010, pp. 386–391.

[32] R. Morgan and D. MacEachern, "SIGAR - System Information Gatherer And Reporter," 2010. [Online]. Available: https://support.hyperic.com/display/SIGAR/Home. [Accessed: 18-Jan-2016].

[33] J. Tayeb, K. Bross, C. S. Bae, C. Li, and S. Rogers, "Intel Energy Checker Software Development Kit User Guide," 2010. [Online]. Available: https://goo.gl/Yrtbn9. [Accessed: 01-Jul-2016].

[34] T. Johann, M. Dick, S. Naumann, and E. Kern, "How to measure energy-efficiency of software: Metrics and measurement results," in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, 2012, pp. 51–54.

[35] K. Naik and D. S. L. Wei, "Software Implementation Strategies for Power-Conscious Systems," *Mob. Networks Appl.*, vol. 6, no. 3, pp. 291–305, 2001.

[36] N. Amsel and B. Tomlinson, "Green Tracker : A Tool for Estimating the Energy Consumption of Software," in *Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems - CHI EA '10*, 2010, pp. 3337–3342.

[37] N. Amsel, Z. Ibrahim, A. Malik, and B. Tomlinson, "Toward Sustainable Software Engineering (NIER Track)," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, 2011, pp. 976–979.

[38] A.-L. Kor, C. Pattinson, I. Imam, I. AlSaleemi, and O. Omotosho, "Applications, energy consumption, and measurement," in *2015 International Conference on Information and Digital Technologies*, 2015, pp. 161–171.

[39] A. Sinha and A. P. Chandrakasan, "JouleTrack - A Web Based Tool for Software Energy Profiling," in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 220–225.

[40] A. Kansal, F. Zhao, and A. A. Bhattacharya, "Virtual Machine Power Metering and Provisioning," in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, 2010, pp. 39–50.

[41] M. Yi and J. De Vega, "Intel® Power Gadget," 2015. [Online]. Available: https://software.intel.com/en-us/articles/intel-power-gadget-20. [Accessed: 18-Jan-2016].

[42] R. Yamini, "Power Management in Cloud Computing Using Green Algorithm," in *2012 International Conference On Advances In Engineering, Science And Management (ICAESM)*, 2012, pp. 128–133.

[43] R. Yanggratoke, F. Wuhib, and R. Stadler, "Gossip-based Resource Allocation for Green Computing in Large Clouds," in *2011 7th International Conference on Network and Service Management*, 2011, pp. 171–179.

[44] H.-M. Chen and R. Kazman, "Architecting ultra-large-scale green information systems," in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, 2012, pp. 69–75.

[45] J. Baliga, R. W. a Ayre, K. Hinton, and R. S. Tucker, "Green Cloud Computing:

Balancing Energy in Processing, Storage, and Transport," *Proc. IEEE*, vol. 99, no. 1, pp. 149–167, Jan. 2011.

[46] D. Fang, X. Liu, L. Liu, and H. Yang, "TARGO: Transition and reallocation based green optimization for cloud VMs," *Proc. - 2013 IEEE Int. Conf. Green Comput. Commun. IEEE Internet Things IEEE Cyber, Phys. Soc. Comput. GreenCom-iThings-CPSCom 2013*, pp. 215–223, 2013.

[47] S. J. Chinenyeze, X. Liu, and A. Al-dubai, "An Aspect Oriented Model for Software Energy Efficiency in Decentralised Servers," in *2nd International Conference on ICT for Sustainability*, 2014, vol. 2, pp. 112–119.

[48] Y. Jadeja and M. Kirit, "Cloud Computing - Concepts, Architecture and Challenges," in *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)*, 2012, pp. 877–880.

[49] W.-T. Tsai, X. Sun, and J. Balasooriya, "Service-Oriented Cloud Computing Architecture," in *2010 Seventh International Conference on Information Technology: New Generations (ITNG)*, 2010, pp. 684–689.

[50] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES/ISSS '10*, 2010, pp. 105–114.

[51] N. Vallina-rodriguez, P. Hui, J. Crowcroft, and A. Rice, "Exhausting Battery Statistics," in *Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds - MobiHeld '10*, 2010, pp. 9–14.

[52] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android Java code for on-demand computation offloading," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12*, 2012, vol. 47, no. 10, p. 233.

[53] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Futur. Gener. Comput. Syst.*, vol. 29, no. 1, pp. 84–106, 2013.

[54] S. Zachariadis, C. Mascolo, and W. Emmerich, "SATIN: A component model for mobile self organisation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3291, Springer Berlin Heidelberg, 2004, pp. 1303–1321.

[55] M. Wichtlhuber, J. Rückert, D. Stingl, M. Schulz, and D. Hausheer, "Energy-efficient mobile P2P video streaming," in *2012 IEEE 12th International Conference on Peer-to-Peer Computing, P2P 2012*, 2012, pp. 63–64.

[56] E. E. Marinelli, "Hyrax : Cloud Computing on Mobile Devices using MapReduce," Carnegie Mellon University, Pittsburgh, PA 15213, 2009.

[57] P. Yu, X. Ma, J. Cao, and J. Lu, "Application mobility in pervasive computing: A survey," *Pervasive Mob. Comput.*, vol. 9, no. 1, pp. 2–17, 2013.

[58] T. Justino and R. Buyya, "Outsourcing resource-intensive tasks from mobile apps to clouds: Android and aneka integration," in *2014 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2014, pp. 1–8.

[59] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim, "Architecture-aware automatic computation offload for native applications," in *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, 2015, pp. 521–532.

[60] T. Lemlouma and N. Layaida, “Context-aware adaptation for mobile devices,” in *IEEE International Conference on Mobile Data Management, 2004. Proceedings. 2004*, 2004, pp. 106–111.

[61] S. Miyake and M. Bandai, “Energy-Efficient Mobile P2P Communications Based on Context Awareness,” in *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, 2013, pp. 918–923.

[62] S. Huang and J. Mangs, “Pervasive Computing: Migrating Applications to Mobile Devices: A Case Study,” in *2008 2nd Annual IEEE Systems Conference*, 2008, pp. 1–8.

[63] M. Satyanarayanan, “Pervasive computing: vision and challenges,” *IEEE Pers. Commun.*, vol. 8, no. 4, pp. 10–17, 2001.

[64] D. Saha, “Pervasive Computing: A Paradigm for the 21st Century,” *Computer (Long. Beach. Calif).*, vol. 36, no. 3, pp. 25–31, 2003.

[65] “PowerTutor,” 2013. [Online]. Available: https://github.com/msg555/PowerTutor. [Accessed: 18-Jan-2016].

[66] M. Mari and N. Eila, “The impact of maintainability on component-based software systems,” in *Proceedings of the 20th IEEE Instrumentation Technology Conference (Cat No 03CH37412) EURMIC-03*, 2003, pp. 25–32.

[67] A. Immonen, “A method for predicting reliability and availability at the architecture level,” *Softw. Prod. Lines Res. Issues Eng. Manag.*, pp. 373–422, 2006.

[68] A. Immonen and E. Niemelä, “Survey of reliability and availability prediction methods from the viewpoint of software architecture,” *Softw. Syst. Model.*, vol. 7, no. 1, pp. 49–65, 2008.

[69] “NQueen,” 2015. [Online]. Available: https://play.google.com/store/apps/details?id=com.memmiolab.queens. [Accessed: 18-Jan-2016].

[70] “Mezzofanti,” 2009. [Online]. Available: https://code.google.com/p/mezzofanti/. [Accessed: 18-Jan-2016].

[71] “Picaso,” 2013. [Online]. Available: https://code.google.com/p/picaso-eigenfaces/. [Accessed: 18-Jan-2016].

[72] “MatCalc,” 2012. [Online]. Available: https://github.com/kc1212/matcalc. [Accessed: 18-Jan-2016].

[73] “MathDroid,” 2013. [Online]. Available: https://f-droid.org/repository/browse/?fdid=org.jessies.mathdroid. [Accessed: 18-Jan-2016].

[74] “ZXing,” 2016. [Online]. Available: https://github.com/zxing/zxing. [Accessed: 18-Jan-2016].

[75] Droidslator, “Droidslator,” 2010. [Online]. Available: https://code.google.com/p/droidslator/. [Accessed: 18-Jan-2016].

[76] “JJIL,” 2009. [Online]. Available: https://code.google.com/p/jjil/. [Accessed: 18-Jan-2016].

[77] “OSMAnd,” 2016. [Online]. Available: https://github.com/osmandapp/Osmand. [Accessed: 18-Jan-2016].

[78] P. Čokulov, “Linpack,” 2014. [Online]. Available: https://github.com/pedja1/Linpack. [Accessed: 18-Jan-2016].

[79] “XRace,” 2008. [Online]. Available: https://code.google.com/p/xrace-sa/. [Accessed: 18-Jan-2016].

[80] Y. Gu, V. March, and B. S. Lee, “GMoCA: Green mobile cloud applications,” in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, 2012, pp. 15–20.

[81] R. Laddad, *AspectJ In Action: Enterprise AOP with Spring Applications*, Second Ed. Manning Publications Co., 2010.

[82] M. Forgáč and J. Kollár, “Static and Dynamic Approaches to Weaving,” in *5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics*, 2007, no. 1, pp. 201–210.

[83] S. Chiba, Y. Sato, and M. Tatsubori, “Using HotSwap for Implementing Dynamic AOP Systems,” *1st Workshop on Advancing the State-of-the-Art in Run-time Inspection*. 2003.

[84] S. J. Chinenyeze, X. Liu, and A. Al-dubai, “DEEPC : Dynamic Energy Profiling of Components,” in *40th IEEE Computer Society International Conference on Computers, Software & Applications*, 2016, pp. 1–6.

[85] The AspectJ Team, “The AspectJ (TM) Programming Guide,” 2003. [Online]. Available: http://www.eclipse.org/aspectj/doc/released/progguide/index.html. [Accessed: 18-Jan-2016].

[86] L. Gaouar, A. Benamar, and F. T. Bendimerad, “Model Driven Approaches to Cross Platform Mobile Development,” in *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, 2015, pp. 1–5.

[87] V. Cortellessa, A. Di Marco, and P. Inverardi, “Software performance model-driven architecture,” in *Proceedings of the 2006 ACM symposium on Applied computing - SAC ’06*, 2006, pp. 1218–1223.

[88] D. C. Schmidt, “Model-Driven Engineering,” *Computer (Long. Beach. Calif).*, vol. 39, no. 2, pp. 25–31, 2006.

[89] M. A. Laguna and B. Gonzalez-Baixauli, “Requirements Variability Models: Meta-model based Transformations,” in *Proceedings of the 2005 symposia on Metainformatics - MIS ’05*, 2005, vol. 214, pp. 1–9.

[90] A. Carton, S. Clarke, A. Senart, and V. Cahill, “Aspect-Oriented Model-Driven Development for Mobile Context-Aware Computing,” in *First International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments (SEPCASE ’07)*, 2007, pp. 1–4.

[91] S. Komatineni and D. MacLean, “Introducing the Android Computing Platform,” in *Pro Android 4*, Berkeley, CA: Apress, 2012, pp. 1–22.

[92] F. A. Kraemer, “Engineering android applications based on UML activities,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6981 LNCS, pp. 183–197, 2011.

[93] M. Usman, M. Z. Iqbal, and M. U. Khan, “A Model-Driven Approach to Generate Mobile Applications for Multiple Platforms,” in *2014 21st Asia-Pacific Software Engineering Conference*, 2014, pp. 111–118.

[94] A. G. Parada and L. B. de Brisolara, “A Model Driven Approach for Android Applications Development,” in *2012 Brazilian Symposium on Computing System Engineering (SBESC)*, 2012, pp. 192–197.

[95] K. Lamhaddab and K. Elbaamrani, “Model Driven Reverse Engineering: Graph Modeling For Mobiles Platforms,” in *2015 15th International Conference on Intelligent Systems Design and Applications (ISDA)*, 2015, pp. 392–397.

[96] X. Jia and C. Jones, “Cross-Platform Application Development Using AXIOM as an Agile Model-Driven Approach,” in *Software and Data Technologies*, vol. 411 CCIS, Springer Berlin Heidelberg, 2013, pp. 36–51.

[97] A. G. Parada, E. Siegert, and L. B. De Brisolara, “Generating Java code from UML class and sequence diagrams,” in *2011 Brazilian Symposium on Computing System Engineering*, 2011, pp. 99–101.

[98] “applause,” 2014. [Online]. Available: https://github.com/applause/applause. [Accessed: 18-Jan-2016].

[99] Automobile, “The AutoMobile Project | Automated Mobile App Development,” 2014. [Online]. Available: http://automobile.webratio.com. [Accessed: 18-Jan-2016].

[100] C. Jones and X. Jia, “Using a Domain Specific Language for Lightweight Model-Driven Development,” in *Evaluation of Novel Approaches to Software Engineering*, vol. 551 CCIS, L. A. Maciaszek and J. Filipe, Eds. Springer International Publishing, 2015, pp. 46–62.

[101] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, “Cross-Platform Model-Driven Development of Mobile Applications with MD2,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC ’13*, 2013, pp. 526–533.

[102] Z. Hemel and E. Visser, “Declaratively programming the mobile web with Mobl,” *ACM SIGPLAN Not.*, vol. 46, no. 10, pp. 695–712, 2011.

[103] Z. Hemel and E. Visser, “Mobl: The New Language of the Mobile Web,” in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH ’11*, 2011, pp. 23–24.

[104] Y. Falcone and S. Currea, “Weave droid: aspect-oriented programming on Android devices: fully embedded or in the cloud,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 2012, pp. 350–353.

[105] W. Zhang, Y. Wen, and D. O. Wu, “Energy-efficient scheduling policy for collaborative execution in mobile cloud computing,” in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 190–194.

[106] A. Saarinen, M. Siekkinen, Y. Xiao, J. K. Nurminen, M. Kemppainen, and D. T. Labs, “Can Offloading Save Energy for Popular Apps?,” in *Proceedings of the seventh ACM international workshop on Mobility in the evolving internet architecture - MobiArch ’12*, 2012, pp. 3–10.

[107] A. Rohatgi, “WebPlotDigitizer - Extract data from plots, images, and maps,” 2016. [Online]. Available: http://arohatgi.info/WebPlotDigitizer/. [Accessed: 18-Jan-2016].

[108] C. Solís and X. Wang, “A study of the characteristics of behaviour driven development,” in *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011*, 2011, pp. 383–387.

[109] M. Kuna, H. Kolaric, I. Bojic, M. Kusek, and G. Jezic, “Android / OSGi-based Machine-

to-Machine Context-Aware System," in *Proceedings of the 2011 11th International Conference on Telecommunications (ConTEL)*, 2011, pp. 95–102.

[110] K. Beck and W. Cunningham, "A laboratory for teaching object oriented thinking," *ACM SIGPLAN Not.*, vol. 24, no. 10, pp. 1–6, 1989.

[111] "JGraphX," 2016. [Online]. Available: https://github.com/jgraph/jgraphx. [Accessed: 18-Jan-2016].

[112] M. Hüttermann, "Specification by Example," in *DevOps for Developers*, Berkeley, CA: Apress, 2012, pp. 157–170.

[113] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "AppScope: Application Energy Metering Framework for Android Smartphones Using Kernel Activity Monitoring," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012, p. 36.

[114] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh, "Who killed my battery?: analyzing mobile browser energy consumption," in *Proceedings of the 21st international conference on World Wide Web*, 2012, pp. 41–50.

[115] A. Shackelford, "Monitoring the Application," in *Beginning Amazon Web Services with Node.js*, Berkeley, CA: Apress, 2015, pp. 171–208.

[116] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "COSMOS : Computation Offloading as a Service for Mobile Devices," in *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing - MobiHoc '14*, 2014, pp. 287–296.

[117] C. Seo, S. Malek, and N. Medvidovic, "Estimating the energy consumption in pervasive java-based systems," in *6th Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2008*, 2008, pp. 243–247.

[118] "Testing from the Command-Line | Android Developers." [Online]. Available: http://developer.android.com/tools/testing/testing_otheride.html#RunTestsCommand. [Accessed: 18-Jan-2016].

[119] RobotiumTech, "Robotium," 2016. [Online]. Available: https://github.com/robotiumtech/robotium. [Accessed: 01-Jul-2016].

Appendix A Abbreviations and Acronyms

All the abbreviations and acronyms used in this thesis are defined below.

Abbreviation /Acronyms	**Description**
ACTS	Aspect Context Task Service design pattern
AMEE	Aspect-oriented Model for Energy-Efficiency at server layer
AOP	Aspect Oriented Programming
BDD	Behaviour-Driven Development
BEFTIGRE	Behaviour-driven Full-tier Green Evaluation
CRAC	Context-driven Requirements Analysis for Caller-Callee model
DEEPC	Dynamic Energy Profiling of Components
DVM	Dalvik Virtual Machine
DSL	Domain Specific Language
EE	Energy-efficiency
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
I/O	Input/output
IDE	Integrated Development Environment
JVM	Java Virtual Machine
MANGO	Model-driven Architecture for integration of software quality with Green Optimisation in MCAs
MCA	Mobile Cloud Applications
MCC	Mobile Cloud Computing
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MOSAIC	Model-based Selective Approach for Identification of Computation intensive tasks
OOP	Object-Oriented Programming
PaaS	Platform as a Service
PIM	Platform Independent Model
PSM	Platform Specific Model
SaaS	Software as a Service
SDLC	Software Development Life Cycle
SDP	Software Development Process
VM	Virtual Machine

**Italicised* are contributed from this research: *ACTS*, *AMEE*, *BEFTIGRE*, *CRAC*, *DEEPC*, *MANGO*, and *MOSAIC*.

Appendix B Selection Criteria for Case Studies

The case studies used to evaluate Mango, are chosen from the pool of applications used to evaluate the offloading schemes/approaches in the research. A complete listing of the case studies used in the literatures are given in Table 1 (which gives 12 apps in total).

Table 1: List of case studies used in the literature

S/N	**Apps**[31]	**Tax**	**Sample Offloading Schemes**			**Description**
			POMAC/Elicit	EFDM	DPartner	
1	Picaso [71]	D	✓			Face recognition app
2	MatCalc [72]	D	✓			Matrix calculator
3	MathDroid [73]	D	✓			Calculator
4	NQueen [69]	C	✓			NQueen game
5	Droidslator [75]	CD	✓	✓		Translation app
6	Mezzofanti [70]	C	✓	✓		OCR app
7	ZXing [74]	D	✓	✓		Bar code reader
8	JJIL [76]	C		✓		Face recognition app
9	OsmAnd [77]	C		✓		Street map
10	Andgoid [52]	CD			✓	Chess game
11	Linpack [78]	C			✓	Linear algebra benchmark app
12	XRace [79]	CD			✓	Car racing game

Key. Tax: Application taxonomy. C: Computation intensive. D: Data intensive.

Notice that each literature uses both computation and data -intensive applications to evaluate the schemes. Similarly, for this research, two computation intensive and two data intensive applications have been chosen.

The criteria for selection of the case study applications are code accessibility, application correctness and network robustness (Table 2).

- Code Accessibility: the source code for a selected app must be accessible, not just the android application package (APK installer). This is important to be able to perform static analysis and offload refactoring. Note that other offload techniques which optimise at bytecode level or at runtime, may not need/require the source code for experimentation, hence obsolete apps were feasible case studies for such research, however Mango approach requires source code access.

[31] The Apps are the source code required for experimentation. Note that the references appended to the Apps links to the source code or google play app.

- App Correctness: a selected application must be able to execute with no errors – i.e. having the relevant features in performing as expected. The correctness of the app is important as error-composed/buggy application can impact experimental results of the research. Moreover, applications which are not functionally correct may even compromise offloading decision – if these segments are the computation intensive components of the app. e.g. Mezzofanti, which is missing a language pack.
- Network Robustness: a selected app has to be able to run their key features with or without an internet connection. This criteria guarantees that all key requirements of the app are as well available in the local execution scenario.

Criteria values: 'Yes' means the condition is satisfied, 'No' means the condition was not satisfied. '-' means the condition was not determined as the code was not accessible. Note that; for code accessibility, if only APK file is found but no source code found, 'No' is marked. In a situation where all conditions/criteria are satisfied, the app is selected as a case study.

Table 2: Case studies matching the selection criteria.

S/N	**Apps**[32]	**Code Accessibility**	**App Correctness**	**Network Robustness**
1	Picaso [71]	Yes	No	Yes
2	MatCalc [72]	Yes	Yes	Yes
3	MathDroid [73]	Yes	Yes	Yes
4	NQueen [69]	Yes	Yes	Yes
5	Droidslator [75]	Yes	No	No
6	Mezzofanti [70]	Yes	No	No
7	ZXing [74]	Yes	Yes	Yes
8	JJIL [76]	No	-	-
9	OsmAnd [77]	Yes	Yes	No
10	Andgoid [52]	No	-	-
11	Linpack [78]	Yes	Yes	Yes
12	XRace [79]	Yes	No	No

OsmAnd was found to contain build errors which could not be resolved due to some inaccessible modules. After refactoring, the refined app was found to be tightly coupled to remote services, consequently, it was not selected as a case study. Similarly Droidslator, Mezzofanti and XRace failed the network robustness selection criteria. Some other apps such as NQueen, JJIL and Andgoid could not be determined as the source code of the

[32] The Apps are the source code required for experimentation. Note that the references appended to the Apps links to the source code or google play app.

applications could not be found. Picaso was missing core face database, and thus could not be used. Out of the 12 samples presented in Table 2, four passed the criteria for selection – as shown in Table 3. The selected case studies are; Linpack, MatCalc, MathDroid and NQueen.

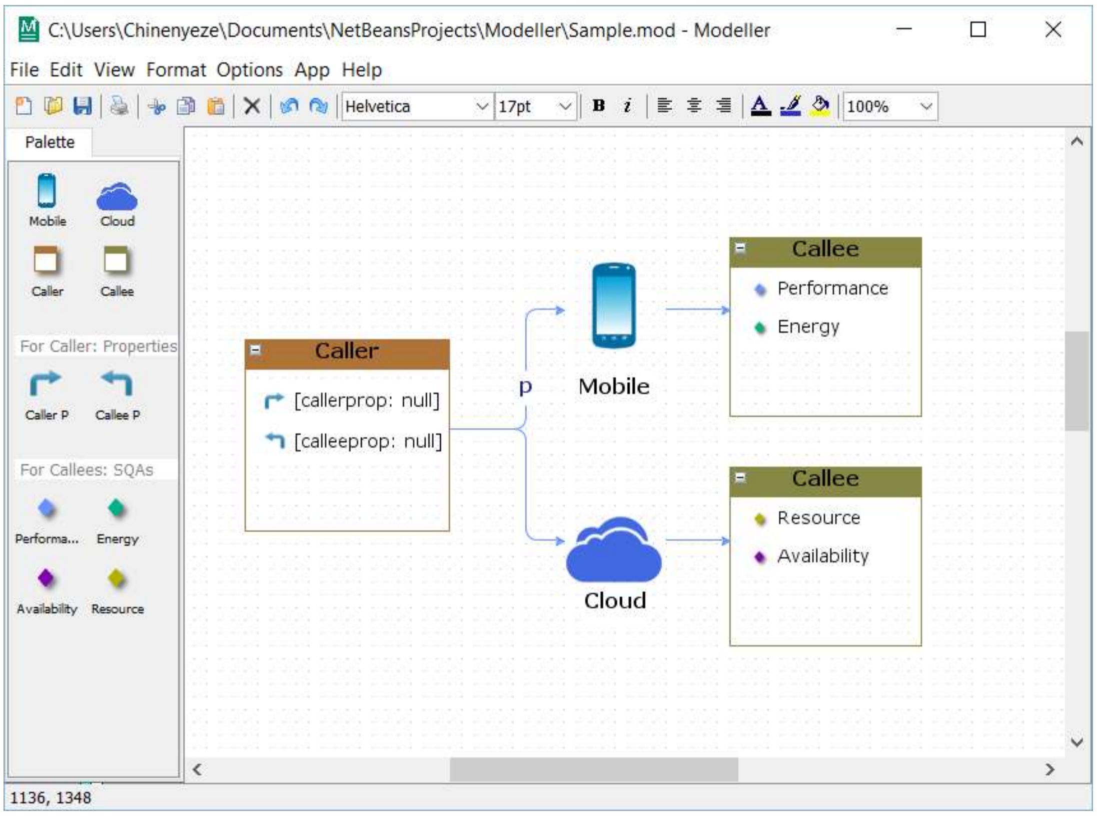
Table 3: Selected Case Studies.

Apps	**Computation intensive**	**Data intensive**
Linpack [78]	✓	
MatCalc [72]		✓
MathDroid [73]		✓
NQueen [69]	✓	

As shown in the analysis of case studies above, the kind of taxonomies predominant in the literatures are computation intensive and data intensive applications. Consequently, the experiments for MANGO aims to demonstrate that the model is efficient in these taxonomy of applications.

Appendix C Mosaic Modeller

C.1 Modeller showing sample Caller-Callee model diagram



C.2 Mosaic Model File (.mod)

```
<?xml version="1.0" encoding="UTF-8"?>
<mxGraphModel>
  <root>
    <mxCell id="0" />
    <mxCell id="1" parent="0" />
    <caller id="2" name="Caller">
      <mxCell parent="1" style="caller" vertex="1">
        <mxGeometry as="geometry" height="150" width="150" x="120" y="250" />
      </mxCell>
    </caller>
    <callerprop id="21" name="Caller Name">
      <mxCell parent="2" style="label;image=/images/p_caller.png" vertex="1">
        <mxGeometry as="geometry" height="20" width="120" x="10" y="40" />
      </mxCell>
    </callerprop>
    <calleeprop id="22" name="Caller Name">
      <mxCell parent="2" style="label;image=/images/p_callee.png" vertex="1">
        <mxGeometry as="geometry" height="23" width="130" x="10" y="69" />
      </mxCell>
    </calleeprop>
    <mobile id="3" name="Mobile">
      <mxCell parent="1" style="image;image=/images/mobile.png" vertex="1">
        <mxGeometry as="geometry" height="100" width="80" x="370" y="380" />
      </mxCell>    </mobile>
    <cloud id="4" name="Cloud">
      <mxCell parent="1" style="image;image=/images/cloud.png" vertex="1">
        <mxGeometry as="geometry" height="75" width="80" x="380" y="190" />
      </mxCell>
    </cloud>
    <mxCell edge="1" id="5" parent="1" source="2" style="" target="3" value="">
      <mxGeometry as="geometry" relative="1">
        <mxPoint as="sourcePoint" x="410.0" y="230.0" />
        <mxPoint as="targetPoint" x="250.0" y="480.0" />
      </mxGeometry>
    </mxCell>
    <mxCell edge="1" id="6" parent="1" source="2" style="" target="4" value="">
      <mxGeometry as="geometry" relative="1">
        <mxPoint as="sourcePoint" x="410.0" y="230.0" />
        <mxPoint as="targetPoint" x="620.0" y="490.0" />
      </mxGeometry>
    </mxCell>
    <callee id="11" name="Callee">
      <mxCell parent="1" style="callee" vertex="1">
        <mxGeometry as="geometry" height="140" width="150" x="560" y="360" />
      </mxCell>
    </callee>
    <performance id="17" name="Performance">
      <mxCell parent="11" style="label;image=/images/m_perform.png" vertex="1">
        <mxGeometry as="geometry" height="23" width="130" x="10.0" y="30" />
      </mxCell>
    </performance>
    <energy id="18" name="Energy">
      <mxCell parent="11" style="label;image=/images/m_energy.png" vertex="1">
        <mxGeometry as="geometry" height="23" width="130" x="10" y="60" />
      </mxCell>
    </energy>
    <callee id="12" name="Callee">
      <mxCell parent="1" style="callee" vertex="1">
        <mxGeometry as="geometry" height="140" width="150" x="560" y="160" />
      </mxCell>
    </callee>
    <availability id="19" name="Availability">
      <mxCell parent="12" style="label;image=/images/m_avail.png" vertex="1">
        <mxGeometry as="geometry" height="23" width="130" x="10" y="60" />
      </mxCell>
    </availability>
    <resource id="20" name="Resource">
      <mxCell parent="12" style="label;image=/images/m_resrc.png" vertex="1">
        <mxGeometry as="geometry" height="23" width="130" x="10" y="30" />
      </mxCell>
    </resource>
    <mxCell edge="1" id="15" parent="1" source="3"
style="edgeStyle=mxEdgeStyle.EntityRelation;fontSize=18" target="11" value="p">
      <mxGeometry as="geometry" relative="1">
        <mxPoint as="sourcePoint" x="220.0" y="430.0" />
        <mxPoint as="targetPoint" x="230.0" y="730.0" />
      </mxGeometry>
    </mxCell>
    <mxCell edge="1" id="16" parent="1" source="4" style="fontSize=18"
target="12" value="r">
      <mxGeometry as="geometry" relative="1">
        <mxPoint as="sourcePoint" x="620.0" y="450.0" />
        <mxPoint as="targetPoint" x="660.0" y="720.0" />
      </mxGeometry>
    </mxCell>
  </root>
</mxGraphModel>
```

Appendix D Mosaic Templates for ACTS

D.1 Aspect Template

```
package mango;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;

@org.aspectj.lang.annotation.Aspect
public class Aspect {

    @Pointcut("call(* [Callee](..)) && args([ArgumentIDs])")
    public static void offloadMethod([Arguments]) {
    }

    @Around("offloadMethod([ArgumentIDs]) && !within(Aspect) && !within(Task)")
    public [Return] aroundOffloadMethodCall(ProceedingJoinPoint jp, [Arguments]) throws Throwable {
        return new Task().execute(new Object[]{[ArgumentIDs]}).get();
    }
}
```

D.2 Context Template

```
package mango;

import android.app.Activity;
import android.content.SharedPreferences;
/*Import an Activity*/;

public class Context {
    private static Activity activity = /*activity*/;
    public static boolean refresh    = false;

    private static final String PREFERENCES = "mangoPreferences";
    private static SharedPreferences settings = activity.getSharedPreferences(PREFERENCES, 0);
    private static SharedPreferences.Editor editor = settings.edit();

    public static long getCloudTime() {
        return settings.getLong("cloudTime", 0);
    }

    public static void setCloudTime(long cloudTime) {
        editor.putLong("cloudTime", cloudTime);
        editor.commit();
    }

    public static long getMobileTime() {
        return settings.getLong("mobileTime", 0);
    }

    public static void setMobileTime(long mobileTime) {
        editor.putLong("mobileTime", mobileTime);
        editor.commit();
    }

    public static String getMode() {
        if (refresh) {
            refresh = false;
            return "mobile";
        }
        return settings.getString("mode", "mobile");
    }

    public static void setMode(String mode) {
        editor.putString("mode", mode);
        editor.commit();
    }
}
```

D.3 Task Template

```
package mango;

import android.os.AsyncTask;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class Task extends AsyncTask<Object, Integer, [Return]> {
    private static [Return] result = null;
    private static String mode; //execution mode
    private static final String MOBILE  = "mobile";
    private static final String CLOUD   = "cloud";
    private static final String DECIDER = "decider";
    private static final int TIMEOUT  = /*Value ms*/;

<sr:p>
    private static final int OVERHEAD = 0;
</sr:p>
<sr:e>
    private static final int OVERHEAD = /*Value ms*/;
</sr:e>
<sr:pe>
    private static char pa = /*get priority attribute from UI*/;

    private static int overhead(){
        if(pa=='e'){
            return /*Value ms*/;
        }
        return 0;
    }
</sr:pe>

    @Override
    protected [Return] doInBackground(final Object[] params) {
        dispatcher();

        switch (mode) {
            case MOBILE:
                long mobileStart = System.currentTimeMillis();
                runOnMobile(params);
                long mobileTime = System.currentTimeMillis() - mobileStart;
                Context.setMobileTime(mobileTime);
                break;
            case CLOUD:
                long cloudStart = System.currentTimeMillis();
                runOnCloud(params);
                long cloudTime = System.currentTimeMillis() - cloudStart;
                Context.setCloudTime(cloudTime);
                break;
        }

        return result;
    }

    private void dispatcher() {
        switch (Context.getMode()) {
            case DECIDER:
                if ((Context.getMobileTime() + OVERHEAD) > Context.getCloudTime()) {//or overhead()
                    mode = CLOUD;
                } else {
                    mode = MOBILE;
                }
                break;
            case MOBILE:
                mode = MOBILE;
                Context.setMode(CLOUD);
                break;
            case CLOUD:
                mode = CLOUD;
                Context.setMode(DECIDER);
                break;
        }
    }

    private void doAvailability(Object[] params) {
        if (result == null) {
            runOnMobile(params);
        }
    }

    private void runOnMobile(Object[] params) {
        result = [Callee]([CastedArguments]);
    }

    private void runOnCloud(Object[] params) {
        try {
            Socket socket = new Socket("[Host]", [Port]);
            socket.setSoTimeout(TIMEOUT);
            ObjectOutputStream outputStream = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream inputStream   = new ObjectInputStream(socket.getInputStream());
            outputStream.writeObject(params);           //write
            result = ([Return]) inputStream.readObject(); //read
            inputStream.close();
            doAvailability(params);
        } catch (Exception ex) { runOnMobile(params); }
    }
}
```

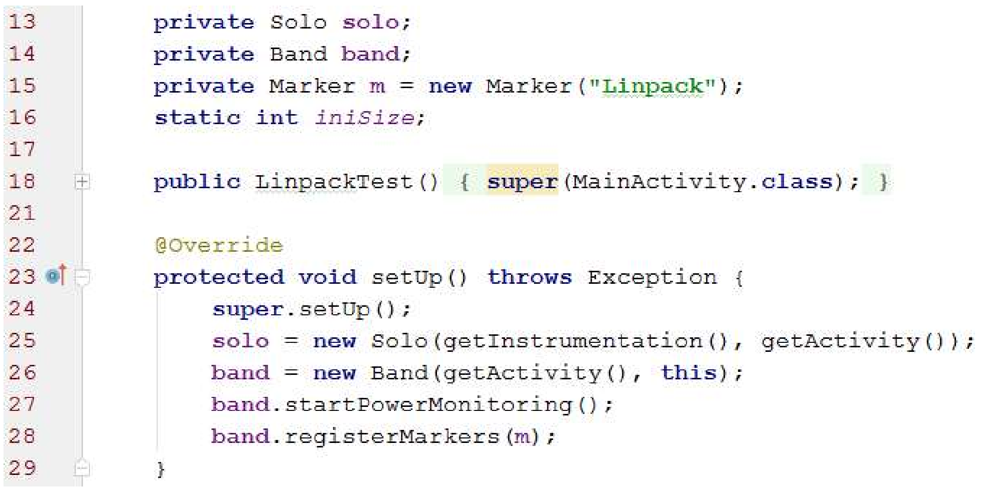
D.4 Service Template

```
package mango;

import org.hyperic.sigar.*;
import java.io.*;
import java.net.*;

public class Service {
    private static final int CPU_THRESHOLD  = /*Value % */;
    private static final int TIME_THRESHOLD = /*Value ms*/;
    private static [Return] result = null;
    private static Thread thread = new Thread();

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket([Port]);
            while (true) {
                Socket socket  = serverSocket.accept();
                ObjectInputStream inputStream = new ObjectInputStream(socket.getInputStream());
                ObjectOutputStream outputStream = new ObjectOutputStream(socket.getOutputStream());

                Object[] params = (Object[])inputStream.readObject(); //read parameters
                dispatcher(params);                                  //execute Callee
                outputStream.writeObject(result);                    //send result to mobile

                outputStream.close();
            }
        } catch (Exception ex) {  }
    }

<sr:a>
    public static void dispatcher(Object[] params) {
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                result = [Callee]([CastedArguments]); //Callee on this server
            }
        });
        thread.start();
        try {
            thread.join(TIME_THRESHOLD);
            if (thread.isAlive()) thread.interrupt();
        } catch (InterruptedException ex) {  }
    }
</sr:a>
<sr:r>
    public static void dispatcher(Object[] params) {
        if (availableCPU() < CPU_THRESHOLD) {
            result = /*reference to the Callee on an alternate server*/
        } else {
            result = [Callee]([CastedArguments]); //Callee on this server
        }
    }
</sr:r>
<sr:ar>
    public static void dispatcher(Object[] params) {
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                if (availableCPU() < CPU_THRESHOLD) {
                    result = /*reference to the Callee on an alternate server*/
                } else {
                    result = [Callee]([CastedArguments]); //Callee on this server
                }
            }
        });
        thread.start();
        try {
            thread.join(TIME_THRESHOLD);
            if (thread.isAlive()) thread.interrupt();
        } catch (InterruptedException ex) {  }
    }
</sr:ar>

    public static int availableCPU() {
        int usedcpu = 0;
        try {
            Sigar sigar = new Sigar();
            usedcpu = (int) Math.round(sigar.getCpuPerc().getCombined() * 100);
        } catch (SigarException ex) {  }
        return 100 - usedcpu;
    }
}
```

Appendix E Beftigre BAND API Screenshots

E.1 Band API Setup method

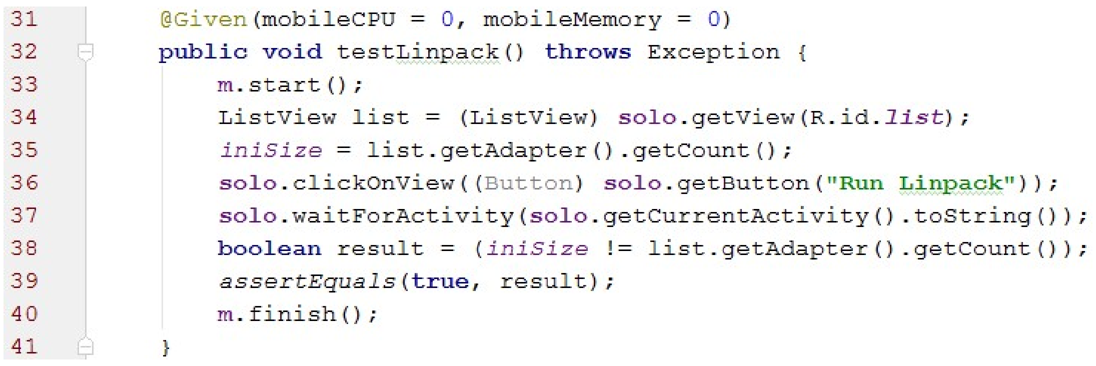
Required imports are `com.beftigre.band.Band`, `com.beftigre.band.Marker` and `com.beftigre.band.annotations.*`. During setup; *(i)* Band is initialised with activity (i.e. getActivity(), necessary for the test process) and test object (i.e. 'this', used to get annotations), *(ii)* power monitor is started and *(iii)* markers are registered.



```
    private Solo solo;
    private Band band;
    private Marker m = new Marker("Linpack");
    static int iniSize;

    public LinpackTest() { super(MainActivity.class); }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        solo = new Solo(getInstrumentation(), getActivity());
        band = new Band(getActivity(), this);
        band.startPowerMonitoring();
        band.registerMarkers(m);
    }
```

E.2 Band API Test method

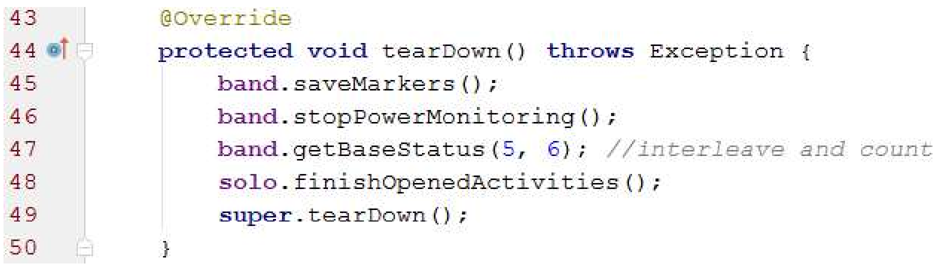
Notice that @Given annotation attributes are set to 0, this is for evaluation. For the case of comparison all annotations attributes will require a real value assigned to them.
Marker start and finish methods are called before and after the execution of the test, so as to capture the test process.



```
    @Given(mobileCPU = 0, mobileMemory = 0)
    public void testLinpack() throws Exception {
        m.start();
        ListView list = (ListView) solo.getView(R.id.list);
        iniSize = list.getAdapter().getCount();
        solo.clickOnView((Button) solo.getButton("Run Linpack"));
        solo.waitForActivity(solo.getCurrentActivity().toString());
        boolean result = (iniSize != list.getAdapter().getCount());
        assertEquals(true, result);
        m.finish();
    }
```

E.3 Band API TearDown method

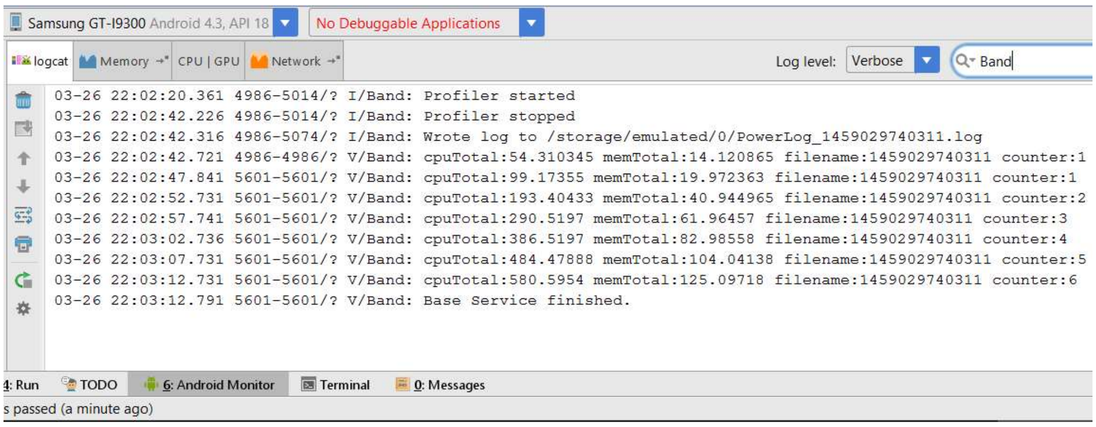
At teardown markers are saved, power monitor is stopped and the BaseService is started to obtain % CPU and memory availability of the mobile device.



```java
    @Override
    protected void tearDown() throws Exception {
        band.saveMarkers();
        band.stopPowerMonitoring();
        band.getBaseStatus(5, 6); //interleave and count
        solo.finishOpenedActivities();
        super.tearDown();
    }
```

E.4 Band API BaseService logcat output

After the test is passed/completed the BaseService runs (to compute % CPU and memory availability of the mobile device) until the count value (i.e. second argument) specified in getBaseStatus API call is completed. Note the first argument of getBaseStatus is interleave or interval – in seconds. A toast message is also sent by the API to the mobile when the BaseService is completed.



Log level: Verbose

Band

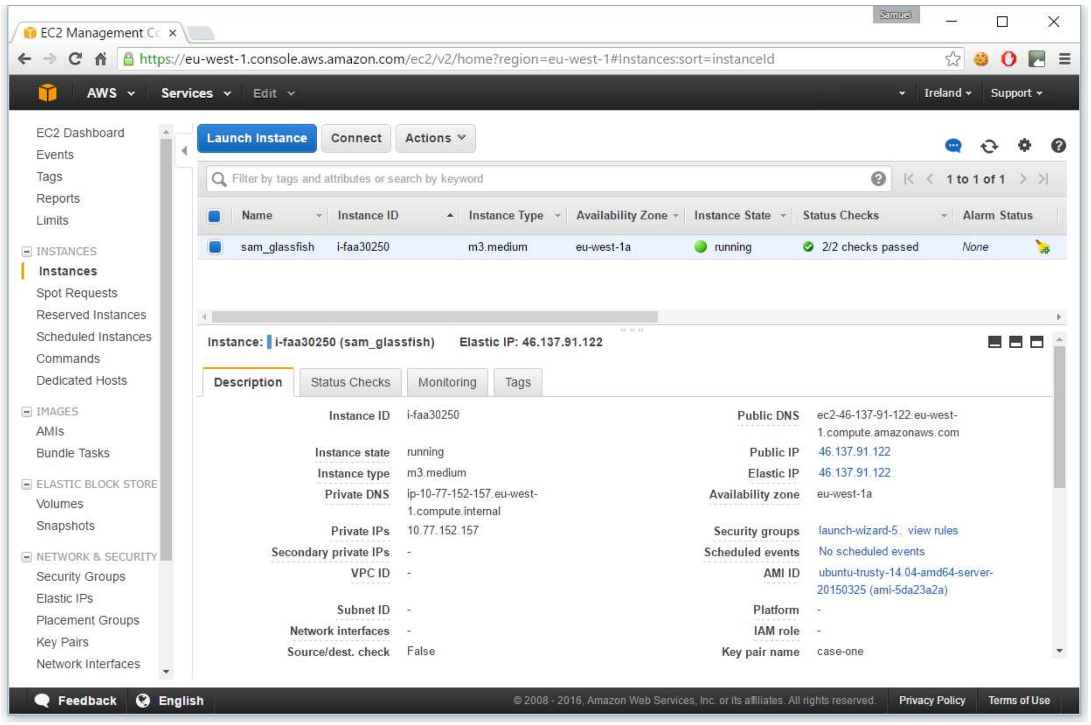
```
03-26 22:02:20.361 4986-5014/? I/Band: Profiler started
03-26 22:02:42.226 4986-5014/? I/Band: Profiler stopped
03-26 22:02:42.316 4986-5074/? I/Band: Wrote log to /storage/emulated/0/PowerLog_1459029740311.log
03-26 22:02:42.721 4986-4986/? V/Band: cpuTotal:54.310345 memTotal:14.120865 filename:1459029740311 counter:1
03-26 22:02:47.841 5601-5601/? V/Band: cpuTotal:99.17355 memTotal:19.972363 filename:1459029740311 counter:1
03-26 22:02:52.731 5601-5601/? V/Band: cpuTotal:193.40433 memTotal:40.944965 filename:1459029740311 counter:2
03-26 22:02:57.741 5601-5601/? V/Band: cpuTotal:290.5197 memTotal:61.96457 filename:1459029740311 counter:3
03-26 22:03:02.736 5601-5601/? V/Band: cpuTotal:386.5197 memTotal:82.98558 filename:1459029740311 counter:4
03-26 22:03:07.731 5601-5601/? V/Band: cpuTotal:484.47888 memTotal:104.04138 filename:1459029740311 counter:5
03-26 22:03:12.731 5601-5601/? V/Band: cpuTotal:580.5954 memTotal:125.09718 filename:1459029740311 counter:6
03-26 22:03:12.791 5601-5601/? V/Band: Base Service finished.
```

s passed (a minute ago)

Appendix F Beftigre BEFOR Tool Screenshots

F.1 EC2 Ubuntu instance setup

Beftigre has been tested on EC2. Notice the `case-one` key pair (pem) file, this is used to connect to Befor API.



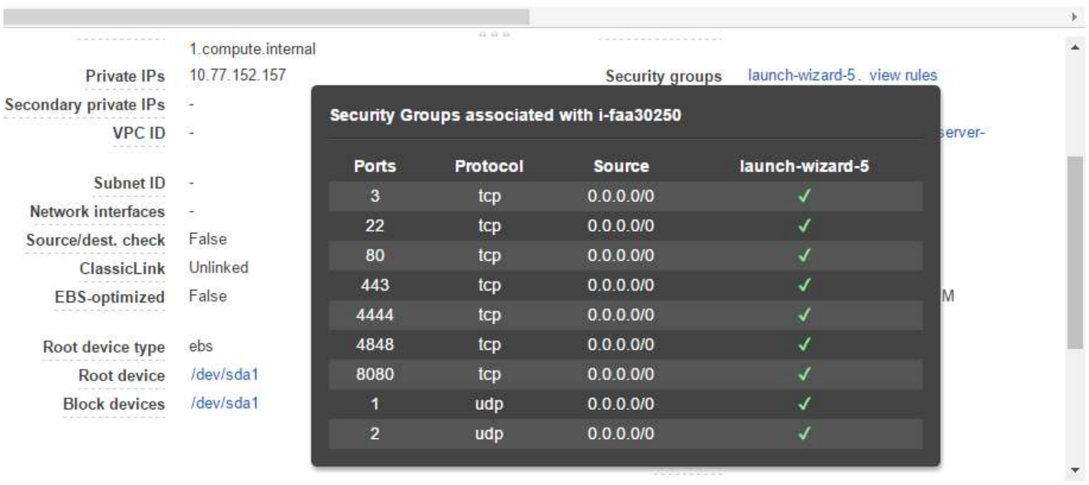
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status
sam_glassfish	i-faa30250	m3.medium	eu-west-1a	running	2/2 checks passed	None

Instance: i-faa30250 (sam_glassfish) Elastic IP: 46.137.91.122

Instance ID	i-faa30250	Public DNS	ec2-46-137-91-122.eu-west-1.compute.amazonaws.com
Instance state	running	Public IP	46.137.91.122
Instance type	m3.medium	Elastic IP	46.137.91.122
Private DNS	ip-10-77-152-157.eu-west-1.compute.internal	Availability zone	eu-west-1a
Private IPs	10.77.152.157	Security groups	launch-wizard-5. view rules
Secondary private IPs	-	Scheduled events	No scheduled events
VPC ID	-	AMI ID	ubuntu-trusty-14.04-amd64-server-20150325 (ami-5da23a2a)
Subnet ID	-	Platform	-
Network interfaces	-	IAM role	-
Source/dest. check	False	Key pair name	case-one

F.2 EC2 Security groups

Notice that ports 22, 8080, 4848, 1, 2, and 3 have been added in the security group for Beftigre test. These are the default ports for Beftigre test.



	1.compute.internal
Private IPs	10.77.152.157
Secondary private IPs	-
VPC ID	-
Subnet ID	-
Network interfaces	-
Source/dest. check	False
ClassicLink	Unlinked
EBS-optimized	False
Root device type	ebs
Root device	/dev/sda1
Block devices	/dev/sda1

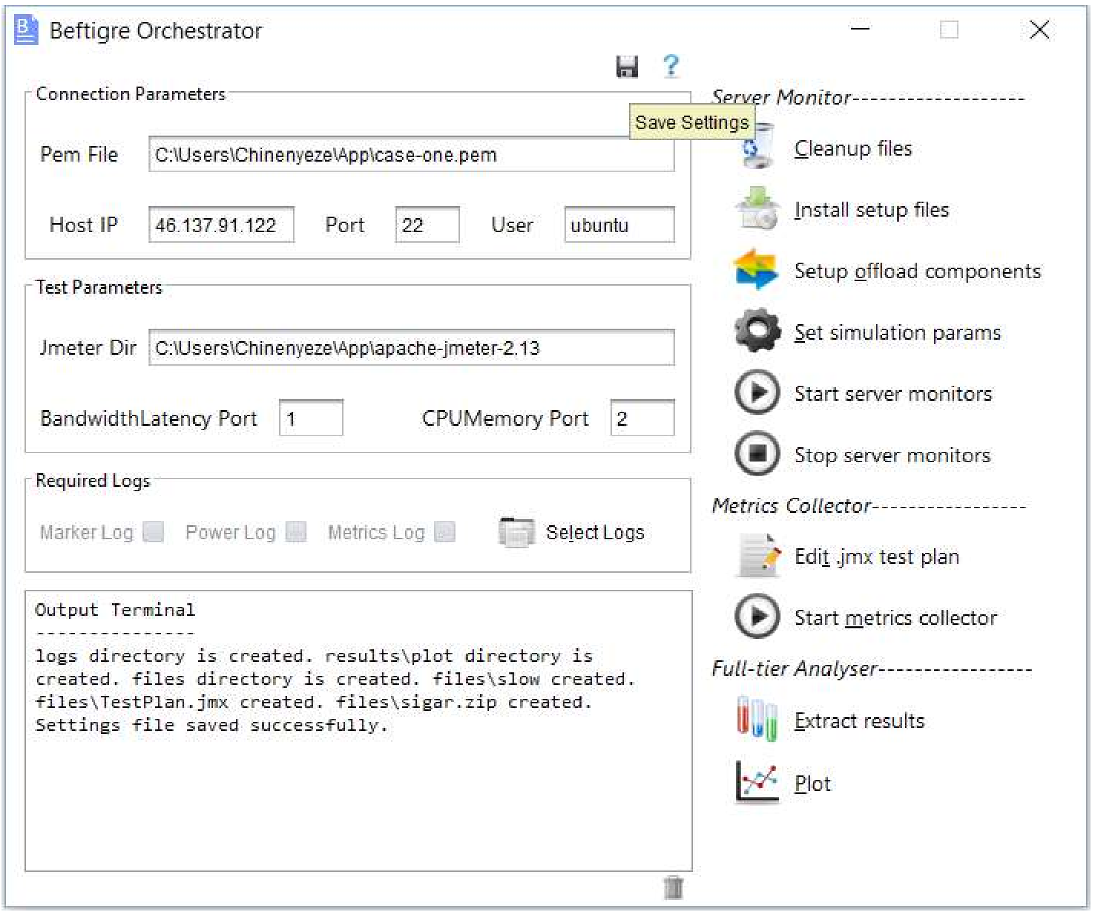
Security groups launch-wizard-5. view rules

Security Groups associated with i-faa30250

Ports	Protocol	Source	launch-wizard-5
3	tcp	0.0.0.0/0	✓
22	tcp	0.0.0.0/0	✓
80	tcp	0.0.0.0/0	✓
443	tcp	0.0.0.0/0	✓
4444	tcp	0.0.0.0/0	✓
4848	tcp	0.0.0.0/0	✓
8080	tcp	0.0.0.0/0	✓
1	udp	0.0.0.0/0	✓
2	udp	0.0.0.0/0	✓

F.3 Connection and Test Parameter settings

Notice `case-one.pem` file from EC2 setup, and ports 22, 1 and 2 already opened in security groups. Also achieved by `params` API command.



```
Output Terminal
---------------
logs directory is created. results\plot directory is
created. files directory is created. files\slow created.
files\TestPlan.jmx created. files\sigar.zip created.
Settings file saved successfully.
```

F.4 Install setup files

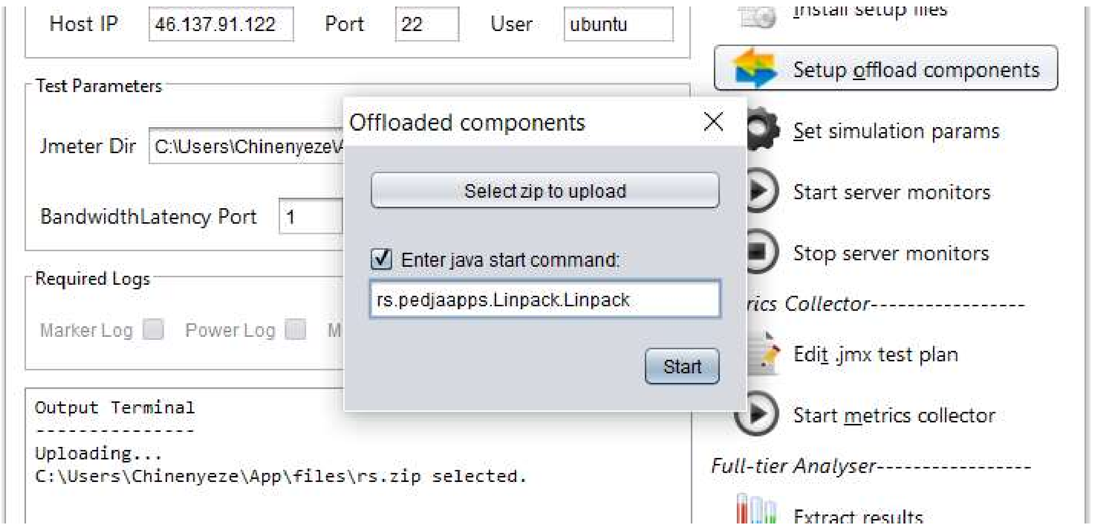
Below is the output highlighting the six files/programs used to setup the server for Beftigre test, as mentioned in Section 6.4.4.3. Also achieved by `setup` API command.

```
Output Terminal
---------------
BandwidthLatencyServer port: 1
CPUMemoryServer port: 2
files\BandwidthLatencyServer.java created.
files\CPUMemoryAvailServer.java created.
BandwidthLatencyServer.java, CPUMemoryAvailServer.java and sigar.zip copied to
server.
Archive:  sigar.zip
   creating: sigar/
  inflating: sigar/.sigar_shellrc
  inflating: sigar/libsigar-amd64-freebsd-6.so
  inflating: sigar/libsigar-amd64-linux.so
  inflating: sigar/libsigar-amd64-solaris.so
  inflating: sigar/libsigar-ia64-hpux-11.sl
  inflating: sigar/libsigar-ia64-linux.so
  inflating: sigar/libsigar-pa-hpux-11.sl
  inflating: sigar/libsigar-ppc-aix-5.so
  inflating: sigar/libsigar-ppc-linux.so
  inflating: sigar/libsigar-ppc64-aix-5.so
  inflating: sigar/libsigar-ppc64-linux.so
  inflating: sigar/libsigar-s390x-linux.so
  inflating: sigar/libsigar-sparc-solaris.so
  inflating: sigar/libsigar-sparc64-solaris.so
  inflating: sigar/libsigar-universal-macosx.dylib
  inflating: sigar/libsigar-x86-freebsd-5.so
  inflating: sigar/libsigar-x86-freebsd-6.so
  inflating: sigar/libsigar-x86-linux.so
  inflating: sigar/libsigar-x86-solaris.so
  inflating: sigar/log4j.jar
  inflating: sigar/sigar-amd64-winnt.dll
  inflating: sigar/sigar-x86-winnt.dll
  inflating: sigar/sigar-x86-winnt.lib
  inflating: sigar/sigar.jar
exit-status: 0
SIGAR API setup completed.
exit-status: 0
BandwidthLatencyServer and CPUMemoryAvailServer setup completed.
Remember to set up your offloadable component from the terminal.
Reading package lists...
Building dependency tree...
Reading state information...
unzip is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 171 not upgraded.
Archive:  ServerAgent-2.2.1.zip
  inflating: startAgent.sh
  inflating: startAgent.bat
   creating: lib/
  inflating: lib/libsigar-x86-freebsd-6.so
  inflating: lib/libsigar-pa-hpux-11.sl
 extracting: lib/sigar.jar
  inflating: lib/libsigar-s390x-linux.so
  inflating: lib/libsigar-x86-solaris.so
  inflating: lib/libsigar-ppc-aix-5.so
  inflating: lib/libsigar-ia64-hpux-11.sl
  inflating: lib/sigar-amd64-winnt.dll
  inflating: lib/libsigar-ppc64-aix-5.so
  inflating: lib/libsigar-ppc-linux.so
  inflating: lib/libsigar-universal-macosx.dylib
  inflating: lib/libsigar-amd64-solaris.so
  inflating: lib/libsigar-ppc64-linux.so
  inflating: lib/libsigar-sparc64-solaris.so
  inflating: lib/sigar-x86-winnt.dll
  inflating: lib/libsigar-ia64-linux.so
  inflating: lib/sigar-x86-winnt.lib
  inflating: lib/libsigar-x86-linux.so
  inflating: lib/libsigar-sparc-solaris.so
  inflating: lib/libsigar-amd64-linux.so
  inflating: lib/libsigar-x86-freebsd-5.so
 extracting: lib/log4j.jar
  inflating: lib/libsigar-universal64-macosx.dylib
  inflating: lib/libsigar-amd64-freebsd-6.so
 extracting: lib/cmdrunner-1.0.1.jar
 extracting: lib/jorphan-2.6.jar
 extracting: lib/logkit-2.0.jar
 extracting: lib/sigar-1.6.4.jar
 extracting: lib/avalon-framework-4.1.5.jar
 extracting: ServerAgent.jar
extracting: CMDRunner.jar
```

```
  inflating: LICENSE
Monitor setup completed.
exit-status: 0
Network Throttler (slow) setup completed.
Reading package lists...
Building dependency tree...
Reading state information...
The following NEW packages will be installed:
  stress
0 upgraded, 1 newly installed, 0 to remove and 171 not upgraded.
Need to get 0 B/17.0 kB of archives.
After this operation, 73.7 kB of additional disk space will be used.
Selecting previously unselected package stress.
(Reading database ... 52190 files and directories currently installed.)
Preparing to unpack .../stress_1.0.1-1ubuntu1_amd64.deb ...
Unpacking stress (1.0.1-1ubuntu1) ...
Processing triggers for install-info (5.2.0.dfsg.1-2) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Setting up stress (1.0.1-1ubuntu1) ...
Stress setup completed.
exit-status: 0
```

F.5 Setup offload components

Using Linpack android app as an example, the .class files of the offloadable compute-intensice component (rs.pedjaapps.Linpack.Linpack) has been zipped into rs.zip, alongside its dependency files (in this case rs.pedjaapps.Linpack.Result). The main method to start the program is within rs.pedjaapps.Linpack.Linpack. Thus, the offload setup is as shown in the screenshot below;



Host IP 46.137.91.122 Port 22 User ubuntu

Jmeter Dir C:\Users\ChinenyezeV

BandwidthLatency Port 1

Offloaded components

☑ Enter java start command: rs.pedjaapps.Linpack.Linpack

```
Output Terminal
---------------
Uploading...
C:\Users\Chinenyeze\App\files\rs.zip selected.
```

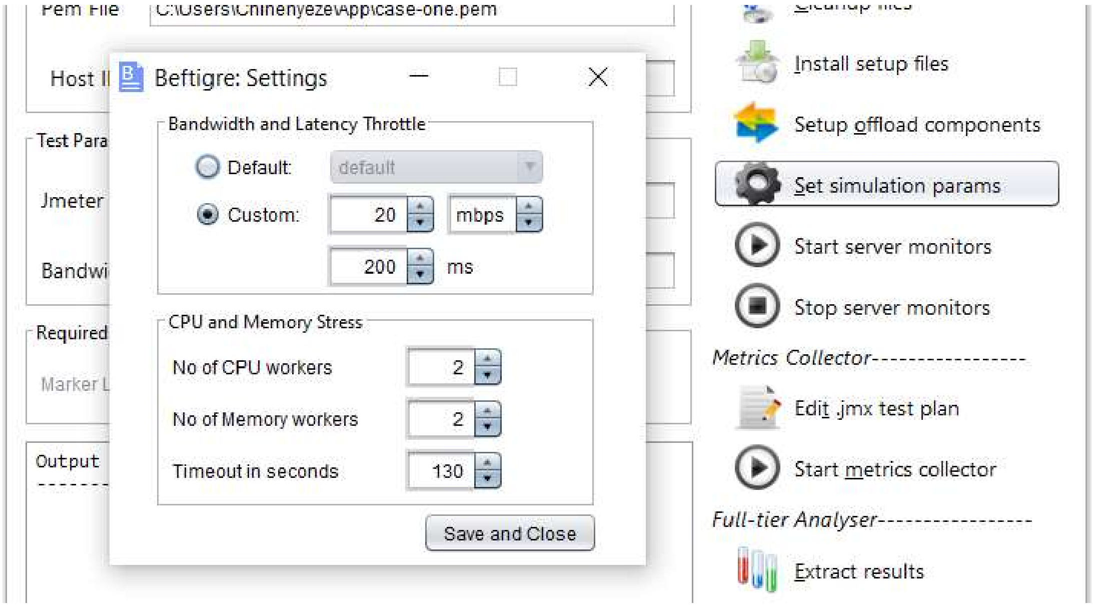
When start is clicked the following output is displayed: showing that the zip has been extracted on the server, and the program has been launched using the start command entered. Also achieved by `offload` API command. Note: `exit-status: 0` shown in the output signifies that the process or command was successfully executed at the server.

```
Output Terminal
---------------
Uploading...
C:\Users\Chinenyeze\App\files\rs.zip selected.
Uploading...
Components successfully setup.
Archive:  rs.zip
   creating: rs/
   creating: rs/pedjaapps/
   creating: rs/pedjaapps/Linpack/
  inflating: rs/pedjaapps/Linpack/Linpack.class
  inflating: rs/pedjaapps/Linpack/Result.class
exit-status: 0
Starting...
exit-status: 0
Start command successfully issued.
```

Note that when server monitor is stopped the offload components are also stopped – this is because stopping server monitoring process kills all java processes. To start the offload components again during a test, just enter only the start command without a zip (as the zip has already been uploaded the first time). Note: there could be a likely case where starting an application requires a library in the class path (used to compile), in that case, ensure that the library's jar(s) is/are uploaded in the zip alongside the classes, then run the application as follows: `-cp .:path/to/lib.jar mainclass` i.e. including class path in the start command.

F.6 Set simulation params

The screenshot below shows simulation parameters of 20mbps bandwidth, 200ms latency, 2 CPU and memory loads with 130s timeout. The parameters are saved in SimLog. Also achieved by `simulate` API command.



Pem File C:\Users\Chinenyeze\App\case-one.pem

Beftigre: Settings

Bandwidth and Latency Throttle
Default: default
Custom: 20 mbps
200 ms

CPU and Memory Stress
No of CPU workers 2
No of Memory workers 2
Timeout in seconds 130

Save and Close

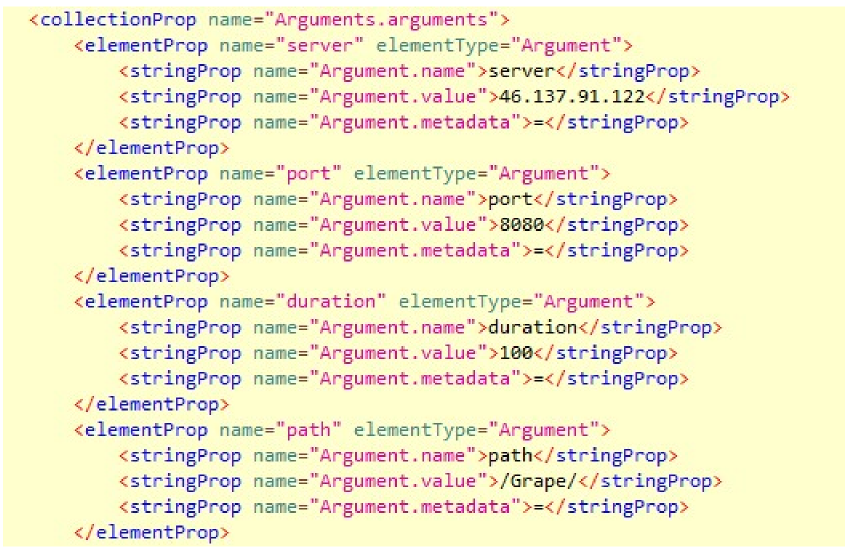
F.7 Start server monitors

The output below highlights the (three) server monitors started alongside stress and throttle utilities for simulations. Also achieved by `start` API command.

```
Output Terminal
---------------
Throttle type:custom bandwidth:20mbps latency:200ms
Stress: cpu, mem, time: 2 2 130
CPU and Memory stress started.
Stress will stop automatically after timeout
exit-status: 0
ServerAgent monitor started.
CPUMemoryAvailServer monitor started.
BandwidthLatencyServer monitor started.
Remember to start your offloadable component from the terminal.
param 20mbps
param 200ms
command=slow
bandwidth=20mbps
latency=200ms
Adding new queuing discipline
Throttler started.
exit-status: 0
SimLog created.
```

F.8 Edit .jmx test plan

This launches a text editor with the generated test plan template. Also achieved by `editplan` API command. The important aspect to edit in the test plan are presented below. Notice port 8080 is being used, and already opened in the security group. The server argument is same as the IP address used to connect Befor API. Users and rampup value can be left as 1. Refer to http://jmeter.apache.org/usermanual/test_plan.html for understanding terms in testplan. Another important argument is the path, which refers to any hosted html file or resource, which can be publicly accessed by jmeter for the test.



```
<collectionProp name="Arguments.arguments">
    <elementProp name="server" elementType="Argument">
        <stringProp name="Argument.name">server</stringProp>
        <stringProp name="Argument.value">46.137.91.122</stringProp>
        <stringProp name="Argument.metadata">=</stringProp>
    </elementProp>
    <elementProp name="port" elementType="Argument">
        <stringProp name="Argument.name">port</stringProp>
        <stringProp name="Argument.value">8080</stringProp>
        <stringProp name="Argument.metadata">=</stringProp>
    </elementProp>
    <elementProp name="duration" elementType="Argument">
        <stringProp name="Argument.name">duration</stringProp>
        <stringProp name="Argument.value">100</stringProp>
        <stringProp name="Argument.metadata">=</stringProp>
    </elementProp>
    <elementProp name="path" elementType="Argument">
        <stringProp name="Argument.name">path</stringProp>
        <stringProp name="Argument.value">/Grape/</stringProp>
        <stringProp name="Argument.metadata">=</stringProp>
    </elementProp>
```

```
        <elementProp name="users" elementType="Argument">
            <stringProp name="Argument.name">users</stringProp>
            <stringProp name="Argument.value">1</stringProp>
            <stringProp name="Argument.metadata">=</stringProp>
        </elementProp>
        <elementProp name="rampup" elementType="Argument">
            <stringProp name="Argument.name">rampup</stringProp>
            <stringProp name="Argument.value">1</stringProp>
            <stringProp name="Argument.metadata">=</stringProp>
        </elementProp>
    </collectionProp>
```

F.9 Start metrics collector

This begins metrics collection for the amount of time in seconds, specified by the `duration` argument within the test plan. Thus, metrics collection automatically stops after the time elapses – after which it is then adequate to stop the server monitor if wished to. The collected metrics are saved in MetricsLog. Also achieved by `collect` API command.

Notice from the output below that the Socket clients (`BandwidthLatencyClient` & `CPUMemoryClient`) are first used to retrieve the bandwidth, latency, %CPU and memory availability from the server, prior to the jmeter test – which then begins metrics collection based on %CPU and memory usage.

```
Output Terminal
---------------
BandwidthLatencyClient port: 1
CPUMemoryClient port: 2
MetricsLog created.
Bandwidth and Latency received.
%CPU and %Memory avail. received.
Test started.
Test finished.
You could stop the Monitor if you wish.
```

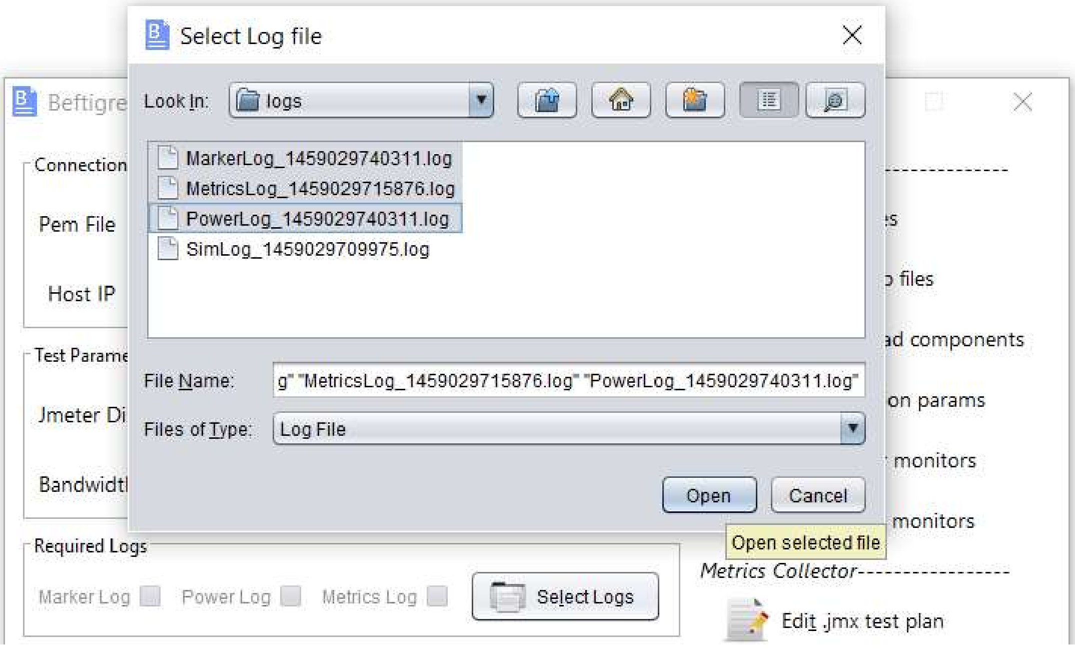
F.10 Stop server monitor

This stops the ServerAgent monitor, Socket monitors (i.e. BandwidthLatencyServer and CPUMemoryAvailServer), and throttle utility. The stress utility is automatically stopped after the specified timeout (as shown in *Set simulation params* section). Also achieved by `stop` API command.

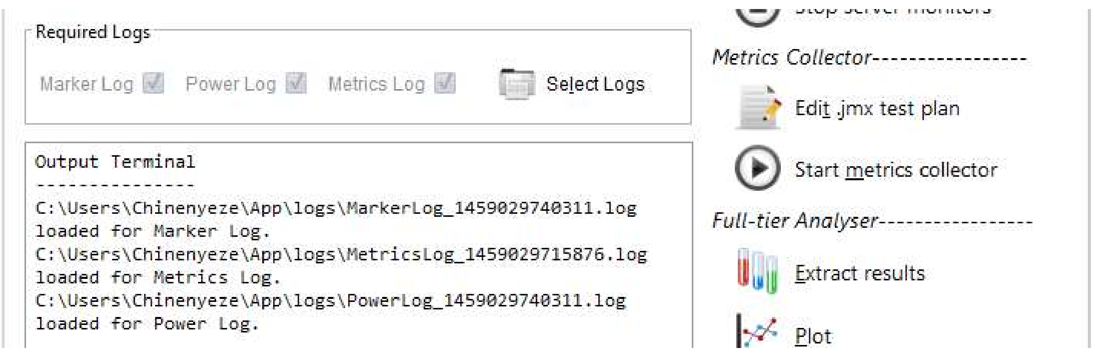
```
Output Terminal
---------------
ServerAgent monitor stopped.
param clear
command=clear
bandwidth=100kbps
latency=350ms
resetting queueing discipline
Throttler stopped.
Socket monitors stopped
exit-status: 0
```

F.11 Extract results

This extracts results (dat and csv files) from logs. Also achieved by `extract` API command. First MarkerLog, MetricsLog and PowerLog have to be selected, as below

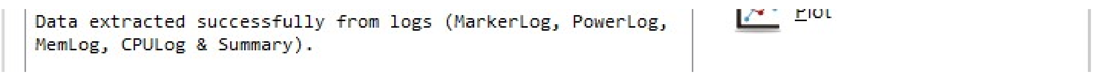


When the right required logs are selected and opened, the checkboxes for the 'Required Logs' panel are checked, as below;



```
Output Terminal
---------------
C:\Users\Chinenyeze\App\logs\MarkerLog_1459029740311.log
loaded for Marker Log.
C:\Users\Chinenyeze\App\logs\MetricsLog_1459029715876.log
loaded for Metrics Log.
C:\Users\Chinenyeze\App\logs\PowerLog_1459029740311.log
loaded for Power Log.
```

When the 'Extract results' button is clicked, the data files (CPULog.dat, MarkerLog.dat, MemLog.dat and PowerLog.dat) and results summary file (summary.csv) are extracted into results directory.

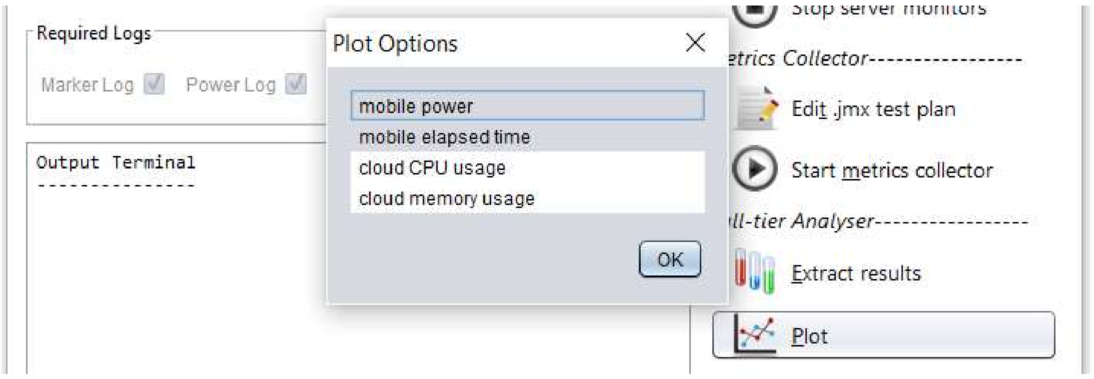


```
Data extracted successfully from logs (MarkerLog, PowerLog,
MemLog, CPULog & Summary).
```

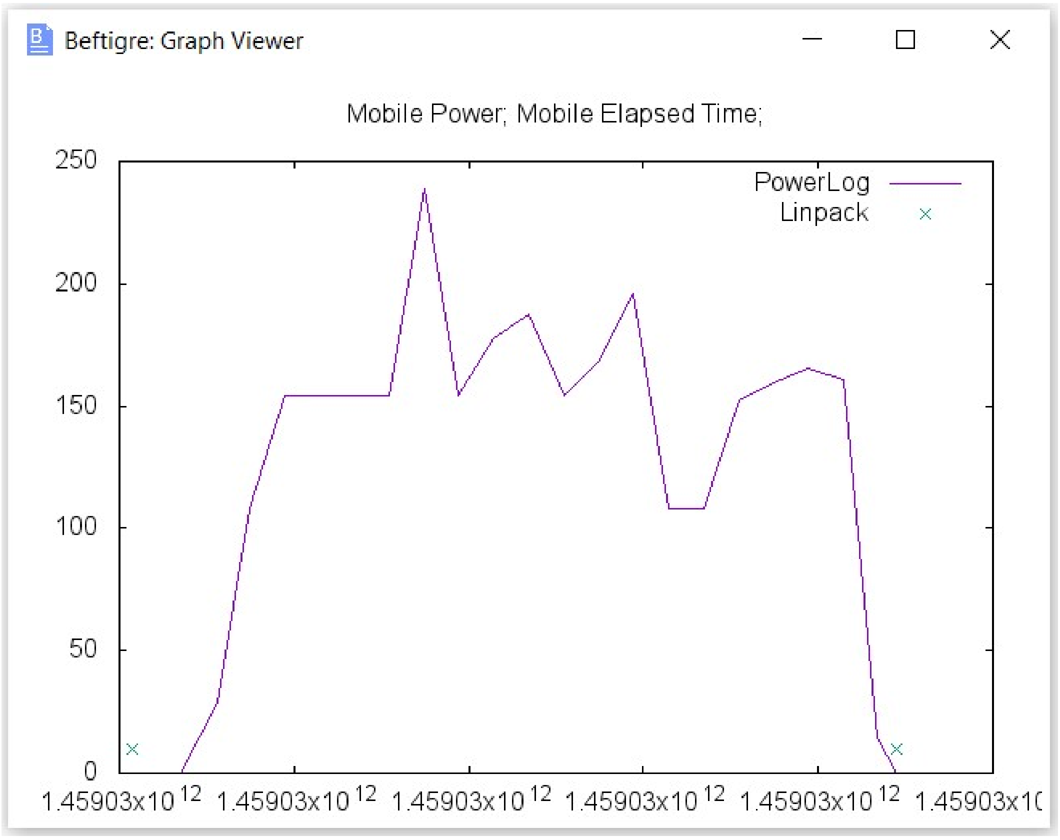
Note that SimLog is not required for computing results, it only used to know which simulation parameters achieved a result, for repeatability of test.

F.12 Plot

This plots graph from extracted (data) files. Also achieved by `plot` API command. Clicking ‘Plot’ prompts to select the data files to plot.



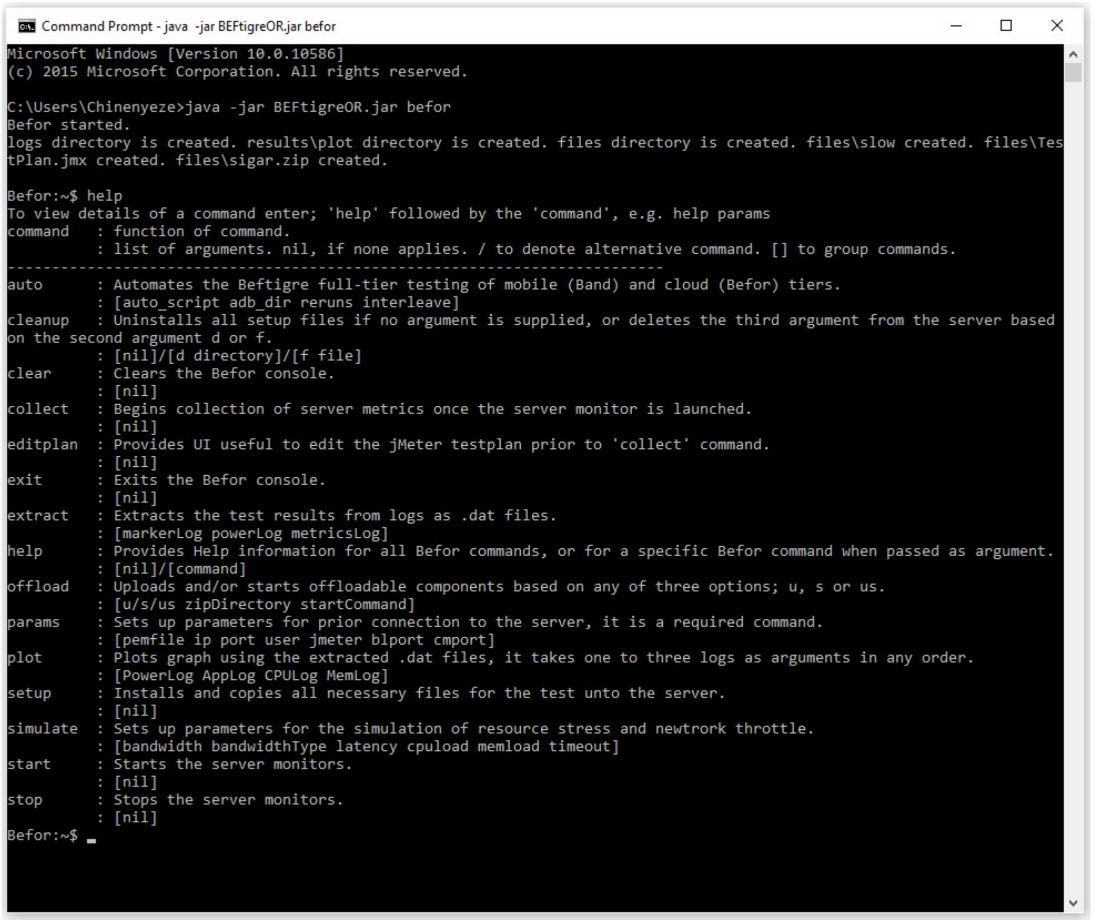
Then, click OK to plot.



The sample graph above shows the plotting for distinct used power (i.e. power at different timestamp, from PowerLog.dat file) and the elapsed time (i.e. start and finish timestamp, from MarkerLog.dat) of the test.

Appendix G   Beftigre BEFOR API Commands

This appendix presents a list of all Befor API commands and how they are used. Within the API, this information can be obtained using `help` command.



```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Chinenyeze>java -jar BEFtigreOR.jar befor
Befor started.
logs directory is created. results\plot directory is created. files directory is created. files\slow created. files\Tes
tPlan.jmx created. files\sigar.zip created.

Befor:~$ help
To view details of a command enter; 'help' followed by the 'command', e.g. help params
command    : function of command.
           : list of arguments. nil, if none applies. / to denote alternative command. [] to group commands.
--------------------------------------------------------------------------
auto       : Automates the Beftigre full-tier testing of mobile (Band) and cloud (Befor) tiers.
           : [auto_script adb_dir reruns interleave]
cleanup    : Uninstalls all setup files if no argument is supplied, or deletes the third argument from the server based
on the second argument d or f.
           : [nil]/[d directory]/[f file]
clear      : Clears the Befor console.
           : [nil]
collect    : Begins collection of server metrics once the server monitor is launched.
           : [nil]
editplan   : Provides UI useful to edit the jMeter testplan prior to 'collect' command.
           : [nil]
exit       : Exits the Befor console.
           : [nil]
extract    : Extracts the test results from logs as .dat files.
           : [markerLog powerLog metricsLog]
help       : Provides Help information for all Befor commands, or for a specific Befor command when passed as argument.
           : [nil]/[command]
offload    : Uploads and/or starts offloadable components based on any of three options; u, s or us.
           : [u/s/us zipDirectory startCommand]
params     : Sets up parameters for prior connection to the server, it is a required command.
           : [pemfile ip port user jmeter blport cmport]
plot       : Plots graph using the extracted .dat files, it takes one to three logs as arguments in any order.
           : [PowerLog AppLog CPULog MemLog]
setup      : Installs and copies all necessary files for the test unto the server.
           : [nil]
simulate   : Sets up parameters for the simulation of resource stress and newtrork throttle.
           : [bandwidth bandwidthType latency cpuload memload timeout]
start      : Starts the server monitors.
           : [nil]
stop       : Stops the server monitors.
           : [nil]
Befor:~$ _
```

The commands without argument are presented in the table below.

Command	*Function of the command*
`clear`	Clears the Befor console.
`collect`	Begins collection of server metrics once the server monitor is launched.
`editplan`	Provides UI useful to edit the jMeter testplan prior to 'collect' command.
`exit`	Exits the Befor console.
`setup`	Installs and copies all necessary files for the test unto the server.
`start`	Starts the server monitors.
`stop`	Stops the server monitors.
`help`	Provides Help information for all Befor commands, or for a specific Befor command when passed as argument. E.g. `help setup`

- **auto** Command

Automates the Beftigre full-tier testing of mobile (Band) and cloud (Befor) tiers. The command usage have been presented in section . Presented below is the output from auto command execution;

```
Befor started.
logs directory found. results\plot directory found. files directory found.
files\slow already exist. files\TestPlan.jmx already exist. files\sigar.zip
already exist.
Befor:~$           auto           "C:\Users\Chinenyeze\App\files\script.auto"
"C:\Users\Chinenyeze\AppData\Local\Android\sdk\platform-tools" 1 40
The auto script file was not found.
Befor:~$                                                                  auto
"C:\Users\Chinenyeze\Documents\NetBeansProjects\BEFtigreOR\files\script.auto"
"C:\Users\Chinenyeze\AppData\Local\Android\sdk\platform-tools" 1 40
params initialised.
offload initialised.
simulate added to list.
am initialised.

***Automated test started.
params command completed.
exit-status: 0
offload command completed.
SimLog created.
simulate command completed.
CPU and Memory stress started.
Stress will stop automatically after timeout
exit-status: 0
ServerAgent monitor started.
CPUMemoryAvailServer monitor started.
BandwidthLatencyServer monitor started.
Remember to start your offloadable component from the terminal.
param 200mbps
param 180ms
command=slow
bandwidth=200mbps
latency=180ms
Adding new queuing discipline
Throttler started.
exit-status: 0
start command completed.
621 219
ServerCPU:41 ServerMem:74
MetricsLog created.
collect command completed.
WARN    2016-04-09 16:17:04.948 [jmeter.u] (): Unexpected value set for boolean
property:'server.exitaftertest', defaulting to:false
WARN    2016-04-09 16:17:04.995 [jmeter.u] (): Unexpected value set for boolean
property:'jmeterengine.startlistenerslater', defaulting to:true
INFO    2016-04-09 16:17:04.995 [jmeter.e] (): Listeners will be started after
enabling running version
INFO   2016-04-09 16:17:04.995 [jmeter.e] (): To revert to the earlier behaviour,
define jmeterengine.startlistenerslater=false
WARN    2016-04-09 16:17:04.995 [jmeter.u] (): Unexpected value set for boolean
property:'jmeterengine.remote.system.exit', defaulting to:false
WARN    2016-04-09 16:17:04.995 [jmeter.u] (): Unexpected value set for boolean
property:'jmeterengine.stopfail.system.exit', defaulting to:true
WARN    2016-04-09 16:17:04.995 [jmeter.u] (): Unexpected value set for boolean
property:'jmeterengine.force.system.exit', defaulting to:false
```

```
JMeter test started for Metrics collector.
rs.pedjaapps.Linpack.LinpackTest:.
Test results for InstrumentationTestRunner=.
Time: 23.475
OK (1 test)
am command completed.
ServerAgent monitor stopped.
param clear
command=clear
bandwidth=100kbps
latency=350ms
resetting queueing discipline
Throttler stopped.
Socket monitors stopped
exit-status: 0
stop command completed.
***Automated test completed.
```

- **`cleanup`** Command

Uninstalls all setup files if no argument is supplied, or deletes the third argument from the server based on the second argument d or f.

`cleanup`	**`cleanup`** `-d directory`	**`cleanup`** `-f file`

`-d` directory	– the directory to be deleted from the server.
`-f` file	– the file with extension to be deleted from the server, e.g. Sample.java.

- **`extract`** Command

Extracts the test results from logs as .dat files.

`extract` `markerLog powerLog metricsLog`

markerLog	– the absolute file name of MarkerLog_123.log, e.g. "C:\App\logs\MarkerLog_123.log"
powerLog	– the absolute file name of PowerLog_123.log, e.g. "C:\App\logs\PowerLog_123.log"
metricsLog	– the absolute file name of MetricsLog_123.log, e.g. "C:\App\logs\MetricsLog_123.log"

The .log files are generated with timestamps appended to their file names. 123 above represents the timestamp.

- **`offload`** Command

Uploads and/or starts offloadable components based on any of three options; u, s or us.

`offload` `-u zipfile`	**`offload`** `-s mainclass`	**`offload`** `-us zipfile mainclass`

`-u`	– signifies an upload, expecting the following argument to be a zip file.
`-s`	– signifies a start, expecting the following argument to be a start command.
`-us`	– combines the functionality of `-u` and `-s`.
zipfile	– the zip file to upload, which gets extracted at the server, e.g. "C:\App\zipfile.zip"

mainclass – the class name used to start an offloaded component by java interpreter; this must include the name of the package too, as per standard programming convention.

Note: there could be a likely case where starting an application requires a library in the class path (used to compile), in that case, ensure that the library's jar(s) is/are uploaded in the zip alongside the classes, then run the application as follows; i.e. including class path in the start command and putting the start command in quotes:

```
offload -s "-cp .:path/to/lib.jar mainclass"
offload -us zipfile "-cp .:path/to/lib.jar mainclass"
```

- `params` Command

Sets up parameters for prior connection to the server. It is a required command, and takes 7 arguments in the specified order.

```
params pemfile ip port user jmeter blport cmport
```

pemfile	– the .pem file from EC2 server setup.
ip	– the ip address of the server.
port	– the port number for the server connection.
user	– the server registered user.
jmeter	– the absolute path of Apache JMeter home directory, e.g. " C:\App\jmeter"
blport	– the port number for BandwidthLatencyServer and Client.
cmport	– the port number for CPUMemoryServer and Client.

- `plot` Command

Plots graph using the extracted .dat files, it takes one to three logs as arguments in any order.

```
plot PowerLog | plot PowerLog AppLog | plot PowerLog AppLog CPULog MemLog
```

PowerLog	– the absolute file name of PowerLog.dat, e.g. "C:\App\results\PowerLog.dat"
AppLog	– the absolute file name of AppLog.dat, e.g. "C:\App\results\AppLog.dat"
CPULog	– the absolute file name of CPULog.dat, e.g. "C:\App\results\CPULog.dat"
MemLog	– the absolute file name of MemLog.dat, e.g. "C:\App\results\MemLog.dat"

- `simulate` Command

Sets up parameters for the simulation of resource stress and network throttle.

```
simulate bandwidth bandwidthType latency cpuload memload timeout
```

bandwidth	– an integer representing the bandwidth.
bandwidthType	– the bandwidth unit type, e.g. bps, kbps or mbps.
latency	– an integer representing the latency in ms.
cpuload	– an integer representing the cpu load.
memload	– an integer representing the memory load.
timeout	– an integer representing the timeout in s for cpu and memory load.

- **`auto`: Automating the Full-tier Test**

`auto` is the Beftigre framework's test automation command which is used to automate the Beftigre full-tier testing of mobile (Band) and cloud (Befor) tiers. This makes it easy to repeat experiments on the Beftigre Framework (The output logs and data files from Beftigre are presented in Appendix H). As shown in the snippet below, the test automation is initiated by calling the auto command of Befor API with the following three required arguments, and an optional fourth;

```
auto auto_script adb_dir reruns interleave
auto "C:\script.auto" "C:\path\to\Android\sdk\platform-tools" 4 40
```

- first argument: the auto script file (.auto)
- second argument: the full path to adb.exe (i.e. Android Debug Bridge)
- third argument: the number of reruns of the experiment
- fourth argument: the interleave (in seconds) between reruns

The purpose of the interleave argument is to allow the BaseService of Band API to complete execution – as this is necessary for full-tier evaluation.

An auto script file must specify commands useful for full-tier test. The format is given below; *(See Appendix G for details on Befor API commands)*. Since the file is for full-tier test, the `auto` command only supports five Befor commands relevant for testing the cloud tier; `params`, `offload`, `simulate`, `start`, `collect`, and `stop`. The `am` command is used with the adb.exe to launch the test on the mobile tier.

```
params pemfile ip port user jmeter blport cmport
offload -s mainclass
simulate bandwidth bandwidthType latency cpuload memload timeout
simulate bandwidth bandwidthType latency cpuload memload timeout
simulate bandwidth bandwidthType latency cpuload memload timeout
start
collect
am instrument -w -e class rs.pedjaapps.Linpack.LinpackTest
rs.pedjaapps.Linpack.test/android.test.InstrumentationTestRunner
stop
```

Figure 6.2 `auto` Script File

The required commands for constructing the script file to execute `auto` command are `params`, `offload`, `simulate` and `am`. One or more lines of `simulate` can be provided. `start`, `collect`, and `stop` are optional. As they do not require any argument they are automatically handled by `auto` command in Befor API.

```
Algorithm auto execution algorithm
Require: auto_script, adb_dir, reruns and interleave
 1: read auto_script file
 2: list ← load simulate commands into array list
 3: run params command
 4: if no interleave or interleave < 30 or interleave > 180 then
 5:     interleave = 30
 6: endif
 7: counter = 0
 8: for i=0; i < reruns; i++ do
 9:     run offload start command
10:     run list.get(counter)
11:     run start command
12:     wait (10)
13:     run collect command
14:     run am command using adb.exe at adb_dir
15:     wait (10)
16:     run stop command
17:     wait (interleave)
18:     counter++
19:     if counter >= list.size( ) then
20:         counter = 0
21:     endif
22: endfor
```

Algorithm 6.3 presents the execution procedure of auto command. Notice from the sample script file (Figure 6.2) that simulate command is parsed thrice (Lines 3-5 of Figure 6.2), this implies that the reruns of the experiment will be performed based on the given simulate commands. This is achieved (within Befor API) by sequentially looping through a list of simulate commands for each run (Line 10 of Algorithm 6.3). When all simulate commands are looped through but reruns are not completed then the system will restart simulation from the top of the list (Line 20 of Algorithm 6.3). The offload start command (Line 2 of Figure 6.2) is required in auto script to launch the offloadable component at the server. auto script, can be killed at any point using Ctrl+C. The interleave argument of auto command is optional, and defaults to 30 seconds. Also 30 seconds interleave is used if the provided interleave is below the default or above 180 max set threshold (Lines 4-6 of Algorithm 6.3). The 10 seconds wait at Lines 12 and 15 (in Algorithm 6.3) are used to ensure that the start and am commands are completed before the metrics collection and stop command respectively. The first wait (Line 12) is important as the collect command runs the socket clients, which requires socket server monitors to be already running and listening (on specified ports in params command). Similarly, the second wait (Line 15) ensures that the metrics collection does not overlap the mobile device test (am command) during completion – to ensure accuracy of readings.

How to obtain the right am command:

`auto` requires that the test project is already installed on the target devices prior to running the test. This is a prerequisite for executing android test command line.

The application project and test project can be installed on first execution from Android studio (command line option alternatives here []). To check that a device is connected use `adb devices` command. Ensure that the command line directory is changed to the adb location first, e.g.

```
cd C:\Users\Chinenyeze\AppData\Local\Android\sdk\platform-tools
```

Then enter

```
adb shell pm list instrumentation
```

the above command gives a directive of the test projects installed on the connected device, in the format below;

```
instrumentation:rs.pedjaapps.Linpack.test/android.test.InstrumentationTe
stRunner
(target= rs.pedjaapps.Linpack)
```

From the above output the instrumentation points to <test package>/<runner class> and the target specifies the <application package> of the installed app to be evaluated. [118] provides further useful adb documetation.

Given that the class of the test code is rs.pedjaapps.Linpack.LinpackTest, then the `am` command for `auto` script in Befor API can be constructed as follows;

```
am instrument -w -e class <test code class> <test package>/<runner
class>
```

```
am instrument -w -e class rs.pedjaapps.Linpack.LinpackTest
rs.pedjaapps.Linpack.test/android.test.InstrumentationTestRunner
```

Appendix H Beftigre Logs and Data Files

H.1 MarkerLog_123.log

The first part gives the package name of the application under test. The second part gives the values of marker objects. The third part gives the values for the BaseService process.

```
app rs.pedjaapps.Linpack                                      1

M1_label Linpack
M1_start 1459029740369
M1_finish 1459029762226                                       2
M1_anno na

mobileCPU 96.7659
mobileMemory 20.849531                                        3
```

H.2 PowerLog_123.log

The first part gives a listing of the mobile device settings. The second part gives the process IDs of the running applications – which is used to identify the resources used by a process. The third part gives the consumption values of the resources used by different processes.

```
phone-service in-service
phone-network HSDPA
batt_temp 29.5                                                1
batt_charge 8.28
LCD-brightness 255
...

associate 10061 rs.pedjaapps.Linpack@3
associate 10065 com.google.android.music@2513                 2
...

begin 1 1459029741736
total-power 1086
LCD-10356 900
CPU-freq 1134.0
CPU-10061 221
CPU-10029 0
...

begin 2 1459029742802                                         3
total-power 1152
LCD-10356 900
CPU-freq 1728.0
CPU-10061 232
CPU-10029 0
...

begin n [timestamp]
...
```

H.3 MetricsLog_123.log

The first part gives the bandwidth, latency and % cloud CPU and memory availability – actual values for Where clause. The second part gives the % cloud CPU and memory usage, collected by PerfMon Metrics Collector from PerfMon Server Agent, this is the actual values for the cloud tier of Then clause.

```
bandwidth 571
latency 238                                                         1
cloudCPU 43
cloudMemory 73
```

```
2016/03/26 22:01:58.334,26145,46.137.91.122 Memory,,,,,true,0,0,0,0
2016/03/26 22:01:58.351,57142,46.137.91.122 CPU,,,,,true,0,0,0,0
2016/03/26 22:01:59.353,18973,46.137.91.122 Memory,,,,,tr 2 0,0,0,0
2016/03/26 22:01:59.354,58333,46.137.91.122 CPU,,,,,true,0,0,0,0
...
```

H.4 SimLog_123.log

This is the log of the simulation parameters. The log is to inform the parameters that generated a particular results – for reproducibility of test.

```
bandwidth 20mbps
latency 200ms
cpuload 2
memoryload 2
```

H.5 MarkerLog.dat

This is the data file obtained from MarkerLog.log and gives the mobile start and finish timestamp of the test – used to calculate the mobile elapsed time.

```
# Label   Start         Finish
Linpack 1459029740369 1459029762226
```

H.6 PowerLog.dat

This is the data file obtained from an analysis on PowerLog.log and MarkerLog.log. It gives the mobile start and finish timestamp and the trailing power readings associated between these timestamps – used to calculate the mobile used energy.

```
# Timestamp   Power
1459029740369 0.0
1459029742802 28.378378378378375
...
1459029761713 14.594594594594595
1459029762226 0.0
```

H.7 CPULog.dat

This is the data file obtained from an analysis on MetricsLog.log and MarkerLog.log. It gives the mobile start and finish timestamp and the trailing cloud CPU usage readings associated between these timestamps – used to calculate the cloud used CPU.

```
# Timestamp   CPU
1459029740369 0
1459029741369 58
1459029742369 58
...
1459029760384 60
1459029761384 57
1459029762226 0
```

H.8 MemLog.dat

This is the data file obtained from an analysis on MetricsLog.log and MarkerLog.log. It gives the mobile start and finish timestamp and the trailing cloud memory usage readings associated between these timestamps – used to calculate the cloud used Memory.

```
# Timestamp   Memory
1459029740369 0
1459029741369 26
1459029742369 16
...
1459029760384 26
1459029761384 18
1459029762226 0
```

H.9 Summary.csv

This is a summary file computed from all data files. The sample below gives the result for evaluation (not comparison). Comparison output has been presented with case studies.

Marker	Label	MobileCPU	MobileMemory	Bandwidth	Latency	CloudCPU	CloudMemory	mElapsedTime(ms)	mUsedEnergy(mJ)	cUsedCPU(%)	cUsedMemory(%)	FinalAssert
S1-F1	Linpack	97	21	571	238	43	73	21857	2899.46	58	21	-

*The .log files are generated with timestamps appended to their file names. 123 appended to the filenames of .log files represents timestamps.

Appendix I Case Studies Test Classes

I.1 Linpack Test

```java
package rs.pedjaapps.Linpack;

import android.test.ActivityInstrumentationTestCase2;
import android.widget.Button;
import com.beftigre.band.Band;
import com.beftigre.band.Marker;
import com.beftigre.band.annotations.Given;
import com.robotium.solo.Solo;

public class LinpackTest extends ActivityInstrumentationTestCase2 {
    private Solo solo;
    private Band band;
    private Marker m = new Marker("Linpack");

    public LinpackTest() {
        super(MainActivity.class);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        solo = new Solo(getInstrumentation(), getActivity());
        band = new Band(getActivity(), this);
        band.startPowerMonitoring();
        band.registerMarkers(m);
    }

    @Given(mobileCPU = 0, mobileMemory = 0)
    public void testLinpack() throws Exception {
        m.start();
        solo.clickOnView((Button) solo.getButton("Run Linpack"));
        boolean result = solo.waitForText("Callee completed.");
        assertEquals(true, result);
        m.finish();
    }

    @Override
    protected void tearDown() throws Exception {
        band.saveMarkers();
        band.stopPowerMonitoring();
        band.getBaseStatus(5, 6); //interleave and count
        solo.finishOpenedActivities();
        super.tearDown();
    }
}
```

I.2 MatCalc Test

```java
package com.android.matcalc;

import android.test.ActivityInstrumentationTestCase2;
import android.widget.EditText;
import com.beftigre.band.Band;
import com.beftigre.band.Marker;
import com.beftigre.band.annotations.Given;
import com.cong89.matcalc.R;
import com.robotium.solo.Solo;

public class MatCalcTest extends ActivityInstrumentationTestCase2 {
    private Solo solo;
    private Band band;
    private Marker m = new Marker("MatCalc");

    public MatCalcTest() {
        super(MainActivity.class);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        solo = new Solo(getInstrumentation(), getActivity());
        band = new Band(getActivity(), this);
        band.startPowerMonitoring();
        band.registerMarkers(m);
    }

    @Given(mobileCPU = 0, mobileMemory = 0)
    public void testMatCalc() throws Exception {
        m.start();
        solo.enterText((EditText) solo.getView(R.id.matrixA), "1,2,3\n4,5,6\n7,8,0");
        solo.enterText((EditText) solo.getView(R.id.matrixB), "0.5\n2\n8");
        solo.clickOnButton("AB");
        boolean result = solo.waitForText("28.5\n60\n19.5");
        assertEquals(true, result);
        m.finish();
    }

    @Override
    protected void tearDown() throws Exception {
        band.saveMarkers();
        band.stopPowerMonitoring();
        band.getBaseStatus(5, 6); //interleave and count
        solo.finishOpenedActivities();
        super.tearDown();
    }
}
```

I.3 MathDroid Test

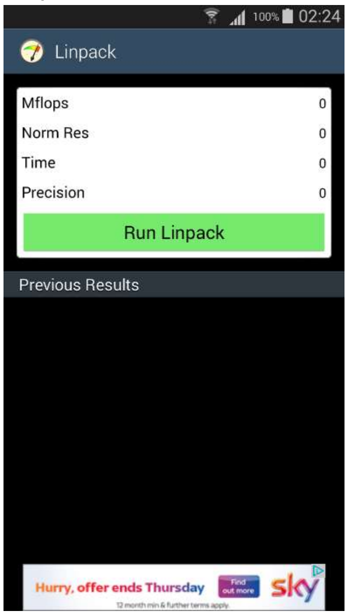
```java
package org.jessies.mathdroid;

import android.test.ActivityInstrumentationTestCase2;
import android.widget.Button;
import com.beftigre.band.Band;
import com.beftigre.band.Marker;
import com.beftigre.band.annotations.Given;
import com.robotium.solo.Solo;

public class MathdroidTest extends ActivityInstrumentationTestCase2 {
    private Solo solo;
    private Band band;
    private Marker m = new Marker("Mathdroid");

    public MathdroidTest() {
        super(Mathdroid.class);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        solo = new Solo(getInstrumentation(), getActivity());
        band = new Band(getActivity(), this);
        band.startPowerMonitoring();
        band.registerMarkers(m);
    }

    @Given(mobileCPU = 0, mobileMemory = 0)
    public void testMathdroid() throws Exception {
        m.start();
        solo.clickOnView(solo.getView(R.id.menu_clear));
        solo.clickOnView((Button) solo.getButton("3"));
        solo.clickOnView((Button) solo.getView(R.id.times));
        solo.clickOnView((Button) solo.getButton("7"));
        solo.clickOnView((Button) solo.getView(R.id.exe));
        boolean result = solo.waitForText("21");
        assertEquals(true, result);
        m.finish();
    }

    @Override
    protected void tearDown() throws Exception {
        band.saveMarkers();
        band.stopPowerMonitoring();
        band.getBaseStatus(5, 6); //interleave and count
        solo.finishOpenedActivities();
        super.tearDown();
    }
}
```

I.4 NQueen Test

```java
package com.mango.queens;

import android.test.ActivityInstrumentationTestCase2;
import android.widget.EditText;
import com.beftigre.band.Band;
import com.beftigre.band.Marker;
import com.beftigre.band.annotations.Given;
import com.robotium.solo.Solo;

public class NQueenTest extends ActivityInstrumentationTestCase2 {
    private Solo solo;
    private Band band;
    private Marker m = new Marker("NQueen");

    public NQueenTest() {
        super(NQueen.class);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        solo = new Solo(getInstrumentation(), getActivity());
        band = new Band(getActivity(), this);
        band.startPowerMonitoring();
        band.registerMarkers(m);
    }

    @Given(mobileCPU = 0, mobileMemory = 0)
    public void testMathdroid() throws Exception {
        m.start();
        solo.enterText((EditText) solo.getView(R.id.nqEdit), "14");
        solo.clickOnView(solo.getView(R.id.nqBtn));
        boolean result = solo.waitForText("365596");
        assertEquals(true, result);
        m.finish();
    }

    @Override
    protected void tearDown() throws Exception {
        band.saveMarkers();
        band.stopPowerMonitoring();
        band.getBaseStatus(5, 6); //interleave and count
        solo.finishOpenedActivities();
        super.tearDown();
    }
}
```

I.5 Screenshots of Test

Linpack: before test



Linpack

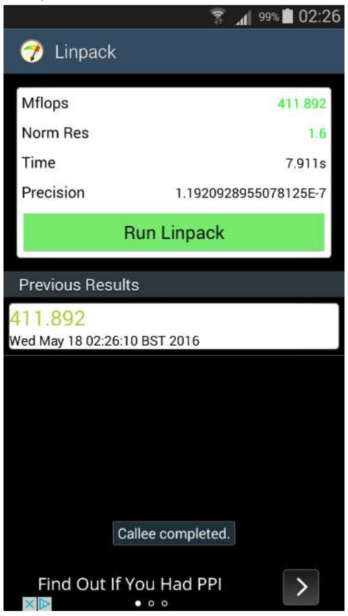
Mflops 0

Norm Res 0

Time 0

Precision 0

Linpack: after test



Linpack

Mflops 411.892

Norm Res 1.6

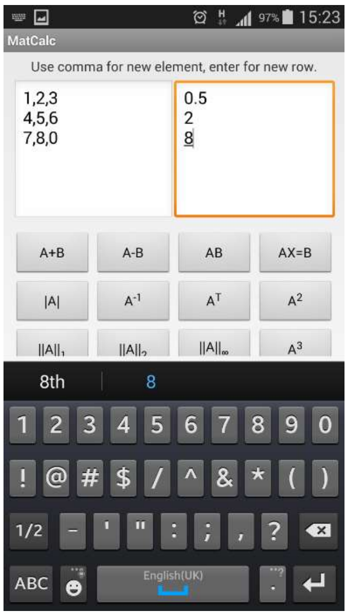
Time 7.911s

Precision 1.1920928955078125E-7

411.892
Wed May 18 02:26:10 BST 2016

Callee completed.

MatCalc: before test

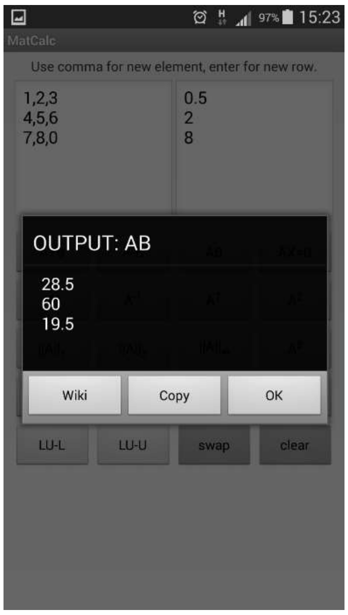


MatCalc

Use comma for new element, enter for new row.

```
1,2,3        0.5
4,5,6        2
7,8,0        8
```

MatCalc: after test



MatCalc

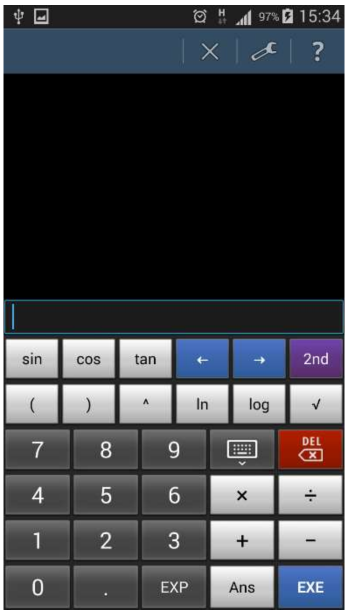
Use comma for new element, enter for new row.

```
1,2,3        0.5
4,5,6        2
7,8,0        8
```

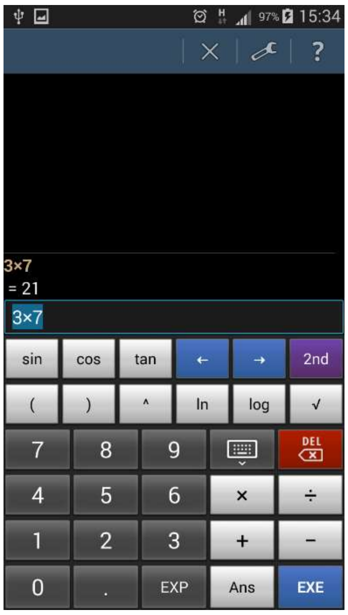
OUTPUT: AB

```
28.5
60
19.5
```

MathDroid: before test

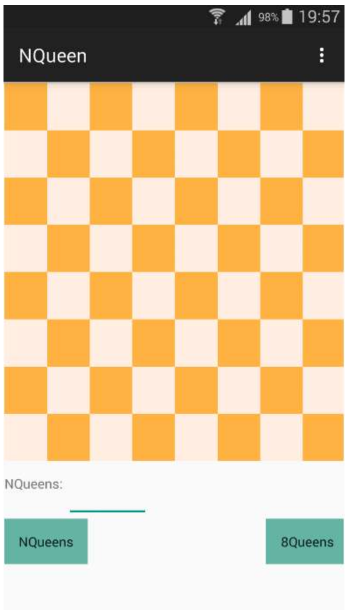


Mathdroid: after test

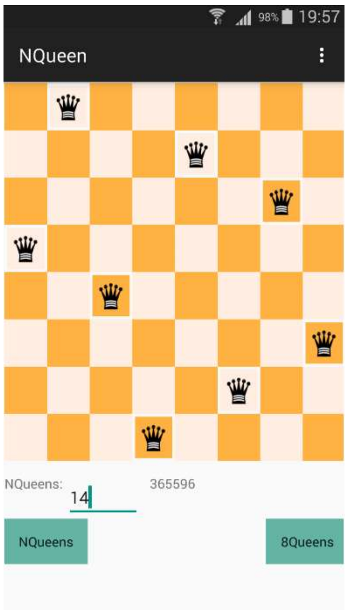


```
3×7
= 21
3×7
```

NQueen: before test



NQueen: after test



Appendix J Case Studies ACTS Snippets

Note: Appendix J only shows the mosaic generated (and manually refactored) sections of the source code for the ACTS classes of the case studies. The underlined code segments in the figures are manually added while the rest of the class body are generated by Mosaic. The highlighted code segments show the placeholders transformed by Mosaic for the specific case study app. Only part of the template has been shown here, as emphasis are on the transformed placeholders and refactored sections. For complete generic template structure of ACTS components see Appendix D.

J.1 Linpack Aspect

```java
package mango;

import ...

@org.aspectj.lang.annotation.Aspect
public class Aspect {

    @Pointcut("call(* rs.pedjaapps.Linpack.MainActivity.runLinpack(..)) && args(arg_0)")
    public static void offloadMethod(Class arg_0) {
    }

    @Around("offloadMethod(arg_0) && !within(Aspect) && !within(Task)")
    public Result aroundOffloadMethodCall(ProceedingJoinPoint jp, Class arg_0) throws Throwable {
        return new Task().execute(new Object[]{arg_0}).get();
    }
}
```

J.2 Linpack Context

```java
package mango;

import ...
import rs.pedjaapps.Linpack.MainActivity;

public class Context {
    private static Activity activity = MainActivity.activity;
    ...
}
```

J.3 Linpack Task

```
package mango;

import ...
import rs.pedjaapps.Linpack.MainActivity;
import rs.pedjaapps.Linpack.Result;

public class Task extends AsyncTask<Object, Integer, Result> {
    private static Result result      = null;
    ...
    private static final int TIMEOUT  = 5000;
    private static final int OVERHEAD = 2542; //25% OVERHEAD

    @Override
    protected Result doInBackground(final Object[] params) {
        ...
    }

    private void runOnMobile(Object[] params) {
        result = MainActivity.runLinpack((Class)params[0]);
    }

    private void runOnCloud(Object[] params) {
        try {
            Socket socket = new Socket("46.137.91.122", 3);
            ...
            result = (Result) inputStream.readObject(); //read
            ...
        } ...
    }
    ...
}
```

J.4 Linpack Service

```
package mango;

import ...
import org.hyperic.sigar.*;
import java.io.*;
import java.net.*;

public class Service {
    private static Result result = null;
    private static final int CPU_THRESHOLD  = 32;   //%
    private static final int TIME_THRESHOLD = 5000; //ms
    ...

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(3);
            ...
        } catch (Exception ex) {  }
    }

    public static void dispatcher(Object[] params) {
        ...
        result = MainActivity.runLinpack((Class)params[0]);
        ...
    }
    ...
}
```

J.5 MatCalc Aspect

```
package mango;

import ...

@org.aspectj.lang.annotation.Aspect
public class Aspect {

    @Pointcut("call(* com.android.matcalc.MainActivity.customTimes(..)) && args(arg_0, arg_1)")
    public static void offloadMethod(Matrix arg_0, Matrix arg_1) {
    }

    @Around("offloadMethod(arg_0, arg_1) && !within(Aspect) && !within(Task)")
    public Matrix aroundOffloadMethodCall(ProceedingJoinPoint jp, Matrix arg_0, Matrix arg_1) throws Throwable {
        return new Task().execute(new Object[]{arg_0, arg_1}).get();
    }
}
```

J.6 MatCalc Context

```
package mango;

import ...
import com.android.matcalc.MainActivity;

public class Context {
    private static Activity activity = MainActivity.activity;
    ...
}
```

J.7 MatCalc Task

```
package mango;

import ...
import com.android.matcalc.MainActivity;
import Jama.Matrix;

public class Task extends AsyncTask<Object, Integer, Matrix> {
    private static Matrix result      = null;
    ...
    private static final int TIMEOUT  = 5500;
    private static final int OVERHEAD = 828;  //25% OVERHEAD

    @Override
    protected Matrix doInBackground(final Object[] params) {
        ...
    }

    private void runOnMobile(Object[] params) {
        result = MainActivity.customTimes((Matrix)params[0], (Matrix)params[1]);
    }

    private void runOnCloud(Matrix[] params) {
        try {
            Socket socket = new Socket("46.137.91.122", 3);
            ...
            result = (Matrix) inputStream.readObject(); //read
            ...
        } ...
    }
    ...
}
```

J.8 MatCalc Service

```java
package com.android.matcalc;

import ...
import org.hyperic.sigar.*;
import java.io.*;
import java.net.*;

public class Service {
    private static Matrix result = null;
    private static final int CPU_THRESHOLD  = 32;   //%
    private static final int TIME_THRESHOLD = 5500; //ms
    ...

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(3);
            ...
        } catch (Exception ex) {  }
    }

    public static void dispatcher(Object[] params) {
        ...
        result = MainActivity.customTimes((Matrix)params[0], (Matrix)params[1]);
        ...
    }
    ...
}
```

J.9 MathDroid Aspect

```java
package mango;

import ...

@org.aspectj.lang.annotation.Aspect
public class Aspect {

    @Pointcut("call(* org.jessies.mathdroid.Mathdroid.computeAnswer(..)) && args(arg_0)")
    public static void offloadMethod(String arg_0) {
    }

    @Around("offloadMethod(arg_0) && !within(Aspect) && !within(Task)")
    public Node aroundOffloadMethodCall(ProceedingJoinPoint jp, String arg_0) throws Throwable {
        return new Task().execute(new Object[]{arg_0}).get();
    }
}
```

J.10 MathDroid Context

```java
package mango;

import ...
import org.jessies.mathdroid.Mathdroid;

public class Context {
    private static Activity activity = Mathdroid.activity;
    ...
}
```

J.11 MathDroid Task

```java
package mango;

import ...
import org.jessies.calc.Node;
import org.jessies.mathdroid.Mathdroid;

public class Task extends AsyncTask<Object, Integer, Node> {
    private static Node result        = null;
    ...
    private static final int TIMEOUT  = 8000;
    private static final int OVERHEAD = 1462; //25% OVERHEAD

    @Override
    protected Node doInBackground(final Object[] params) {
        ...
    }

    private void runOnMobile(Object[] params) {
        result = Mathdroid.computeAnswer((String)params[0]);
    }

    private void runOnCloud(Object[] params) {
        try {
            Socket socket = new Socket("46.137.91.122", 3);
            ...
            result = (Node) inputStream.readObject(); //read
            ...
        } ...
    }
    ...
}
```

J.12 MathDroid Service

```java
package mango;

import ...
import org.hyperic.sigar.*;
import java.io.*;
import java.net.*;

public class Service {
    private static Node result = null;
    private static final int CPU_THRESHOLD  = 32;   //%
    private static final int TIME_THRESHOLD = 8000; //ms
    ...

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(3);
            ...
        } catch (Exception ex) {  }
    }

    public static void dispatcher(Object[] params) {
        ...
        result = Mathdroid.computeAnswer((String)params[0]);
        ...
    }
    ...
}
```

J.13 NQueen Aspect

```java
package mango;

import ...

@org.aspectj.lang.annotation.Aspect
public class Aspect {

    @Pointcut("call(* com.mango.queens.NQueen.nQueenCount(..)) && args(arg_0)")
    public static void offloadMethod(int arg_0) {
    }

    @Around("offloadMethod(arg_0) && !within(Aspect) && !within(Task)")
    public Integer aroundOffloadMethodCall(ProceedingJoinPoint jp, int arg_0) throws Throwable {
        return new Task().execute(new Object[]{arg_0}).get();
    }
}
```

J.14 NQueen Context

```java
package mango;

import ...
import com.mango.queens.NQueen;

public class Context {
    private static Activity activity = NQueen.activity;
    ...
}
```

J.15 NQueen Task

```java
package mango;

import ...
import com.mango.queens.NQueen;

public class Task extends AsyncTask<Object, Integer, Integer> {
    private static Integer result      = null;
    ...
    private static final int TIMEOUT  = 6500;
    private static final int OVERHEAD = 1578; //25% OVERHEAD

    @Override
    protected Integer doInBackground(final Object[] params) {
        ...
    }

    private void runOnMobile(Object[] params) {
        result = NQueen.nQueenCount((int)params[0]);
    }

    private void runOnCloud(Object[] params) {
        try {
            Socket socket = new Socket("46.137.91.122", 3);
            ...
            result = (Integer) inputStream.readObject(); //read
            ...
        } ...
    }
    ...
}
```

J.16 NQueen Service

```java
package mango;

import ...
import org.hyperic.sigar.*;
import java.io.*;
import java.net.*;

public class Service {
    private static Integer result = null;
    private static final int CPU_THRESHOLD  = 32;   //%
    private static final int TIME_THRESHOLD = 5000; //ms
    ...

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(3);
            ...
        } catch (Exception ex) {  }
    }

    public static void dispatcher(Object[] params) {
        ...
        result = NQueen.nQueenCount((int)params[0]);
        ...
    }
    ...
}
```