ULOOF: USER-LEVEL ONLINE OFFLOADING FRAMEWORK

JOSÉ LEAL DOMINGUES NETO

ULOOF: USER-LEVEL ONLINE OFFLOADING FRAMEWORK

Dissertação apresentada ao Programa de Pós--Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: JOSÉ MARCOS S. NOGUEIRA
COORIENTADOR: DANIEL F. MACEDO

Belo Horizonte

Março de 2016

JOSÉ LEAL DOMINGUES NETO

ULOOF: USER-LEVEL ONLINE OFFLOADING FRAMEWORK

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: JOSÉ MARCOS S. NOGUEIRA
CO-ADVISOR: DANIEL F. MACEDO

Belo Horizonte
March 2016

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Domingues Neto, José Leal.

D671u Uloof: user-level online offloading framework. / José Leal Domingues Neto. Belo Horizonte, 2016.
xxiii, 96 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: José Marcos Silva Nogueira.
Coorientador: Daniel Fernandes Macedo.

1. Computação – Teses. 2. Redes de computadores – Teses. 3. Computação em nuvem – Teses. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*22(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Uloof: user-level online offloading framework

JOSÉ LEAL DOMINGUES NETO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

PROF. JOSÉ MARCOS SILVA NOGUEIRA - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. DANIEL FERNANDES MACEDO - Coorientador
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. ÍTALO FERNANDO SCOTÁ CUNHA
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. LUIS HENRIQUE MACIEL KOSMALSKI COSTA
Departamento de Eletrônica e Computação - UFRJ

Belo Horizonte, 31 de março de 2016.

Acknowledgments

This work could not be completed without the support and love from many people across the world.

I would like to firstly thank my mother and father. Mother, you are a true inspiration to me. Without your guidance, love and care I would not be able to glance back and cherish this joyful moment. Father, your acts of encouragement were crucial. Your success and kindness serve as lifetime examples to me. My sisters can also claim a big piece of today's accomplishments, I am very grateful for having you.

I thank my advisor, José Marcos, for granting me great opportunities and for guiding me along the period of my Master.

I express my full gratitude to my co-advisor, Daniel, for helping me throughout tough moments in the writing of this dissertation. I have great respect for your person and hope our ways can cross again in the future.

All this would not be possible without the help from the LIP6 professors, Stefano and Rami. Your inputs, ideas, constructive criticism and long meetings gave birth to full sections of this dissertation.

I want to specially thank my beautiful and loving girlfriend Anna, for bearing with all long lost weekends and stressful periods. You played an irreplaceable role these months, supporting my decisions, helping me to implement crazy ideas and making me happy.

I dedicate this work also to my best friends from Brazil, the lekitas: Alberto and Rodrigo. And to my best friends from abroad, the M&Ms: Miki and Mohsin.

Finally, I thank the international cooperation project "WINDS: Systems for Mobile Cloud Computing" for giving me the opportunity of completing my internship at LIP6 in France, which was sponsored by Fapemig (Brazil) and CNRS (France). I also thank the UFMG university and the DCC.

“Wo gehobelt wird, da fallen Späne”
(German proverb)

Abstract

Our fast paced day-to-day life has been facilitated by the use of mobile phones. Our growing demand on applications that are used for business, entertainment and research areas has been pushing forward the development of the mobile platform. Problems arise when such devices intrinsically furnished with limited resources (e.g. battery, memory and CPU) can not offer the expected quality of service and can barely go through a normal working day without a recharge. Cloud-assisted Mobile Offloading consists of techniques to save battery and opportunistically improve execution time of applications. In this scenario, code is optimally executed on the cloud, depending on some heuristics, culminating in lower mobile resource usage. Offloading frameworks provide an easy start-up and a powerful set of tools to enable Offloading in mobile applications. Identifying common shortcomings of previous works, this dissertation develops ULOOF, an User Level Online Offloading Framework. Our work is a general-use transparent framework that was designed to work in a plug-and-play fashion. We employ an accurate online energy model derived from real device data; and a time model that employs a black-box approach for execution time estimation. Our contribution also includes prediction of latency according to user location, resulting in enhanced contextualized Offloading decisions while incurring low overhead. We created two experimental applications, and their evaluation indicates that ULOOF can reduce the application's energy usage up to 80% and lower execution time up to 47.64% under our setup.

Palavras-chave: Mobile Cloud Computing, Computer Networks, Offloading.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

Portable cellphones are evidently earning growing market share by entering our routines. They have become a useful tool in business and personal life, often used for shopping, payments, reservations, communication, management and many more other common scenarios. The cellphone has become arguably a crucial device ubiquitously present.

With this exponential growing demand on top of this tiny hand held device, one may accept the natural fact that a hardware limitation has to be someday reached. In a quotidian usage of the cellphone, battery charge shortage is a common event, lasting barely a full day under normal usage patterns. Many mobile applications perform operations that are very costly to the limited battery lifetime available in a full charge. As more and more applications are conceived and ported to the mobile platform, this problem tends to be only aggravated. Developers should avoid wasting battery and improve their application's energy consumption pattern. Those that fail to do so are breaching the user contract, providing poor user experience [Ickin et al., 2012] and have higher probability of being uninstalled.

Mobile devices follow the pattern of the well known Moore's law, which states that the processing power will double every two years. One may also see graphics power, storage capacity and high speed connectivity growing at this fast pace. The battery on the other hand does not follow this pattern [Schlachter, 2013] and will become a clear bottleneck [Cuervo et al., 2010]. Other means in software have to be developed in order to extend battery life and improve user experience.

In the Operating System level, Power Saving profiles can be activated in Android [Android OS, 2016] and as well in Apple's iOS [Apple iOS, 2016], slowing down CPU clock and employing several measures to spend less energy. In the application level, a series of techniques using Cloud Computing can be employed to solve the problem of resource limitation from the mobile platform. Coupled with a remote backend, problems with constraints on memory, CPU and network can be solved. The presence of technologies like WiFi and

4G make the fast connectivity to the internet a reality, giving us the possibility of exploiting such techniques that were not possible before . This resource saving technique that runs in the application level is called Mobile Offloading.

Mobile computational offloading migrates executions from the local processor to a powerful platform e.g. a cloud computing environment [Kristensen, 2010]. This may happen transparently to the user. Offloading may reduce the response time and energy consumption on the mobile device.

Memory offloading aims to provide larger memory space by using the assistance of the cloud. Developers often need to be very aware of the hardware limitation, hindering the development of pervasive software and raising costs in maintenance [Chen et al., 2003]. Available memory can be a serious limitation for applications [MobileWebApps, 2015]. Memory offloading has a very similar idea to Chord [Stoica et al., 2001], which is a simple distributed store that solves this memory limitation.

Network offloading partitions the application in such way that the network traffic is reduced. This implies migrating parts of the application that parse raw data and make multiple requests, with the goal of lowering the number of bytes exchanged by method calls. The result is an application that can fetch content faster and still exchange less bytes.

This work focuses in Computational Offloading. When the term Offloading appears without context, computational offloading is intended.

1.1 Motivation

Computational offloading has been a relevant theme for a long time. Since one of its first appearances in the literature, it was demonstrated that one may save energy and conveniently also time by employing Offloading [Li et al., 2001]. Many frameworks were created to assist Offloading and provide cloud-assisted remote code execution. But the state-of-art still fails to provide a combination of features that helps the developer throughout a realistic and easy startup on Offloading.

Some frameworks from the state-of-art affect the development of the application by imposing a specific way of programming, or requiring special privileges in order to be executed. Moreover, some need prior input containing the energy usage of their code instructions or pre-defined test inputs. These can make the adoption curve of a framework quite steep and may lack sense of reality, since it is impossible to simulate all possible use cases of a real product to generate this data.

Seeing these shortcomings, this work is inspired in features gathered from different frameworks and develops further techniques and algorithms. We aim for a general use user-level framework that is location aware.

1.2 Problem statement

This section tries to characterize shortcomings from the state-of-art and justify why this may be relevant to this work's context.

1. *Lack of reality and integration*
   Some works impose a serious burden on the developer requiring drastic code changes, such as developing the application in a specific way or in modules. Some others require a detailed profile of the application, together with input samples for offline profiling. A modified Operating System or runtime is also sometimes needed by some. This lowers the applicability of the framework and makes the projects less realistic to be used in practice.

   We aim to remedy this by creating a plug-and-play framework, where the only code change necessary is the labeling of the Offloading candidates. The Offloading candidates are the set of instructions that may be executed remotely. It gradually learns with each iteration, not needing any extra input. Furthermore, it runs at user-level, requiring no special permissions or modifications to the Operating System.

2. *High Overhead*
   Some of the current frameworks present high overhead in order to make the Offloading decision. Due to code profiling or by means of energy consumption monitoring, significant overhead is incurred offsetting results very often. By actively monitoring resources, the framework also takes its share in energy consumption.

   This work develops mechanisms that try to avoid this burden by creating a low footprint model.

3. *Imprecise runtime estimation*
   Most works employ a simple estimation based in historical execution times. They do not consider the function input nor the result in their calculations. That means that estimation will be poor when (1) transferring the objects and result, and (2) when estimating executions with different arguments. An algorithm may present a different execution time when receiving different input parameters.

   We create the concept of argument assessment, using a black-box approach to estimate method execution time.

4. *Unable to adapt quickly to changes*
   The mobile cellphones are notorious for their portability. Yet most works do not account for mobility in their models. The bandwidth can suffer quick changes when internet connection is shifted from one interface to the other (e.g. from WiFi to 4G), or even within the same interface by changing the access point (e.g. different SSID for WiFi, different cellphone tower for 4G). Most Offloading frameworks employ the available bandwidth and execution speed as input parameters [Cuervo et al., 2010; Chun et al., 2011; Shi et al., 2014], however they do not take into account the possibility of varying bandwidth among cell towers and WiFi networks. This problem is exacerbated with Femtocells [Lopez-Perez et al., 2009], since users are expected to roam among antennas more frequently.

   We develop a location aware model that stores the perceived bandwidth according to tower and SSID. Furthermore, bandwidth is also predicted by a slightly modified Shannon equation.

1.3 Contributions

After identifying common unsolved problems in the works studied, this dissertation developed **ULOOF:** User Level Online Offloading Framework. The framework works in a plug-and-play fashion in real world scenarios. The only compulsory step is to include the framework library in the project and mark the methods as Offloading candidates. After this point, the framework will learn iteratively, gradually improving its estimations and decisions.

More formally, the contributions of this project to the community are:

1. *Realistic User-Level Decision Engine*
   This work is intended to work in real world scenarios with little intrusiveness. Our framework works in user-level without the need of special runtime requirements or a particular Android distribution. Additionally, it aims to provide a general use decision engine that works online. Some works require profiling or an offline round, executing all possible application use cases, in order to create needed data to their models. We strongly believe that this lowers the chance of realistically using the framework in real life.

2. *Remote platform application and communication protocol*
   This dissertation develops a new Android application to respond to remote commands, execute arbitrary methods and return results. This is done by spawning a HTTP server to respond to requests. Additionally, an optimal communication protocol was created

to receive serialized Java objects as method arguments and ship their subsequent results.

3. *Dynamic low overhead accurate energy and execution time model*
   Many solutions require model input parameters that are expensive to acquire. After identifying this occurred overhead when profiling and monitoring resources, this work conceived a novel solution that avoids active profiling of resources and is at the same time sufficiently accurate. Our approach takes contextualized decisions at runtime and estimates method energy consumption and execution time issuing low overhead.

4. *Location awareness*
   The location of the mobile platform has been ignored so far. We believe that user location is an important metric for a good Offloading decision. As such, our model uses indirect user location through WiFi SSID and cellphone tower identifiers to adjust perceived bandwidth estimation. This results in better energy and execution time predictions, and as consequence in better Offloading decisions.

5. *Efficient Decision Engine for both energy and execution time improvement*
   Some of the solutions studied focused on only one Offloading attribute, being execution time the most commonly seen. Our work presents a deep analysis of the trade-off between energy and execution time. Our model, based in Energy-Response time Weighted Sum (ERWS) [Wu et al., 2015], accepts a configurable constant, α, to weight the desired goal according to the situation.

6. *Bytecode manipulator*
   This work develops a bytecode manipulator using Soot [Raja Vallee-Rai and Co, 1999]. It is a post-compiler that statically analyses Offloading candidates, recognizing static objects and arguments that need to be transferred to allow remote execution. It manipulates bytecode to insert entry-points into the methods, providing instrumentation and control to our decision engine. It outputs a modified Android application with the framework embedded.

1.4 Outline

This dissertation is organized as follows. Chapter 2 introduces the concepts behind Offloading and enumerate the related works, identifying their shortcomings. Chapter 3 explains in detail the first two modules of our proposal, the Post-compiler and the Remote Platform. Chapter 4 details the implementation and ideas of the last module, the Offloading library.

Chapter 5 presents the experiments and evaluation of the framework. It also performs an accuracy measurement of our predictions and an analysis of our scaling constant α. Finally, Chapter 6 presents our conclusions and the future works.

Chapter 2

Concepts and Related work

Cloud-assisted remote execution has been used for almost fifteen years already [Li et al., 2001]. At that early time, graph based Offloading schemes were envisioned and fostered by the existing potential residing on the cloud. With technology evolution and internet infrastructure improvement, research took a step forward by making this technique run transparently to the user, turning code execution hybrid by running it remotely or locally.

2.1 Cloud computing

We extensively use the cloud as a mean to outsource the resource availability of the mobile devices and provide high-performance infrastructure. In this section we would like to briefly explain how the cloud evolved to its current state and why is it a suitable platform for Offloading.

Since the beginning of computers, clients used to share the same mainframe, having each their separate work space. They were already capable of accessing a central resource, having simple dumb terminals to send commands. Mainframes were too expensive, and the company would not have financial means to buy each user a computer. This evolved to the isolated virtualized environment, offering ways to run different Operating Systems at the same underlying physical machine. In parallel, the world of personal computers (PC) started to evolve. With the arrival of the microprocessor, computers started appearing at homes and were not only owned by large corporations or government institutions. The first PC was the size of a refrigerator and was prohibitively expensive.

Computer networks were soon after invented, with a vision that if machines could communicate, a network could be created and services could be provided by sharing a central resource server. Engineers started programming software that could serve multiple requests with a server-client paradigm to solve early limitations with synchronization of information. Information was now within close reach and software could shape processes inside a corporation.

This computer network was expanded in a global network, the internet, offering services and data with no geographical limitations, delivering computation as a public utility. Servers that used to lie in a typical local setup can now be placed in Datacenters: facilities with high availability that can hold multiple servers with high-rate data exchange capacity. Datacenters were the crib of the internet, delivering internet pages and hosting servers available 24 hours per day, 7 days per week.

It was only in the nineties that considerable bandwidth facilitated the development of usable end-user services, evolving to the Web 2.0 from the twenty-first century [O'reilly, 2005]. With fast internet speed growth, demand was pushing enterprise applications to be ported to the internet, which offered high availability, scalability and process simplification. An example is the Service-Oriented architecture (SOA) [Papazoglou, 2003], which emerged around this time. In 2002, Amazon launched a series of web services which provided storage and computation as a service. Later on, Amazon created Elastic Compute Cloud (EC2), which is a mark of the first infrastructure as a service (IaaS) provider. Beyond that, the benefit of the cloud was now clearly visible: a low-cost virtualized pervasive service is now a reality.

Since then, Cloud computing has been a rising trend in modern architectures. Cloud computing is not only a buzzword that one may hear in an informal conversation. It is a real mean of solving quotidian recurrent problems. As technology advances we see internet speed and overall availability growing at fast pace. The one-size-fits-all in-house solutions started not to make sense anymore, when big market players started to provide on-demand virtual servers. Managing and hosting a server inside an enterprise is not interesting, because of implied responsibilities such as a safe cooled room, reliable internet access and a operation engineer to look after it. The location where the servers were hosted suddenly is not relevant anymore, becoming nebulous or in the cloud that the internet represents.

There is a range of problems that can be solved with Cloud Computing and they are intrinsically connected with computing. Examples can be the processing of several files laying in a file server or the post-processing of a database. In our case, we outsource computing power by using the cloud infrastructure.

Businesses tend to choose the cloud model due to its obvious advantages over the classic do-it-yourself model. Several vendors now provide IaaS (Infrastructure as a Service), where the client can pay monthly instalments to spin server instances on-demand. This is very advantageous, since most vendors will only charge the used share of resources during the period.

2.2 Dalvik Virtual Machine

The Android Operating System uses a virtual machine (VM) to run all services and applications on top of the mobile cellphone hardware. [1] It is called the Dalvik Virtual Machine [dalvik, 2015], which is a process virtual machine. As discussed later in Section 3, we implement our work on top of Android, and in consequence on top of the DalvikVM.

The DalvikVM is a register based VM that executes applications in Android. It uses Java bytecode as middle language, converting .class files into its own variant: Dalvik EXecutable, .dex. The Dalvik executable is a compact representation highly tuned for devices limited in memory and processing power.

Common virtual machines, like the Java VM, manage their variables on stacks. This is known to reduce the complexity of instructions due to the architectural nature of the stack. On the other hand, DalvikVM is based on registers, which aims at lowering the volume of instructions and thus the amount of data to be processed and managed on memory.

To create an application the developer may [2] program in the Java language. A normal Android development flow comprises of writing the application in Java, compiling it to Java bytecode, converting it to a Dalvik executable and finally packing all resources together with the executables in an Android PacKage (APK) which is a ZIP-File variant. In this work we have developed a post-compiler that modifies the APK and the underlying DEX files.

2.3 Mobile Offloading

Mobile phones are enduring a tough phase in the history of their existence. According to [Index, 2016], mobile data traffic grew 74% in 2015 and an average mobile phone will generate monthly 4.4GB of traffic by 2020, opposed to the average 929MB of 2015. There are more mobile phones than people in the world, reaching 1.5 mobile phones per capita by 2020, trending to be the most used day-to-day platform when accessing internet services. End-users are constantly demanding more capability and quality to aid them executing complex tasks. Mobile phones are the reference device when it comes to personal tool and ubiquitous service entry-point. Mobile applications develop essential appliances, and conquered their place as crucial personal assistant in many tasks. They are pervasively present in many areas, ranging from business to personal affairs.

All this success comes with a price: cellphone technology must be frequently pushed to the edge to follow user's requirements. Devices follow the Moore's law, doubling their pro-

[1]This project was lately discontinued. New Android versions use the Android Runtime (ART) to run its binaries. ART uses the same bytecode and .dex files.

[2]We use "may" here because many languages can now be compiled to Java bytecode.

cessing power every two years. Other cellphone modules also tail said law, having storage capacity, memory size, graphical power and connectivity speed meeting expected growth. Unfortunately, battery capacity has not been following this, growing at slower pace [Perrucci et al., 2011]. We can see this evidence by simply looking at autonomy of a typical battery charge: The cellphone's battery can barely last a full working day. Currently the non-sustainable solutions to this problem consist of buying additional batteries or carrying a portable battery charger. This approach does not scale well, making software solutions an interesting research area.

Mobile Offloading is a technique that can greatly lower battery consumption and opportunistically improve the execution time of mobile applications. It executes code on the cloud from parts of the application that are eligible for transferring, profiting from a powerful platform to avoid using local resources. Offloading can be done in several ways and its engine implementation may vary vastly, but they all face an important issue: Whether offloading is at the moment the optimal choice [Saarinen et al., 2012]. Cloud-assisted techniques share the common problem of having to communicate with a remote platform. There is an involved overhead incurred by network latency, creating an offset in execution time and energy consumption. It means that only specific cases are worth offloading. One of the big questions related to an Offloading framework is how to efficiently calculate this.

Mobile Offloading can come in three flavours: Computational Offloading, Memory Offloading and Network Offloading.

- Mobile computational offloading migrates executions from the local processor to a powerful platform e.g. a cloud computing environment [Kristensen, 2010]. This may happen transparently to the user. Offloading may reduce the response time and energy consumption on the mobile device.

- Memory offloading aims to provide larger memory space by using the assistance of the cloud. Available memory can be a serious limitation making applications run slower [MobileWebApps, 2015]. Developers need to be very aware of the memory available at the device, increasing software development costs [Chen et al., 2003]. Memory offloading has a very similar idea to Chord [Stoica et al., 2001], which is a simple distributed store.

- Network offloading partitions the application in such way that the network traffic is reduced. This implies migrating parts of the application that parse raw data and make multiple requests, with the goal of lowering the number of bytes exchanged by method calls. The result is an application that can fetch content faster and still exchange less bytes.

Offloading frameworks help developers implementing Mobile Offloading. The idea is to offer a tool that deals with the heavy lifting and details of Offloading. A framework aims at proving a fast start-up and easy integration on applications that wish to enable cloud-assisted execution. They can be transparent or non-transparent. A transparent framework will offload parts of an application without any additional developer and user interference. In fact, in an optimal case they should not be aware of the decision and no difference should be noticed between remote and local executions.

This dissertation develops a transparent computational Offloading framework. Therefore in the remaining of this work, other types of Offloading are not covered and our scope is limited to this single flavour.

2.3.1 Transparent Mobile Computational Offloading

As mentioned before, computational offloading can be used as a great tool to improve battery life and reduce execution time from applications. In the development lifecycle of an application, several factors influence overall battery consumption. Frameworks can ease this process by providing Offloading out-of-the-box. In some cases, the framework does this transparently to the developer and user. This means that none of the actors included will need to worry about which platform is executing the code. The framework will be the main entity taking these optimal decisions and conducting the necessary steps to make the execution happen.

2.3.1.1 Structure of a Computational Offloading Framework

An Offloading Framework may consist of two main parts: local and remote platform. Figure 2.1 shows us the minimal structure necessary for a functioning framework.

The local platform holds the Offloading framework library, which can be an additional embedded library, or simply artificially included code (e.g. by a post-compiler which modifies and includes additional methods or classes). The cloud-assisted execution needs extra methods to support the migration process.

Within many functionalities the library lying inside the device may hold:

- *Communication protocol*: The framework has to communicate with the remote platform in order to post execution requests or check whether the platform is available. There are many communication protocols for execution. RPC (remote procedure call) is a famous paradigm, normally suited for complex execution flow and not made for light communication. Moreover, while packing the application it is required to use their own post-compiler that will generate specialized stubs for the calls. Many projects choose to implement their own communication protocol. We also chose the same path.

- *Instrumentation*: In order to take Offloading decisions, the code has to be analysed and evaluated. Many models use in their equations the execution time or number of CPU ticks among some other attributes from the code. Instrumentation is often used to receive information about the execution.

- *Proxy-methods*: When the original method is called, the framework needs to act transparently, intercepting the call, making the Offloading decision and conducting the execution on the respective platform. As a technique to intercept method calls, proxy-methods are created and injected in the application. They substitute the original method, perform necessary processing and redirect the call to the framework.

- *Data stores*: As seen in previous modules, a big range of information is collected about the method. Some models are based on historical executions, raising the need of an efficient data store.

This list is non-exhaustive as there are other modules related to the managing of Offloading, some being model and implementation specific.

The remote platform may consist of one machine or a cluster that will execute code remotely. This can be an RPC server, a HTTP server, an Android application or anything that would support remote execution. Since we are talking about a transparent framework, the remote counter-part follows normally the client architecture. This is necessary to execute the same code universally. For many frameworks that run on the Android Operating System this is mainly done on a DalvikVM [dalvik, 2015]. One of the challenges related to remote execution is how to replicate the state of the original environment in order to perform the execution. This involves exchanging environment information, and is in some corner cases not straightforward as it may include deep native Operating System variables or cross-referenced variables.

2.3.2 Basic Transparent Computational Offloading Architecture

There are many trade-offs when it comes to the decision of a framework's architecture. These can be tightly coupled with the underlying process and may shape the overall performance. We quickly outline common architectural choices.

2.3.2.1 Decision engine

The decision engine is responsible for taking the important decision of offloading or not. This is notably crucial to the framework, since a bad decision will lead to extra resource expenditure. Decision engines will be extremely dependent on the modeling of the problem, having its implementation tightly coupled with it. A framework may choose not to have a decision engine, simply by offloading when possible.

2.3.2.2 Model

There are several models that can express the problem of Offloading. Some common seen models are graph based, finance based and machine learning based. Most also turn Offloading into an optimization problem. This work models the problem using utility functions in order to represent the trade-off between execution time and energy.

2.3.2.3 Mobile Platform

There are many mobile Operating Systems (OS) available: Android, iOS, Windows, Firefox OS and others. Each project may choose its target OS based on some attribute. The main market players are nowadays Android, iOS and Windows [Smartphone Market Share, 2015]. It was observed by us that many works are based on Android [Kosta et al., 2012; Chun et al., 2011; Shi et al., 2014], probably because it is an open source wide available OS. This makes it an interesting choice to experiment freely. Android applications can be programmed in Java, which is a known programming language. Cheap devices carrying Android can also be bought, providing a non-expensive startup for prototypes.

2.3.2.4 Remote platform

Normally the same code is run in both platforms. A DalvikVM is employed at the remote platform to this end. Otherwise, the server-side code is generated and modified in order to run remotely.

2.3.3 Common Transparent Computational Offloading Frameworks Choices

There are several types of Offloading Frameworks, having each its trade-offs in relation to intrusiveness, accuracy, speed, overhead and goals. In this section we will try to outline common Framework setups and quickly discuss their advantages and disadvantages.

2.3.3.1 Goals

Each framework has its goals when offloading, prioritizing on saving execution time or energy consumption. Some frameworks may aim at lowering execution time of the application, focusing on responsivity and user-friendliness. Other works aim at lowering battery usage from the mobile device.

Some works aim at both. This is the case of this work.

2.3.3.2 Partitioning

To migrate code and execute it remotely, there must first exist an application partitioning. The partition will identify the pieces of code that are candidates for Offloading. The Offloading candidates play an important role in the process. If they are not correctly chosen, Offloading might spend more resources than the original version of the application. There can be two different types of partitioning: Static and Dynamic.

- **Static**: In static partitioning, the Offloading candidates will be predefined and will not change according to context. This is typically done beforehand either by the developer through annotations, or automatically by an offline algorithm. This works very well when the framework has other dynamic methods to overcome this problem, such as a decision engine. Since the subject involves a portable device, situation can change drastically very fast by switching networks or changing the device energy regime currently used. The system must adapt to all the variables involved. Moreover, static partitioning can save time and energy by avoiding frequent overhead occurred when running the partitioning algorithm every Offloading round.

- **Dynamic**: Dynamic partitioning will typically choose the Offloading candidates according to the context. Aided by a profiler, the environment variables are read and a decision is made in runtime, tailored to the situation. As mentioned, dynamic partitioning generates an overhead for each execution, but may work better for projects that do not include a robust decision engine.

2.3.3.3 Code granularity

The remote execution will always happen inside a scope or application partition. This scope may be the whole application or simply a method or class. Most commonly seen, some frameworks allow methods to be offloaded, others may allow coarse grained granularity and offload whole threads.

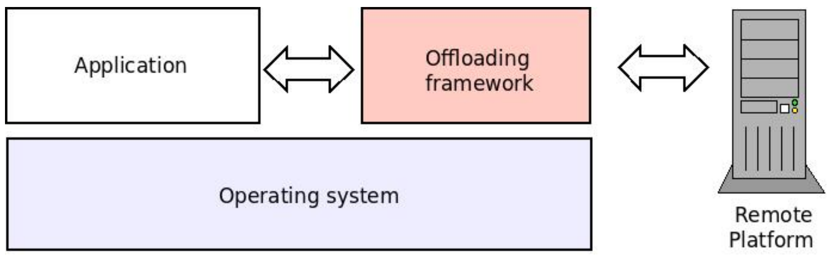

Figure 2.1. Typical offloading framework structure

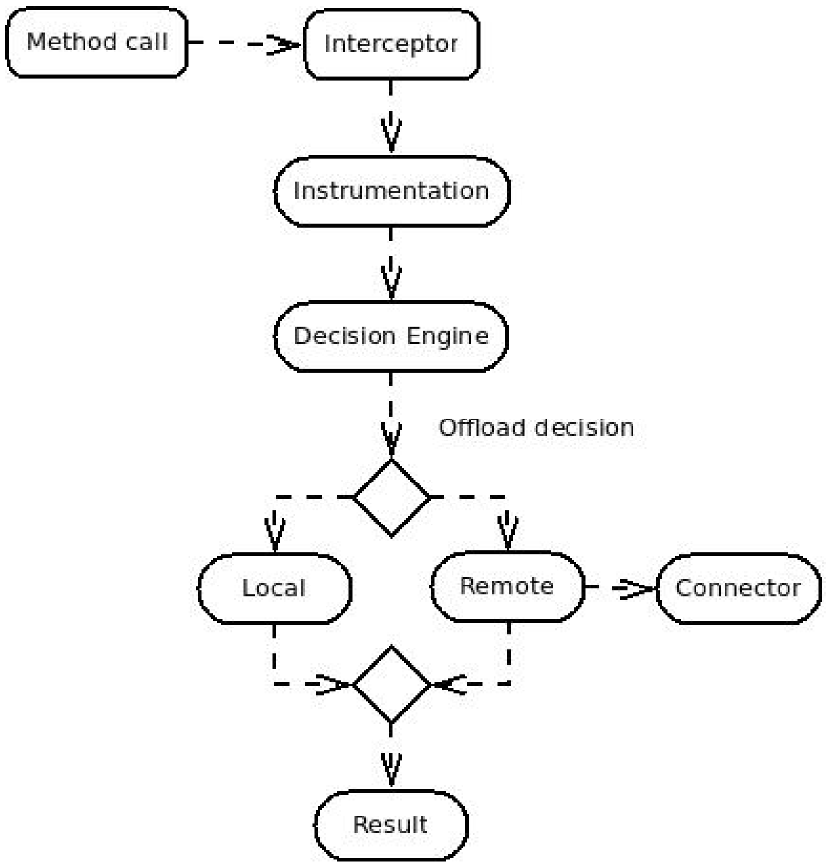

Figure 2.2. Diagram of a typical process life cycle inside the Offloading framework.

2.3.4 Decision engine

Although the decision engine is part of the framework's architecture, we decided to give more insight about the module, due to its complexity. The decision engine plays a crucial role in an Offloading framework. Not all frameworks implement a decision engine, circumventing this problem through dynamic partitioning or other means. A well fitted decision engine will result in good Offloading decisions, thus in less resource usage.

Figure 2.2 shows a small diagram of the typical process life cycle inside the Offloading framework that lies on the mobile platform. As introduced in section 2.3.3, frameworks come in several flavours, this is a simple example for contextualization.

The interceptor works as an entry-point for Offloading, intercepting the method and forwarding the request to the framework. The instrumentation component instruments the candidate methods/functions (unit of code in programming languages) for offloading. Whenever such methods are about to be executed, the instrumentation calls the decision engine in order to determine if Offloading should be performed. The instrumentation component also stores results of method execution times, network usage, partial calculations of bandwidth among others. The decision engine reaches a boolean decision regarding Offloading. In order to do so, it must estimate how much energy and CPU the method would consume if it were executed locally, as well as the time and energy required to transfer the method and its data to an Offloading infrastructure.

The connector module is activated when Offloading is performed, and takes care of remote connectivity and serialization/deserialization of objects.

2.4 Related work

An Offloading framework consists of different domain areas, reaching a wide spectrum of technology. We chose to separate the related work in small sections, each handling a distinct area or introducing a challenge faced in this project.

2.4.1 Framework architecture

The MAUI [Cuervo et al., 2010] framework offers a dynamic profiling mechanism called Maui Solver. Its main goal is to minimize energy consumption. MAUI collects runtime data for later assessment, taking RTT and bandwidth into account. These values are gathered and used as average estimates. It uses historical data to predict future execution time and energy consumption. A compiler generates wrappers for Offloading candidates that transfer application state and communicate with the remote server. MAUI is built in C# and runs in the Windows Mobile Operating System. To model energy consumption, a power meter was attached to a mobile phone, and a relation between CPU and network usage was created. Later on, by monitoring the methods marked as Offloading candidate, energy consumption and execution time are predicted and evaluated. An optimization problem is solved in order to make the Offloading decision. The work does not reference how precise are their predictions if the arguments vary. This can be a limiting point, since the method's execution time is very dependent on input. Our work runs in Android, uses static partitioning as well and derives a similar energy model through an external meter.

CloneCloud [Chun et al., 2011] is a graph based dynamic partitioning Offloading framework that runs in Android. It first profiles the application to create a cost graph for each method, feeding a database with partitions. To generate partitions, it simulates the cost of transferring state between mobile and remote platforms. For energy consumption, it experimentally monitors separate modules and creates a simple cost model. In execution time, it uses an optimization solver to select the best partition (the one with the smallest cost) based in the current scenario.

The framework ThinkAir [Kosta et al., 2012] has an energy model based in PowerTutor [Zhang et al., 2010] to actively monitor hardware components and estimate energy consumption. It combines features from CloneCloud and MAUI, statically choosing the Offloading candidates through method annotations and managing multiple virtual machines for the remote platform. Upon request, ThinkAir can spawn multiple VMs to respond to eventual high volume of requests, splitting computation across nodes if necessary. Multiple profilers run in order to monitor the application and decide in runtime the platform responsible for executing the method.

Cuckoo [Kemp et al., 2012] is an Offloading framework that uses IPC (inter process communication) as its main mechanism. The programmer develops the application with an interface definition language called AIDL (Android Interface Definition Language) and a pre-compiler generates Java code to enable Offloading. It has no built in decision engine, always offloading when possible.

COSMOS [Shi et al., 2014] models the Offloading decision as an optimization problem and works in a greedy fashion, looking into historical data and deciding whether to offload or not. COSMOS focuses on execution time improvement, while this work focuses in both energy and execution time. In the server side, COSMOS also aims to manage the remote platform resources, diminishing costs by effectively allocating and scheduling Offloading requests to reduce the monetary cost per request.

AIOLOS [Verbelen et al., 2012] is a framework based in Open Service Gateway Initiative (OSGi) [OSGi, 2016]. The application is developed in OSGi bundles that can be easily ported to other platforms due to cross-platform implementations of Java. Each bundle has several interfaces exposed that are monitored and profiled at runtime. A bundle monitor gathers data and provides the logic for estimating the parameters of the network and the capabilities of the cloud.

The framework ANYWARE [Pamboris, 2014] uses static code analysis to partition code based in a consumption graph. It uses DTrace to profile the application in an offline round by giving a set of predefined inputs. The result of this is a consumption graph with the associated CPU and memory cost for each function. It splits the application into two parts, running one of the halves on the cloud.

Finally, MARS [Cidon et al., 2011] is an RPC-based (remote procedure call) online framework. It uses a priority queue to choose the best Offloading candidate founded by two metrics: POR and EOR. POR represents the speedup gained from Offloading and EOR represents the energy that would be saved in case of offloading. Every time a new RPC is scheduled these metrics are updated according to the current system status. MARS aims at keeping both network and CPU fully utilized as long as using a resource will significantly improve the energy efficiency of the mobile device. To assess results a simulation was done, opposed to our work that made instead an experiment in a real world scenario.

2.4.2 Remote platform

As discussed later in this dissertation, the remote platform is a crucial piece in the Offloading framework. The whole duration of a remote execution request is directly proportional to remote execution method runtime. Meaning that a slow remote platform renders impossible optimal Offloading [3]. Cloud-assisted executions should provide powerful instances to allow the fixed communication overhead to be amortized.

Although Offloading frameworks or Cyber foraging [Satyanarayanan, 2001] schemes rely heavily in a remote platform, details are often not provided. Therefore fully implementing the remote platform of a framework is not possible.

Cloud virtualization is an important matter to this project. There are several virtualization platforms available like QEMU [Bellard, 2005], Virtualbox, Xen [Barham et al., 2003], VMWare [VMWare, 2016]. The Android's official emulator [Android AVD, 2015] is very handy for tests, but it is unfortunately very slow and unsuited for fast remote execution. Genymotion [Genymotion, 2015] provides commercial fast Android emulators for headless testing.

A main widely used project is Android x86 [Cuervo et al., 2010; Kosta et al., 2012; Chun et al., 2011]. Android x86 is the DalvikVM virtual machine port to the x86 platform. It can be run in virtual containers and is very well suited for remote execution, since it can be installed on cloud providers. This work uses Android x86 as its backend in the remote platform.

[3]This was evaluated in Section 5.3

We use the term remote platform, because it can be comprised of a cluster of servers. The work in Ha et al. [2013] aims fast provisioning of VM images to respond to the elastic demand of computational offloading frameworks. Our implementation is minimal and functions with a single server.

2.4.3 Decision model

Most solutions like MAUI, ThinkAir and CloneCloud are graph model based. They are modelled with each node being a method or function and each edge representing a method call. They are subsequently weighted in the form of a cost function that can be derived of various attributes. The Offloading decision becomes a graph partition problem, where the main goal is to minimize the cost of partitions. AIOLOS uses a history based profile to estimate network and execution parameters, influencing their Offloading decision.

The work in Esteves et al. [2011] exploits financial models, modeling the Offloading decision as a cost-effectiveness problem. This work performs similar data fitting to obtain the utility functions and to choose the local optimal value in a greedy fashion.

We base our decision model in utilities. We estimate execution cost in the form of time and energy utility functions, representing therefore the execution time and energy consumption of a method respectively. To join both costs we introduce a scaling constant α. To arrive at a decision, we compare the remote and local joint costs to choose the smallest one.

2.4.4 Energy model

Offloading frameworks normally use energy models to estimate energy consumption of a method or function. This is done by predicting cost of a set of instructions and later comparing values in both remote and local platforms to reach a decision. The problem of modeling energy consumption still remains an open problem. Mobile cellphones do not provide interfaces for precise measurement of energy consumption, meaning that researchers have to be creative to circumvent this problem. In the user level, this issue is aggravated, since no system modifications can be made to allow access to internal parts of the system.

Several works created device specific energy models by measuring energy usage [Cignetti et al., 2000] [Shye et al., 2009]. PowerTutor [Zhang et al., 2010] is a general framework, running at user-level for energy prediction. By analysing each hardware module's resource usage (e.g. ticks for CPU, bytes for Radio, seconds of screen on for the LCD display) it is able to have an approximation of the power consumption. In this project, we initially considered using PowerTutor. After some tests, we attested high overhead incurred by the active monitoring of these hardware modules. Additionally, the project is not completely versed with modern radio technology (no 4G module). Device specific constants also have to be configured and due to the lack of further information, we feared for the precision of the measurements and decided to take other paths to measure consumption.

MAUI [Cuervo et al., 2010] uses an external tool to measure energy consumption and arrives at a satisfying energy model. Our approach is very similar, as we sample real data to estimate instant energy consumption per module. ThinkAir [Kosta et al., 2012] uses a model inspired in PowerTutor [Zhang et al., 2010], measuring each mobile module at runtime.

2.4.5 Execution time model

For estimating execution time, AIOLOS [Verbelen et al., 2012] uses the size of the method's arguments and result of historical executions to linearly interpolate between these values. For predicting future inputs Balan et al. [2003]; Cuervo et al. [2010]; Kosta et al. [2012] employ a linear model of resource usage to extrapolate future values. This allows an approximate prediction of Offloading inputs when they are not readily available or stale, such as bandwidth and method execution time.

This work improves prediction by interpolating input values using a spline. Our black-box approach is opposite to Mantis [Kwon et al., 2013], which consists of an additional offline round, feeding the system with predefined sample-inputs and method features.

2.4.6 State replication

Cloud-assisted execution relies partly in replicating the state of the mobile on the remote platform. That may imply in serializing and transferring objects through the network. The work in Kwon and Tilevich [2012] applies a forward dataflow analysis to discover objects used by the method. It combines this with a side-effect free analysis, which determines whether the method changes the environment within execution, in order to decide if an object should be transferred. It inserts checkpoints along the method instructions to ensure fault-tolerance, synchronizing both parts employing copy-restore semantics [Tilevich and Smaragdakis, 2008]. As mentioned before, CloneCloud also synchronizes states across partitions.

To synchronize state, heap objects need to be exchanged. In Yang et al. [2013], the author successfully lowered the transfer size of these objects by employing a technique to reduce the number of accessible heap objects. The new filtered list, called Essential Heap Objects (EHO), is obtained by a static analysis and contains only the objects that are referenced in code. Java objects inherit several attributes from its superclasses, accumulating unused heap size. This technique exploits this fact, leaving all these unreferenced attributes out. We do not go far in optimizing the transfer size of our objects. We opted on starting simple with a minimal functioning algorithm to transfer objects related to the method being offloaded.

2.4.7 Remote execution

An Offloading framework contains internally a remote execution mechanism. This is strictly necessary if one desires to execute code remotely. There are existing mechanisms that can be used such as Java RMI [Java RMI, 2016], OSGi [OSGi, 2016] or IPC [Android AIDL, 2016]. AIOLOS [Verbelen et al., 2012] uses OSGi for its solution, Cuckoo uses AIDL (Android Interface Definition Language) to enable client-server execution. Java RMI is not directly supported by Android, therefore cannot be used in this project.

We chose to develop our own mechanism in order to give us more flexibility in the process. Besides that, typical RMI (Remote Method Invocation) solutions do not solve the problem of having to replicate the mobile platform, providing only the implementation for the method call.

2.4.8 ULOOF Tabular comparison

Table 2.1. Comparison between the state-of-art Frameworks and our proposal

Name	Intrusiveness	Decision Engine	OS/Language	Energy Model	User-Level	Plug-and-play
MAUI, Cuervo et al. [2010]	Low	Independent of arg.	Win/C#	Online	✓	✓
CloneCloud, Chun et al. [2011]	Runtime modification	Offline instrumentation	Android/Java	Offline	✗	✗
ThinkAir, Kosta et al. [2012]	Runtime modification	Independent of arg.	Android/Java	Online	✗	✓
Cuckoo, Kemp et al. [2012]	Development in AIDL	✗	Android/Java	✗	✓	✓
COSMOS, Shi et al. [2014]	Code modification	✓	Android/Java	✗	✓	✗
AIOLOS, Verbelen et al. [2012]	Development in OSGi	✓	Android/Java	✗	✓	✗
ANYWARE, Pamboris [2014]	Low	✗	-/C	✗	✓	✗
ULOOF	Low	✓	Android/Java	Online	✓	✓

Table 2.1 compares the major existing works with ULOOF. When the attribute is present, it is marked with "✓", otherwise is marked with "✗". In some cases, some particularities are observed in the fields. Intrusiveness depicts the amount of change necessary in the application for the Offloading framework to work. That can be either Operating System runtime modifications, necessary code modifications and development paradigm such as programming your methods into modules. Some works do not have a decision engine (marked with ✗), choosing always to offload when possible. In this column, two works are marked with "Independent of arg.". It means that their decision engines work independently of argument input, which may incur in less precision in comparison to ULOOF when predicting results of a method. Finally, one important attribute to be compared is the presence of an energy model inside the decision engine. Online energy models will work with real time measurements directly on the device, while offline versions will generally require a previous knowledge base from the application. That can either be profiling information or even predefined input set for analysis. This is a turning point in the application development life cycle, as it is sometimes not possible to run the application with all possible use cases. The column "Plug-and-play" covers this aspect, marking if the framework would work by simply integrating it to the application or if any other information is needed.

2.5 Conclusion

Mobile device usage is shaping market evolution in terms of applications and services provided, becoming the reference device for accessing services. As a consequence, mobile battery consumption is an issue that hardware manufacturers are having a hard time solving. Computational Offloading frameworks were created to aid the developer implementing cloud-assisted functionalities to help on mitigating resource limitation on the mobile platform.

Using the information gathered within these many domain areas this project spans, we have arrived at an Offloading framework that combines several features and techniques from others, filling a gap that no precursor occupies. Our work therefore runs at user-level, works transparently to the user and developer, reaches an Offloading decision online, successfully models energy consumption and execution time, is location aware adjusting its bandwidth quickly depending on user location, integrates itself easily with any application and requires no prior knowledge of the Offloading candidates.

The next chapters will outline our proposal in depth.

Chapter 3

Proposal and implementation

Based on shortcomings and strengths of other works, we now propose **ULOOF**, an Offloading framework that combines techniques to improve common limitations of previous works. Our work's main goal is to provide a general use user-level framework. The framework should work transparently and avoid being intrusive. Our proposal works in a plug-and-play fashion, mitigating the problem of having to instrument and provide energy readings from the application beforehand. Finally, this framework is location aware, relying on indirect user location when taking contextualized decisions.

The Android Operating System was chosen as target platform [Android OS, 2016]. Android is an open-source mobile Operating System that is widely used in the industry. We opted for Android because of the openness and community engagement. The tools available are also of great importance and weighted our decision positively. Many applications and libraries are available in Android's ecosystem, facilitating the framework development.

Android uses Java as its main programming language. Since we aim for a user-level framework, the most natural code granularity is on method-level. We first focus on giving the developer the right tools without imposing any strict policy when developing an application. We allow the indication of Offloading candidates through annotations, being therefore a static partitioned framework. The Offloading candidates are the methods that may be offloaded according to the situation. The developer is also required to import our Java library, which contains methods and interfaces that permit monitoring, instrumenting and communicating with the remote platform.

Our proposal develops a post-compiler to modify the original application and insert interceptors. With the aid of this tool together with our library, the modified application is enabled for offloading.

Finally, this work is additionally composed of a remote platform, which replicates the mobile platform's environment and executes the desired method remotely.

We now describe in detail the proposal and implementation of our Offloading framework. It consists of 3 main parts:

1. *Post compiler*
   The post-compiler stitches the original application with the Offloading framework. It statically analyses the application's Java bytecode and modifies it to intercept the method calls. There is a small logic behind the post-compiler to choose the objects to be serialized and transferred to the remote platform. Additionally, it adds extra calls for instrumenting the Offloading candidate. Section 3.2 explains in detail the proposal and implementation.

2. *Remote platform*
   The remote platform comprises of one or more servers running a customized HTTP server on Android x86 to receive requests and serve remote executions results. Section 3.3 outlines the ideas and implementation details.

3. *Offloading Library*
   The library is a set of Android classes that enable the management, storage, monitoring, instrumenting and decision making of the Offloading framework. In its internals it is divided into 5 components: *OffloadingManager*, *DecisionEngine*, *EnergyModule*, *BandwidthManager* and *ExecutionManager*. In Section 4 we explain each one of these modules and describe their implementation.

This chapter describes the *Post compiler* and the *Remote platform*. The next chapter covers the *Offloading library*. We decided to detail the Offloading library in a separate chapter because of the information volume. In this work, concept and implementation are introduced together. We found this the most natural way to pass the information.

3.1 Process overview

Our proposal uses a simple process to enable Offloading: An application bundle is developed without any restrictions. The only required steps are to import the Offloading framework library and to annotate the desired methods as Offloading candidates. An *APK*, compiled application bundle from Android, should be created just as in the normal Android development lifecycle. This will be forwarded to a post-compiler, which will statically analyse the bundle and look for the annotated Offloading candidates. The post-compiler will output a modified APK with the method interceptors and all the necessary instrumenting code. This newly generated APK can be installed in the device exactly like a normal application and is now enabled for Offloading. It will contact the remote platform and automatically run the required steps to offload according to the decision engine.

3.2 Post-compiler

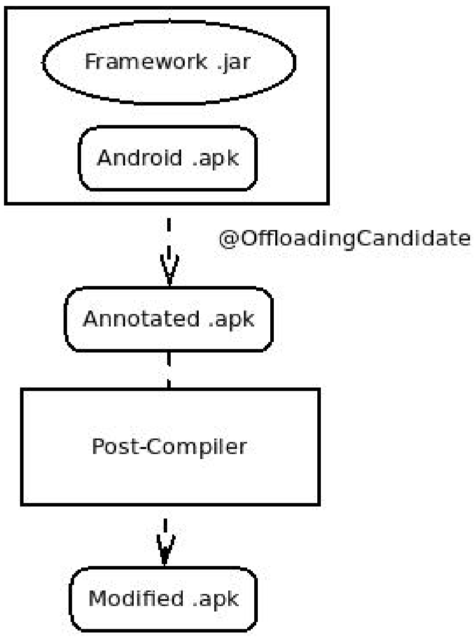

Figure 3.1. Diagram of the process of marking the Offloading candidates and running the post-compiler.

The post-compiler is a separate module of the Offloading framework. In other words, it does not get shipped together with the modified application bundle. Because of its responsibilities, it can be considered as the bootstrap of the Offloading framework. It will bridge the Offloading candidates with the internal parts of the framework by modifying the original method instructions and calling the methods from the Offloading library.

Figure 3.1 shows a simple diagram of the overall process of the module. The Android application is developed without any special attention. The developer should import the Offloading library and annotate the desired methods with the annotation *OffloadingCandidate*. These marked methods are now called Offloading candidates. The application is compiled and an *APK* is generated, following the normal Android development lifecycle. The post-compiler can be now executed, receiving the APK as input. The post-compiler will statically analyse the Java bytecode from this bundle and will look for the annotated methods. It will modify each method, inserting instructions to intercept, instrument and call the decision engine from the Offloading library. The output is a modified APK, ready to be installed in an Android device.

Since the developer has to manually choose and annotate the Offloading candidate, our project may be classificated as static partitioned. Here is a simple example of a method marked as Offloading candidate:

```
@OffloadingCandidate
public void veryCPUIntenseMethod() {
  // Here is the method's code
}
```

3.2.1 Requirements

The Offloading framework is separated into modules, but in its process there are several interfaces that need to be bridged and impose certain requirements. With the big picture in mind, the post-compiler must perform the following tasks:

- Receive a compiled Android application, modify the underlying Java bytecode and output a valid modified APK.
- Modify the Offloading candidates to intercept calls, execute instrumentation code, call subsequently the decision engine, call the method remotely or locally according to the decision and return the results.

To fulfill these requirements, we have employed a library for bytecode manipulation, Soot [Raja Vallee-Rai and Co, 1999]. Using Soot, the method is modified and plain Java methods from the Offloading library are called to perform the tasks of instrumenting and querying the decision engine.

3.2.2 Soot: Bytecode manipulator

Soot is an open-source framework for analysing and transforming Java and Android applications. It was developed by the Sable Research Group of McGill University. It accepts several inputs, such as Java bytecode, Android APKs and plain Java, outputting Java class files and also APKs.

Soot is a very good option for this project, fully serving the purpose of our post-compiler. We have developed a Java application that imported the Soot library and used it directly to manipulate the Offloading candidates. The post-compiler generates a new APK version with Offloading capabilities.

Soot uses an intermediate representation language called Jimple. Jimple [Vallee-Rai and Hendren, 1998] is a 3-address language [1], which simplifies Java representation for optimization and its the heart of Soot. We use Jimple to analyse each of the Offloading candidates and modify their inner instructions.

Although Soot greatly facilitated the manipulation of bytecode, it also presented some surprises. One of the challenges faced, was the boxing and unboxing of primitives. While this is automatically done in Java, we had to program this logic again manually. Besides that, Soot has a steep learning curve, being difficult for a developer to be productive without a necessary assimilation time.

3.2.3 Bootstrapping the Offloading library

Before we continue, we must first introduce the main algorithms behind the static analysis. In order to transfer state between platforms, the Offloading library needs the Offloading candidate class instance (if it is not static), the method input parameters and the possible static objects that the instructions inside the method reference. Since the post-compiler binds the original method with the library, it must comply with this requirement and provide the required arguments. The Offloading library accepts a Java *Map* class, containing the required objects in a pre-defined pattern.

Our post-compiler first copies the original method M, creating a copy M' with the same contents but different name. It then erases all instructions from M and inserts the logic to call the Offloading library.

We introduce three algorithms that are the core of the post-compiler: *EnableOffloading*, *TransformMethod* and *GetRelevantFields*. *EnableOffloading* is called for each Offloading candidate marked. It modifies the method to enable Offloading. It calls *TransformMethod* which inserts the required calls to our Offloading library. Internally, it calls *GetRelevantFields*, which extracts the relevant static fields that method M uses.

Algorithm 1 Enable Offloading in a method

```
1: function EnableOffloading(M)
2:     $M' \leftarrow copy(M)$
3:     $M.instructions.clear()$
4:     $M.instructions \leftarrow TransformMethod(M, M')$
5:     $M'.name \leftarrow "\$copy" + M.name$
6: end function
```

[1]3-address codes are normally used by optimizing compilers as intermediate language to help code transformation.

Algorithm 1 enables Offloading for the given method. Line 2 copies the original method, creating M'. Line 3 cleans the instructions from M. Line 4 calls the algorithm *TransformMethod* to generate new instructions for M and enable Offloading. Finally line 5 changes the name of the newly copied method to avoid clashes. We omitted an additional step from this algorithm for simplification: In case of recursion, we additionally modify M' to point to itself instead of the old M method. This is important to avoid calling the framework repeatedly in case of recursion.

Algorithm 2 Transforms method to bootstrap Offloading library

```
1: function TRANSFORMMETHOD(M, M')
2:     $A \leftarrow$ new Map
3:
4:     if not IsStaticMethod(M) then
5:         $A.put("@this", this)$
6:     end if
7:     $i \leftarrow 0$
8:     for all $a$ in M.arguments do
9:         $A.put("@arg" + i, a)$
10:        $i \leftarrow i + 1$
11:    end for
12:    $F \leftarrow GetRelevantFields(M', \emptyset)$
13:    for all $f$ in F do
14:        $A.put("@field-" + f.name, f)$
15:    end for
16:    $decision \leftarrow ShouldOffload(M.signature, A)$
17:    if decision is TRUE then
18:        $result \leftarrow OffloadMethod(M.signature, A)$
19:        if Exception was thrown then
20:            Go to line 24
21:        end if
22:        return result
23:    end if
24:    $(result, Measurents[]) \leftarrow InvokeLocalMethod(M', A)$
25:    $UpdateInstrumentation(M.signature, Measurements[])$
26:    return result
27: end function
```

The logic behind Algorithm 2 is very simple: Line 2 initializes the Map that will contain the class instance (notably the "this" object), the original function arguments and the used static fields inside M. Line 4 checks whether is necessary to include the "this" object. In Java, if a class is static, it does not have an instance. The for loop in line 8 inserts in the Map all the arguments received by the method. In line 12, a list of relevant static fields used inside the original method are fetched and subsequently inserted in the Map, prepended by "@field-" as name. At a later time, the prepended string is used to identify the entry as a static field. In line 16, we call the decision engine, passing the method's signature and the Map with all the necessary information. The method's signature is an unique identifier, which is used by the library to distinguish calls and store relevant data regarding the candidate. Line 17 checks if the decision is to offload, if yes it calls the function *OffloadMethod* from the library. Line 19 checks whether any exception was thrown when offloading the method. In this case, the execution will continue locally. Line 24 covers the local case, when the decision engine outputs false. It invokes the local method and returns the result along with other important metrics like CPU ticks and bytes exchanged. It then manually calls the *UpdateInstrumentation*, which is defined as Algorithm 9, which will update data about the current local execution. This same step is hidden inside the function *OffloadMethod*, which updates these metrics internally before it returns. Finally the result is returned in both cases.

While this is all done in idiomatic Jimple instructions, Soot does a good job abstracting the platform, allowing easy manipulation and generation of instructions.

We can now introduce the algorithm that fetches the used fields inside the original method M.

Algorithm 3 Recursively fetches the relevant static fields used by method M

```
1: function GETRELEVANTFIELDS(M, Visited)
2:     if $M \subset$ Visited || $IsJavaLibrary(M)$ || $IsAndroidLibrary(M)$ then
3:         return $\emptyset$
4:     end if
5:     $Visited.add(M)$
6:     $R \leftarrow \emptyset$
7:     for all i in M.instructions do
8:         if $IsStaticField(i)$ then
9:             $R.add(i)$
10:        end if
11:        if $IsMethod(i)$ then
12:            $R.addAll(GetRelevantFields(i, Visited))$
13:        end if
14:    end for
15:    return $R$
16: end function
```

Algorithm 3 is a recursive algorithm to detect static fields being used inside method M. In line 2 it presents the stop cases, where it checks whether the method was already analysed in the iteration. It also checks whether the method is a Java or Android system method, where we have no access to the instruction method body. In this case it simply returns an empty list. Line 5 will add the current method to the visited list. In line 7 all instructions inside the method are checked. Line 8 adds the current analysed field if it is static. In order to cover all the method scope, we use recursion on methods that are used within M. In line 12 the recursion happens. Since this function returns a set of fields, we simply delegate the job of checking the inner fields of the next method to the recursion. A list of fields or an empty set can be returned and added to our result list in this step. Finally we return the complete list of static fields used in the scope of M.

3.2.4 Static field list

In order to run the method remotely, one would need to replicate the local environment on the remote platform in order to receive the same result as it would incur on the local device. This task cannot be completed by simply serializing the whole class instance. In Java, static fields used within a method are not part of the class instance. Whether they are being set or get, they are nevertheless needed to replicate the scenario remotely. We claim that by serializing the method's instance, arguments and used static fields, one can replicate the complete state of the local device in relation to the method's scope.

Proof. Inside the method instruction body, there are limited ways of accessing fields: (a) fields belonging to the local scope; (b) Fields belonging to the class instance scope; (c) Static fields used directly by the method; (d) Static fields used indirectly by the method (those that are called inside other called methods).

In (a), we refer to the method's arguments and any newly created local fields. The arguments are being inserted in line 8 from Algorithm 2. The local created fields are going to be replicated in the remote code, as they are originally created in code.

In (b) we refer to fields that are contained inside the class instance. In line 4 from Algorithm 2 we check whether the method is static, and include the *this* object in the map if true. Inside the *this* object is the underlying method instance and consequently also the fields.

In (c) and (d), those are the fields that are not inside the method's instance object, and need to be included in the Map. This is covered by the recursive Algorithm 3 and is later included in the Map that will be transfered to the remote platform.

With this we cover all cases and can successfully transfer the local device's state. ■

In some cases, a static field does not need to be transferred to the remote platform. This may be a common resource already available in the remote part, such as a map, or a file. In the way we developed the post-compiler, if a local field is omitted from the map, the remote platform will simply ignore the setting and continue the execution normally. The developer can use the annotation *IgnoreStaticField* to make the post-compiler skip that field and omit it from the remote platform.

We are aware of further possible optimizations in this area. As mentioned in Section 2, we could have used an approach similar to Yang et al. [2013] to lower transfer size when replicating state. We chose to start with this simple implementation and enhance our solution in the future.

3.2.5 Limitations

This approach has some limitations. As mentioned before in Algorithm 3, one of the stop cases are Java or Android methods. Because this is sometimes compiled machine code, we do not have the instruction body readily available to Soot. As a result, we cannot search for static fields and we will consequently not fully replicate the system's environment if the field lies within this method. One should avoid marking a method as Offloading candidate if it makes use of native libraries. Normally these internal unaccessible states are set by methods that are connected with the local device (e.g. camera, GPS, screen) and are not good Offloading candidates. This limitation is not so severe, considering that if no static internal native variables are changed the state replication will complete successfully.

The current implementation of the post-compiler does not issue any warnings if the method marked as candidate is using local resources or calls native libraries.

3.2.6 Choosing the Offloading candidates

Our post-compiler processes all methods marked with the annotation *OffloadingCandidate*. This delegates the developer the rather big responsibility of choosing the Offloading candidates.

Because of the nature of cloud-assisted execution, many methods are not suitable for Offloading. Methods that will access or manipulate local hardware are a clear limitation. If the method needs to take a picture, move local files, get the user's location or blink the screen, Offloading cannot be at all involved.

Apart from these limitations, there are general recommendations that should be followed when choosing the candidates. Since our technique relies in replicating the state across platforms, annotating methods that receive big size arguments or return big size results will generally render Offloading impracticable. The framework will need to transfer these objects, delaying the remote execution. The framework will do a good job recognizing this situation, but in the discovery time, when the framework first needs to acquire data about the method, it will execute it at least once remotely [2]. Besides that, there is an additional overhead incurred by serialization. It means that big or complex arguments and results should be avoided.

At last, the framework will work better on self contained medium-size methods. In fast running methods (the ones that run in milliseconds), the overhead involved to contact the server will already outrun the local execution time. In big long running methods, we have always the risk of interrupting the connection because of network changes. In this case, the execution is lost and the framework will continue to execute locally starting over the execution.

Choosing the Offloading candidates is for now a task for the developer. Nevertheless the framework does not necessarily need manual candidate selection. With the help of static analysis one could develop a tool that can forward the Offloading candidates to the post-compiler guided by heuristics. This tool would select the optimal partitioning and include the Offloading library in the Android package's classpath. This way no prior interference is necessary on the development cycle.

3.3 Remote platform

Cloud computing has facilitated the virtualization of platforms and has increased the overall availability and speed of information. Our Offloading framework is very dependent on the cloud, and is extensively assisted by a powerful backend. We now present the main ideas and implementation of our remote platform. We may once more remind the reader that although this is a separated module, it has to maintain an interface contract with the Offloading library. Without this premise they are not able to communicate to enable Offloading.

3.3.1 Requirements

The main requirement of the remote platform is to be able to execute code remotely. As said before in Section 3.2, in order to successfully execute the same code provided by the mobile platform and achieve equal results, we need to first replicate the entire original environment in the cloud.

Fortunately, half of task is already being done by our post-compiler and the relevant method fields and objects are delivered via a serialized Java Map. The remote end will now need to only set these new environment changes. To be able to run Android code, we need as well a DalvikVM system. Finally, we must communicate efficiently with the mobile

[2]More about this in Section 4.

platform, exchanging objects and information. We can already outline the main requirements for the remote platform:

- Run a DalvikVM based remote platform to execute Android instructions;
- Develop a communication protocol, for client-server communication;
- Create an optimal serialization mechanism to exchange objects and state between platforms.

3.3.2 HTTP Server

To respond to requests made by the client, we opted on a widely known protocol that is notably used for its scalability and proven success: The HTTP Protocol. HTTP enabled us a fast startup, where we would immediately receive full compatibility with several devices and tools developed. In Android OS, we use the Apache HttpComponents [Apache HttpComponents, 2016] to provide us client capabilities. To actually start up a server and respond to requests we use NanoHTTP [2016], a minimal embeddable HTTP server. In NanoHTTP, one can simply extend its main classes and write the server logic in its own way.

3.3.3 DalvikVM cloud

The are several virtualization solutions available in the market: QEMU, VMWare, XEN, Virtualbox among others. Android also provides its own emulator for testing purposes that could be used for executing remote methods.

Preliminary tests with Android's emulator showed poor performance. The lag occurred by emulating the instructions does not make it a viable solution. Other fast emulators in the market like Genymotion [Genymotion, 2015] could have been used, but we opted for a true DalvikVM port to x86 to profit the most from hardware.

Android was built on top of an ARM architecture. When running Android on an emulator, the container has to emulate all ARM instructions to the native host architecture. To mitigate this, there are currently hardware acceleration support by the host that virtual machines make use to speed up their execution [3]. In a typical virtualized cloud environment, the hardware acceleration support is hidden behind the virtual layer, exacerbating lag in nested virtualized machines. Since our cloud's architecture is based on x86, we chose to use Android-x86 to avoid instruction translation. Android-x86 is an Android open source port to the x86 architecture. The original Android x86 project does not run in nested virtualized

[3]One example for Linux is KVM, for Windows there is Microsoft Hyper-V.

environments, that is, running the system in a virtual machine that is already running in a virtual environment. Works like Chen and Itoh [2010]; Kosta et al. [2012] modify Android x86 to run under these conditions. Because of time constraints, we did not do so, opting for a palliative solution.

We chose to run Android x86 on Virtualbox in a local server. To provide access to the mobile device from outside, we used Autossh [2016] to create a reverse tunnel with a cloud machine. Autossh provides a secure reliable tunnel between two servers. It uses SSH to do so. Example of the command line used:

```
$ autossh -M 0
   -o "ServerAliveInterval 30"
   -o "ServerAliveCountMax 5"
   remoteuser@<remote-platform-address>
   -R <remote-port>:localhost:<local-port>
```

This will forward any incoming requests from the remote platform to the local port chosen. The *ServerAliveInterval* and *ServerAliveCountMax* are parameters that will be passed to SSH to keep the tunnel alive.

After starting Android x86 inside Virtualbox and also setting up the server, the port binded by the HTTP server is not immediately visible outside Virtualbox. In order to actually access the HTTP server we use Android Debug Bridge (ADB), Android's own debugging tool that comes shipped with its Software Develop Kit (SDK). ADB can be used to accomplish various tasks such as analysing logs, installing and uninstalling applications. It can also forward chosen ports from the underlying local machine to an Android system. This can be done with the command:

```
$ adb -a forward tcp:<port> tcp:<port>
```

Figure 3.2 gives a complete overview of the above described process. It all starts with a request from the mobile client, it passes through the cloud and it is forwarded via reverse tunnel to our machine. ADB forwards this to DalvikVM and the request is processed. In this model, there is an additional overhead and lag due to this process. We would like to remind the reader that this is not an ideal setup for a production platform and Android x86 should be run directly on the cloud.

This architecture can be scaled to accommodate more demand. One could employ a Load Balancer [4] as cluster entry-point and run multiple Android-x86 server instances. The Load Balancer would receive the connection and forward them according to the load.

[4]A Load Balancer is an HTTP server that accepts client connections and forwards requests to the application server depending on some heuristic

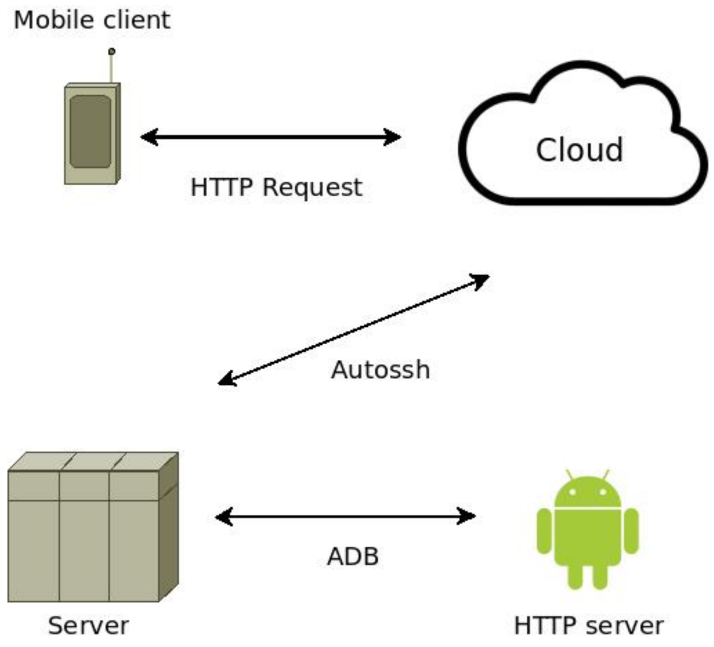

Figure 3.2. Remote platform overview

3.3.4 HTTP server

As said before, NanoHTTP was used as embedded HTTP server. We extend the main class to provide additional functionality and execute the desired method and return its result.

We receive from the application a Java Map containing the class instance (the *this* object), the local method arguments, and the static fields used inside the method. Before executing the method, we need to replicate the environment from the mobile device. The main algorithm for executing the code and return the result, works as follows: it receives the Map with the parameters. It iterates the Map, looking for the static fields inside. By Java reflection, it turns these fields public, if they were not already, and sets their values with the object from the respective map entry. After that, using once more reflection, it turns the desired method public and calls it using the arguments provided. If no *this* object is provided it assumes that the method is static and executes it in the same fashion. The result is stored again in a Map to be sent back to the client.

Algorithm 4 is the main algorithm behind the remote platform. In line 5, the static fields from the map are made accessible to platform and its values are set. Line 9 executes the desired method and stores the result in variable Res, in the next lines a result Map is created and the method result is inserted. In line 12 we compute a delta from the starting time and the ending time. This is necessary for later instrumentation about the execution on the remote platform. This data will be used inside the decision engine.

Once a method is executed by the remote platform, all methods called within this

Algorithm 4 Algorithm used to execute remote method and return results

```
 1: function EXECUTEMETHOD(Signature, Map)
 2:     $s \leftarrow CurrentTimeInMillis$
 3:     for all $f$ in Map.entrySet() do
 4:         if IsField(f) then
 5:             $TurnFieldPublic(f)$
 6:             $SetFieldValue(f, f.value)$
 7:         end if
 8:     end for
 9:     $Res \leftarrow ExecuteMethod(Signature, Map)$
10:     $ResMap \leftarrow$ new Map
11:     $ResMap.put("r", Res)$
12:     $ResMap.put("t", CurrentTimeInMillis - s)$
13:     return ResMap
14: end function
```

execution will be executed by the remote platform.

3.3.5 Serializing objects with Kryo

To make all this possible, we rely heavily in a serialization library: Kryo [Kryo, 2016]. Kryo is an efficient object serialization library for Java. Its main goals are speed and efficiency. It can perform deep object serialization and deserialization, also considering nested objects. The standard Android serialization library is often reported to be slow and inefficient, Kryo comes as a slight improvement.

3.3.6 Limitations

We should now discuss the limitations of our implementation. Firstly, we assume that the Server platform's address is already known by the mobile client. We purposely had no attempts on creating or using a discovery protocol, not to diverge our focus. This will be subject of future work.

The server platform is assumed to have the application code already available. An extra step would be necessary to make mobile code available on the remote platform. On the first contact the application could automatically send the binaries over the HTTP channel. Because of time constraints we chose not to develop this feature. We do strongly believe that this would not affect the library performance, since the binary transfer time would be amortised with multiple future requests, as it needs to be acquired only once.

Besides the result, we send no static fields nor the class instance back. It means that the method must be "free of consequences", not altering any inner static properties. This on the other hand is not a difficult implementation, as we would only need to send the data back and set it on the client. Additionally, the environment reproduced on the remote platform is not isolated per client, meaning that parallel requests may affect each other. Both problems need to be throughly addressed in the future.

The implementation does not attempt to add any encryption or authentication layer, leaving security for future work.

3.3.7 Conclusion

This chapter covered two modules of our proposal: The post-compiler and the remote platform.

The post-compiler is a tool used to modify the Android application and enable Offloading. It is executed after the development of the application and outputs a new APK ready to be installed by the end-user.

The remote platform runs the client code remotely. It accepts execution requests by creating an HTTP server, replicating the mobile environment and executing the method by reflection. This is done by a DalvikVM, the virtual machine that runs Android instructions.

Although these tools may be considered external, they both are nevertheless interconnected with the Offloading library, which is shipped with the application bundle.

In the next chapter we describe the Offloading framework, finishing all details of our proposal.

Chapter 4

Offloading Library

The Offloading library is the main module of the framework. It is embedded in the application as library, being responsible for monitoring, instrumenting, storing historical data and managing all the aspects of the framework. The post-compiler modifies the Offloading candidates, inserting calls to methods inside the Offloading library. With these entry-points, the library is able to record various attributes and reach a contextualized decision.

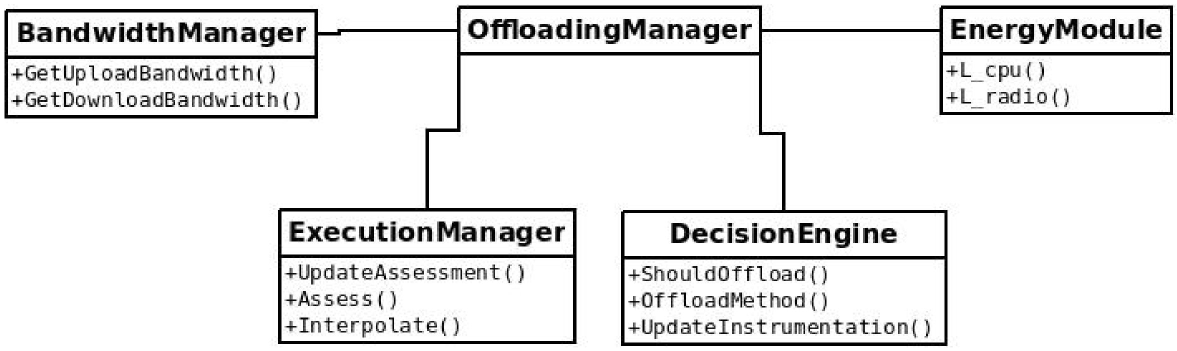

Figure 4.1. Overview of the Offfloading library

Figure 4.1 presents an overview of the internals of the Offloading library. As one may see, the *OffloadingManager* is the center piece of the diagram, being the entity that instantiates and starts the other classes. The *BandwidthManager* is responsible for monitoring bandwidth changes and storing historical bandwidth data. The *ExecutionManager* monitors method executions and their runtime duration, bytes transferred within the execution, result size and CPU cycles used. The information from both *BandwidthManager* and *ExecutionManager* is stored indexed by the method's signature [1]. The *DecisionEngine* is the brain behind the Offloading framework. It will reach a dynamic contextualized decision according

[1]The method's signature is a unique textual representation of a method in Java.

to the parameters collected. We can see three methods inside this class, those are the methods that are called by the modified method. The *EnergyModule* provides energy consumption estimation for CPU and Radio. It is used inside the calculations of the *DecisionEngine*.

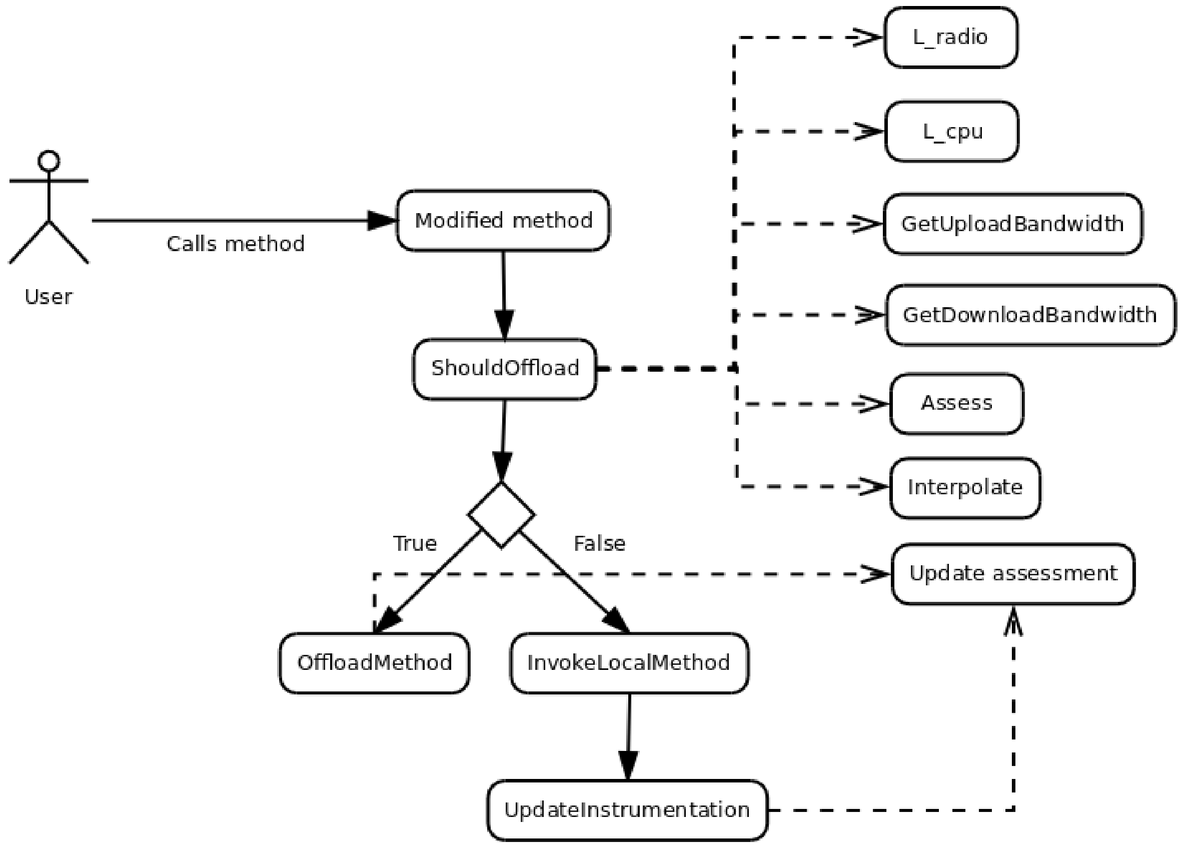

Figure 4.2. Overview of the Offfloading library

The methods declared here in the diagram will be called along the execution flow of the framework, having the *ShouldOffload* as entry point. Figure 4.2 contains the execution flow of the library. The dashed arrows mean dependency between the methods, and the solid arrows depict the execution flow.

4.1 OffloadingManager

The *OffloadingManager* is the main class of the framework. It is a singleton that is started together with the application. On startup it does the following:

1. Initializes the main framework storage, which is based in *SharedPreferences*, an Android system class. It is a key-value storage facility that can store strings. Each module

gets a reference to this persistent storage and is able to manage its own data.

2. Initializes the components *BandwidthManager* and *ExecutionManager*. The *Energy-Module* and *DecisionEngine* do not need special initialization.

3. Sets up listeners for phone state and network changes. This is necessary to know which network interface is currently being used and its attributes (Signal-to-noise ratio, SSID, signal strength, current bandwidth).

4. Initializes the log engine.

The *OffloadingManager* is used as middleman in the communication between modules. Other modules acquire the singleton instance of the *OffloadingManager* and access the properties and methods through it.

4.2 BandwidthManager

The bandwidth manager is the entity that stores and predicts localized bandwidth. The mobile device, besides having several different ways to access the internet, can face abrupt bandwidth changes. Since it is a portable device, the user can move around places, forcing the cellphone to switch between wireless networks and cellphone towers. Within a network or tower, the bandwidth may vary. The wireless network will have a different link speed, and the cellphone tower might be overcrowded, not being able to serve all clients properly.

Conventional frameworks do not account for this change, and can only update their bandwidth after facing this problem, taking sometimes a long time until it converges to the real value. We propose a different approach, using the user's location indirectly to have an educated guess on current bandwidth.

Firstly, to update the bandwidth values in the database, Equation 4.1 is used to avoid drastic changes, where bw is the newly measured bandwidth. This equation is used due to its proven success in smoothing RTT in the TCP protocol.

$$b(t+1) = b(t) \times (1-\beta) + bw \times (\beta), 0 \leq \beta \leq 1 \tag{4.1}$$

We can adjust β according to the application, currently the default value is 0.8. Methods $UpdateUploadBandwith$ and $UpdateDownloadBandwith$ used in Section 4.5 employ this method to acquire and store the new values.

The value bw is actually a **perceived bandwidth**. By storing the start time and the end time of any network data exchange event, we get the duration and the number of bytes exchanged. With this, we acquire a perceived bandwidth at that moment. To be more precise, we add an extra overhead incurred by the framework when serializing and performing the preliminary calculations of offloading. To accomplish this, we start our counting before the actual byte exchange, adding the time needed for all these additional operations. By doing so we are in fact lowering the real bandwidth value, but accounting for the framework's overhead in our equations. A network data exchange event can be defined as any request ULOOF makes, such as pings and remote execution requests.

If one would use GPS data or any other standard methods to locate the user, a significant battery penalty would occur [Carroll and Heiser, 2010]. To circumvent this, we use the WiFi's SSID and the cellphone tower LAC/CID to indirectly locate the device and use the contextualized bandwidth values. The tower LAC/CID is a unique tower identifier that we can efficiently store. As mentioned before, the framework acts differently when using different interfaces from the mobile cellphone, adapting itself to the location context.

We are aware that the SSID from a WiFi's access point (AP) is not unique, and will often be repeated in public places (e.g. chain stores that offer free WiFi). A more suitable identification would be the WiFi's AP MAC address. This was unfortunately not easily available to us, due to some erroneous Android OS implementation bug in our test device. We decided to use on our tests the SSID.

4.2.1 WiFi

When on WiFi, the SSID is used as identifying key. We match the SSID with the perceived bandwidth and store this value for later use. Android provides a useful function to retrieve the initial bandwidth, which is used by us.

4.2.2 4G

When on cellular connectivity, the framework acts differently in order to predict real values for the current bandwidth. A slightly modified version of Shannon's equation [Shannon, 2001] is used, together with the current SNR (signal-to-noise ratio) acquired in real time from the mobile device to estimate the user's bandwidth. A mobile tower serves a much larger area than a WiFi access point, so the bandwidth varies significantly based on location. A value S is recorded in the framework store together with LAC/CID, defined by:

$$S = bw \times log_2(1 + SNR), \tag{4.2}$$

where bw is the perceived bandwidth and SNR is the signal-to-noise ratio.

The result will be then later retrieved to provide a location aware bandwidth, which better describes the conditions at that location, taking into account the signal quality at the moment. To retrieve the bandwidth in a later time, the following reverse equation is used:

$$S' = S/log_2(1 + SNR). \tag{4.3}$$

A new educated guess is calculated according to new values of SNR and S, using Equation 4.2. The value is updated in every data exchange event, such as an Offloading request.

4.2.3 Main functions inside the module

To store the values for bandwidth, we use two different maps: upload and download. The values are saved separately indexed by the SSID or LAC/CID depending on the interface. The functions *GetUploadBandwidth* and *GetDownloadBandwidth* first check the current interface used, and return the value stored on the index. We also keep a general bandwidth value stored, in case the identifier is not already present in the map.

To update these values, functions *UpdateUploadBandwith* and *UpdateDownloadBandwith* employ the same indexing technique, using Equation 4.1 to avoid drastic changes.

4.2.4 Limitations

Our approach is still not completely mature and presents some limitations. Our model for locality while on 4G may be in practice closer to the real bandwidth provided, as we account for different bandwidths according to tower. But the real world scenario is more complex, and one can see bandwidth degrading on rush hours or at certain days. This happens because of uneven distribution of clients between cellphone towers. Modeling this would require employing a machine learning algorithm on top of our technique, most probably coupled with crowd-sourcing for learning each tower's quality of service along the day.

The Shannon equation is not used for WiFi due to problems when acquiring the WiFi's SNR while on user-level.

4.3 ExecutionManager

The *ExecutionManager* is the main store for past executions and the principal class concerning the estimation of method execution time. The *ExecutionManager* uses a black-box approach to correlate the method's inputs with its execution time and result size. In order to do so, we introduce the concept of *assessment*: a mapping between method's arguments and a Java *double* value.

4.3.1 Assessment

This work introduces the concept of input assessment in an attempt to model the method's results as a function of the method's arguments. The concept is simple: A list of arguments of any type is converted into a numerical variable. This variable is directly proportional to the method's complexity, meaning that given bigger assessments, longer execution time and result size can be expected. The algorithm behind this is the $Assess(M)$.

The framework allows direct developer influence on the assessment value. The developer may give hints and set the assessment value of each parameter through Java annotations.

When calculating the assessment value for a method execution, the default behaviour consists on iterating through each parameter and act according to its Java class. Table 4.1 shows the mapping from class to numerical value. After mapping the parameters the values are summed to obtain an assessment value. Modifying the default behaviour is very beneficial, as the most knowledgeable entity in regard to the code is still the developer. The analysis is always performed in execution time to receive different assessment values for different values of parameters.

The developer can add other classes in this mapping, and may also override the default behaviour by providing his own class converter. This is done as follows: The developer creates a class that implements the interface *AssesmentConverter* and annotates each parameter or the whole method referencing this class. This interface requires the implementation of two methods: *convertAssesment* and *convertAllArgumentsToAssesment*. The first will be called when assessing individual parameters. The second is to be used on method-level and will be called when assessing the parameters altogether.

The method *convertAssesment* accepts an object as input, and should return a double. The method *convertAllArgumentsToAssesment* accepts a list of arguments and should also return a double. If the developer wants a simple converter, he can annotate each parameter with its own interface implementation and modify the behaviour per parameter. If the mapping is more complex, the developer may annotate the whole method and implement the method *convertAllArgumentsToAssesment*. This method will receive all method parameters and should return the assessment value as a whole.

The function $Assess(M)$ works exactly this way, accepting both flows: even if both method and parameters are annotated, the function first evaluates the method through the function *convertAllArgumentsToAssesment*. If the return value is null, it then proceeds to evaluate the assessment value of each parameter separately, calling the custom converters if so annotated. The end value is then returned as assessment.

This offers the developer complete control over the Offloading candidates. The devel-

Table 4.1. Default assessment behaviour per parameter class

Class	Numerical value
Int/Double/Float/Decimal/Long/Short/Byte	Numerical value
String	String size
Collection/Iterable (List/Set/Map and others)	Collection size

oper is responsible for choosing the candidates and providing a proper assessment function, which is crucial for the framework's lifecycle. Not all methods are suitable for offloading. This is better explained in Section 3.2.6.

4.3.2 Historical executions and result prediction

With a proper mapping function in hands, bridging complex classes to a simple numerical value, we can now view the method as a black-box. We extract its information by feeding the system its inputs, and storing its results for future evaluation. This way, we can predict its behaviour simply by looking at the historical data available. For each execution we store the assessment value, the execution time, the result size, the number of CPU ticks used, the number of bytes transfered and whether the execution was local or remote. To obtain an estimation we interpolate the assessment for those yet unknown values. After the execution we update the store with the real measurements, converging at each iteration to the real value.

This data needs to be stored in an optimal way and a proper interpolator needs to be selected to avoid value fluctuation at each new insert. As interpolator we used the Akima spline [Akima, 1970]. Akima spline is a piece-wise interpolator that allows great flexibility in comparison to linear or cubic splines. Since it is a piece-wise function, it only gets influenced by its direct neighbor points, facilitating lazy updates. It allows at the same time the accurate evaluation near the original points.

We keep a spline for each type of measurement stored, having one for execution time, one for result size and so on. The local and remote points are stored separately, having also different splines and stores depending on the source of these points. To avoid building the splines at each new update, we created a lazy data structure: We store all points indexed by assessment in an AVL Tree. The AVL Tree is a self balancing tree that maintains key order. That makes the query of key ranges possible, keeping the operation in the magnitude order of $O(log \cdot n)$ [Baer and Schwab, 1977]. When any values are updated, we mark the keys as dirty and do not rebuild the spline. Before interpolating any key v, we first query the tree for the closest keys a and b, $a < v < b$. If a or b are marked as dirty, we first update the spline and mark all points as clean. If they are not, there is no need to build the spline again, since it holds locality for that range. Another good property of the usage of the AVL Tree, is the fact that we do not use the interpolator for assessing keys present in the tree. We use the values already stored by the tree without checking the spline.

Algorithm 5 Updates the assessment store

```
1: function UPDATEASSESSMENT(Assessment, Measurements[])
2:     $v \leftarrow$ new $Measurement(Assessment, Measurements[])$
3:     $K \leftarrow AVL - Tree.insertOrUpdate(Assessment, v)$
4:     $K.markAsDirty()$
5: end function
```

Algorithm 5 simply creates a new internal *Measurement* object, inserts or updates it at the AVL Tree, marking the point as dirty.

Algorithm 6 Interpolates the attribute given an assessment

```
1: function INTERPOLATE(Assessment, MeasurementName)
2:     $v \leftarrow AVL - Tree.find(Assessment)$
3:     if v is not NULL then
4:         return v.get(MeasurementName)
5:     end if
6:
7:     $(a, b) \leftarrow AVL - Tree.findDirectNeighbors(Assessment)$
8:     if a.isDirty or b.isDirty then
9:         $UpdateSplines()$
10:        $AVL - Tree.markAllAsClean()$
11:    end if
12:    $P \leftarrow GetSplineForMeasurement(MeasurementName)$
13:    return $P.interpolate(Assessment)$
14: end function
```

Algorithm 6 interpolates the values given an assessment and the measurement name. The measurement name will be the desired property e.g. execution time, result size. Line 2 looks for the key inside the tree. If the value is already present we can simply return it without further delay. Line 7 finds the direct neighbors a and b from the assessment. If any of them are dirty we update the spline and mark all points as clean. Line 12 gets the spline and interpolates the value.

Our implementation does not make use of any eviction policy so far. These could be a least recently used policy (LRU), most recently used policy (MRU) or even a least frequently used policy (LFU). An eviction policy would be quite important, in case the volume of executions reaches a certain threshold. It would avoid therefore memory and space overloading on the mobile in such situations. This is left for future work.

4.3.3 Limitations

The *ExecutionManager* deals with estimations. Under such scenario, it is important to discuss the limitations of this module.

The first aspect is the assessment of the execution time. The proposal assumes that larger inputs (e.g. larger integer numbers or larger vectors) will usually generate a longer execution time. This is a reasonable assumption for some problems. However, many problems do not follow such a premise, e.g. prime number calculation, where Mersenne numbers (numbers in the form of $2^n - 1$) are much easier to check than ordinary numbers.

Next, the Offloading engine improves its decision with usage, since more data points will generate a more precise estimate. Thus, when more data points exist for the same method and the same location, the estimation will be improved. Conversely, calculations that are run occasionally will have worse predictions. One way to refine the decision is to use crowd sourcing, that is, a server would aggregate the data points related to the execution of many users of one or more applications. This server would periodically upload to the mobile devices the most recent CPU execution curves and even the cellular speed estimates.

4.4 EnergyModule

The *EnergyModule* is the entity that estimates energy consumption of CPU, WiFi and 4G. Some of the other Offloading frameworks monitor other cellphone components like the screen, GPS and Accelerator. We chose to simplify our approach and monitor only two components: CPU and Radio. We find that the relevant problems that are intrinsically connected to Offloading do not make use of these other components. A method that is executed remotely does not have access to local hardware or mobile state. Evaluating the energy consumption of the Offloading candidate yielded by these other modules would not be logical, since they are not present in the remote platform.

This work also aims to lower overhead. Instead of actively monitor the components and get an estimation in a background job, we map attributes from these components that are readily available to us. We sampled these attributes directly from our test device and built a correlation that corresponds to the real instant energy consumption. To obtain the execution's total energy consumption we multiply this value by the time duration of the method execution.

The idea is to map CPU ticks or number of bytes transfered within a method execution to a consumption in milliwatts. For each component, we justify the choice of attributes, keeping in mind that these are all available to user-level applications.

- **CPU**
  The CPU is shared among many processes including the operating system. Execution time alone would not be the optimal attribute to perform the mapping, since the CPU can be busy with other processes while executing the current method. Android provides a very good measurement for this purpose: CPU ticks. The CPU tick is a kernel normalized value that will correspond to the workload. The system provides a tick counter for each process separately. The CPU ticks are normalized to 100 ticks/second. It means that to acquire the average workload of the method, we simply need to take the delta of the CPU ticks, before and after the method, and divide by the number of seconds.

- **Radio**
  For the radio component, time would also not be a suitable measurement. Besides the fact that other applications may be using the module, which would slow down transfer time, we can also have lags on the server side. Moreover, the energy curve only changes by varying the byte traffic [Miettinen and Nurminen, 2010]. Therefore we choose the number of bytes exchanged by the method to perform the mapping.

If both part's instant energy consumption could be measured in the real device, one can calculate the total energy consumption of the method by multiplying it by the total execution time. This can be based on CPU ticks and bytes transferred, which can express the module's total resource usage by method execution without being time dependent.

We define in this module two functions: l_{cpu} and l_{radio} that will be later used by the decision engine. They represent the instant energy consumption by attribute. The functions were derived from our test device, a Samsung S5, and they are chipset specific. It means that if we change the mobile platform's chipset these functions are not accurate anymore. This approach has the positive point of not imposing any restrictions regarding the device. The sampling can be repeated in different chipsets in the future if needed. A public repository may also be employed here in order to store this chipset dependent data. The framework would then automatically download the profile at first contact.

Our first approach was to measure power consumption via software. Android is a unix based operating system. Inside its */proc* directory, one may find some information regarding hardware state. Nevertheless, preliminary tests showed that energy readings are very inaccurate and sometimes even impossible. Qualcomm provides Trepn [Trepn Power Profile, 2016], a software power profiler that works for Qualcomm chipsets. Unfortunately it does not work for our test device, a Samsung S5, since the manufacturer's hardware interface implementation does not provide the needed data. Inside the website's forum we have found a thread concerning this [Trepn Power Profile Forum, 2016].

We also tried PowerTutor [Zhang et al., 2010], which looked very promising at first glance. PowerTutor monitors each component separately watching for current and voltage changes. It monitors CPU ticks, bytes and many other attributes, evaluating the consumption dynamically in runtime. We abandoned the idea after realizing that the project is already some years old and was not updated since then. It also does not support 4G. Internally it uses a device specific table to map measurements to energy consumption. Our test device chipset was not directly supported, meaning that we would need to reverse engineer the real meaning of each map entry and set the correct values according to their equations. Finally, even if this was possible, as mentioned in the previous application, the Samsung's hardware interface implementation is incomplete and does not provide correct values for some measurements.

The idea is to develop an approach that would work for any device and would still be inside the project's financial and time budget. We opted for a commodity hardware measurement device: The inexpensive KCX-017 was used to measure mobile energy consumption.

Figure 4.3. KCX-017 - Power measurement.

The device is placed between the mobile charger and socket. It outputs the current voltage in volts and current in milliamperes. It refreshes every second and it has no connection to the computer, providing its readings only through a small LCD display.

Initial experiments showed that using the computer's USB port as energy source does not result in usable readings, keeping the current at the same level at all times. This happens because of current normalization that the computer provides.

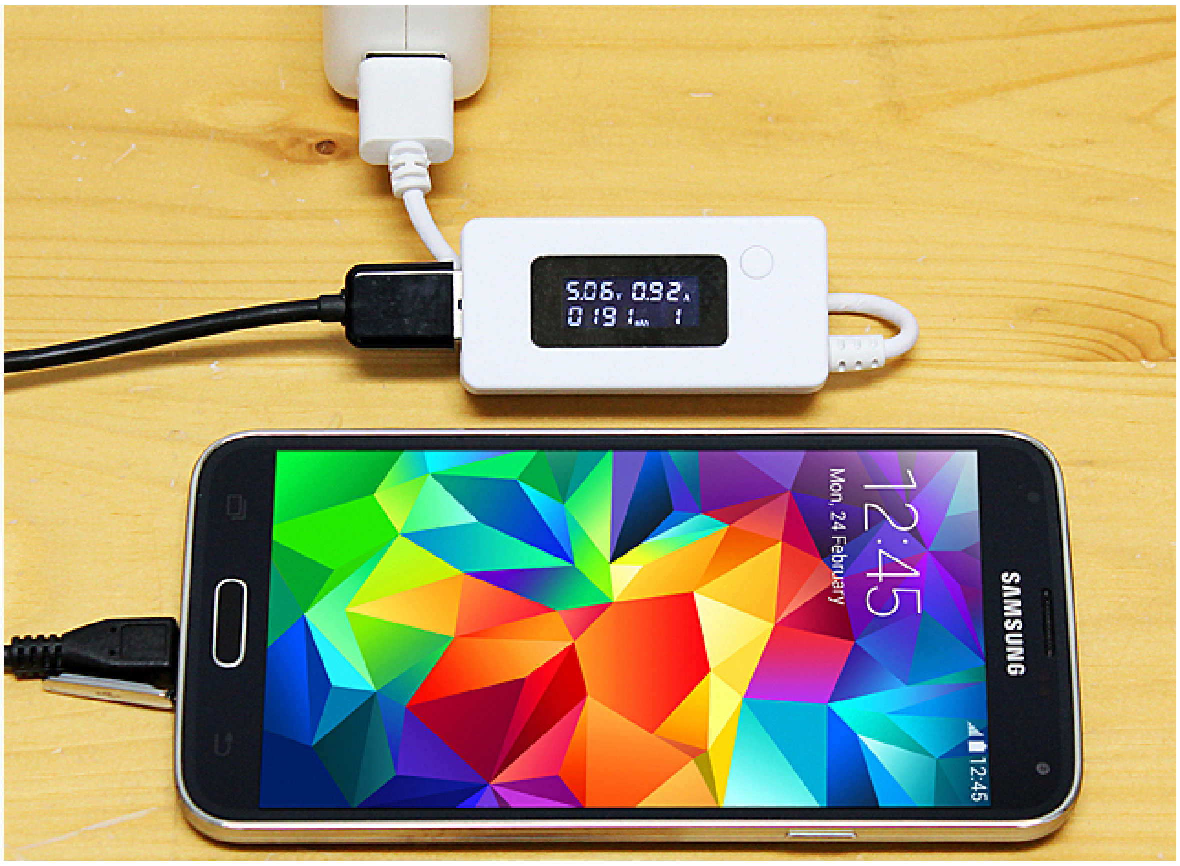

Figure 4.4. KCX-017 - Measuring the phone's energy consumption

The integrated circuit of modern phones account that the battery should be always on when charging. This means that no tests can be performed without a full battery charge, since it will be also drawing additional current to charge itself.

To obtain the distributions of the two modules, two distinct Android applications were created. They test the two different modules separately, monitoring the CPU usage and the bytes exchanged in the operation. All the tests were performed in a Samsung S5 (same device later used in experiments) with all other applications closed and with the airplane mode on when testing CPU. This is necessary to favor isolation and to not affect the test results.

4.4.1 CPU

The application created to test CPU spawns threads to achieve the desired load. The mobile device under test has 4 CPU cores, meaning that 4 threads would make almost full use of the CPU. The algorithm multiplies two big random prime numbers. We get the straightforward sense that any load will use partially $load/25$ cores, $0 \leq load \leq 100$, plus $(load \mod 25)$

percent of the additional core.

To partially load a given core, a small sleep is set every x iterations of the thread. This was calibrated iteratively using the mobile phone. Sleeping 100 milliseconds every $5^{(load \mod 25)/5}$ iterations was found to work well for some given fractioned load $0 \leq load \leq 100$.

We chose the following set of loads for our tests: $L =$ [10, 25, 35, 50, 60, 75, 87, 100] representing CPU load percentage. Each load L_i was kept for two minutes in order to obtain sufficient amount of data.

4.4.2 Radio

For radio, an application and a web server were developed to exchange traffic. Preliminary tests were first performed by varying the total transfer time, keeping the number of bytes fixed. This showed to be unfruitful and the current barely modified along the experiment, confirming the work from Miettinen and Nurminen [2010]. This behaviour is expected, since the radio can keep a low power state during small network loads.

A second batch of tests were performed by programming the web server to receive a total transfer time and a delay parameter. It creates a delay of 1 millisecond every x iterations, lowering the bandwidth and thus forcing the radio to exchange less or more bytes. This showed to reflect the expected result, consuming more energy when exchanging more bytes.

A delay d of 1 millisecond every [0, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 76800, 153600] bytes was used. The experiment was made for each d with a duration of 2 minutes each.

4.4.3 Collecting measurements

The KCX-017 does not offer any interface to collect data. It has to be visualized through the LCD display. To read measurements one would have to sample manually and note down the values periodically. This is of course not feasible when we face experiments that can take up to 1 hour. To circumvent this problem, a script was created to take pictures of the display in a fixed interval. The time of the snapshot is outputted as filename in a pattern fashion. The idea is to interpret these values with an OCR software and match them with the application logs.

Another simple script was additionally made to rotate and crop the image. Because our OCR tests were performing badly, we additionally had to negate and erode the initial image. This was made with the Imagemagick Toolkit [2016].

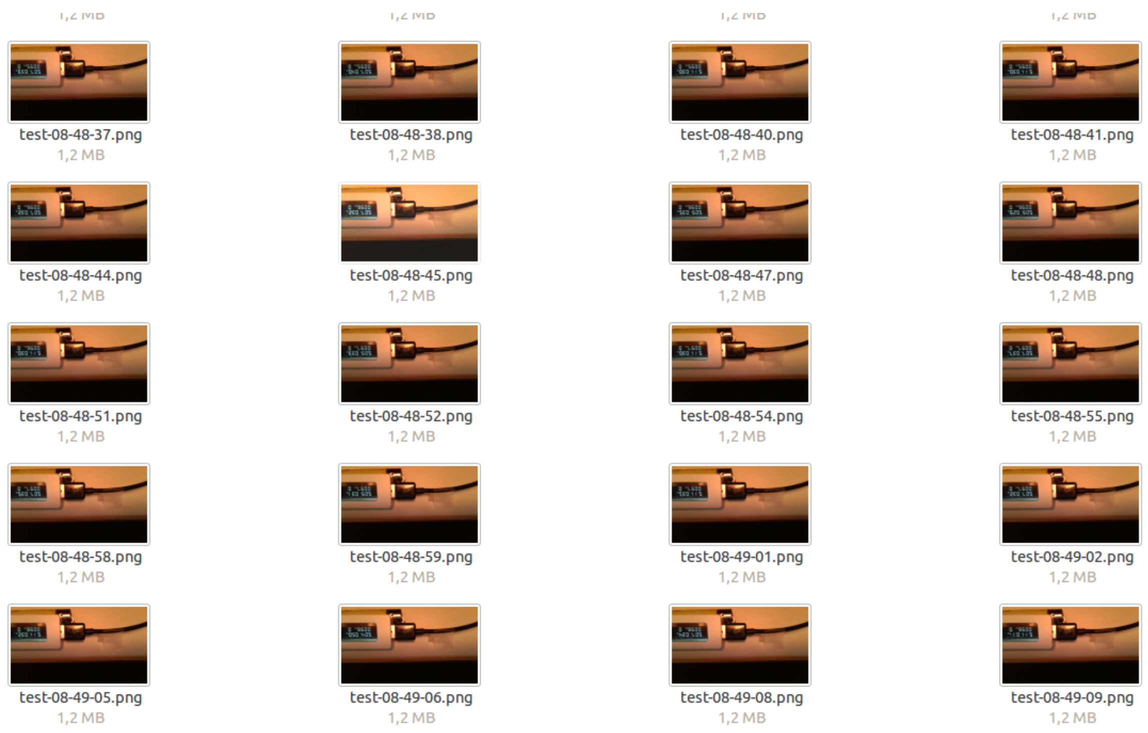

Figure 4.5. Folder with pictures taken from the KCX-017

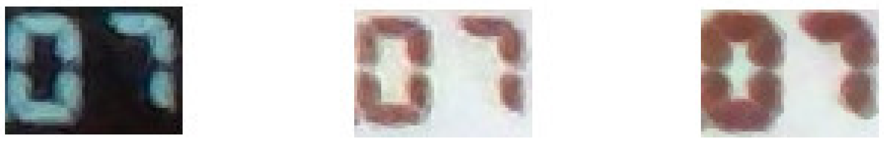

Figure 4.6. From left to right, cropped image, then negated, then eroded with a diamond shape

Using the OCR tool *gocr* for unix systems, the images were recognized and the measurements for voltage and current were acquired with its time stamp residing in the filename. With this we have all the data needed, since we can now match the timestamp in the application logs together with energy consumption measurements.

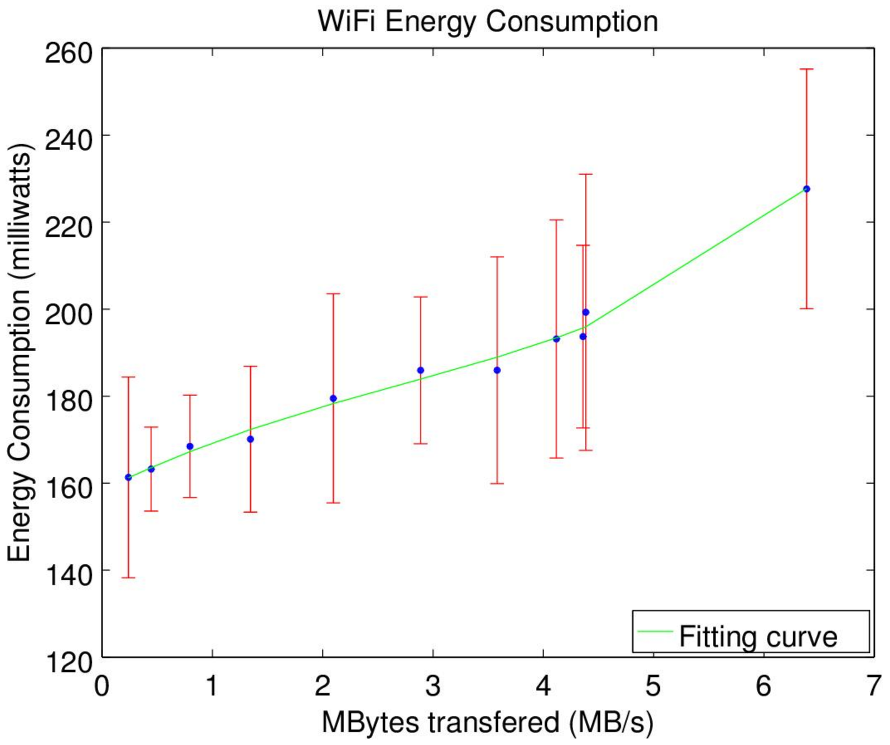

Figure 4.7. Power consumption for WiFi together with its standard deviation

4.4.4 Results

The results followed expectation: CPU spends roughly more energy than the radio under full load. For radio, as we can see in Figure 4.7, energy consumption increases as we transfer more bytes. The results seen here were done by getting the mean value of all measurements within the range of the experiment. The standard deviation is also plotted in red.

In Figure 4.8 we can see the scatter plot of all values obtained by the measurement. In green at the bottom, the CPU load is also plotted, evidencing the expected CPU usage despite the high energy consumption.

The same test was performed using 4G, receiving similar data. We notice that when transferring big size data wirelessly, WiFi does perform better, spending less energy.

One can clearly notice the upward trending curve. We can now use this distribution and do a curve fitting for finding the function that better describes this dataset. The found

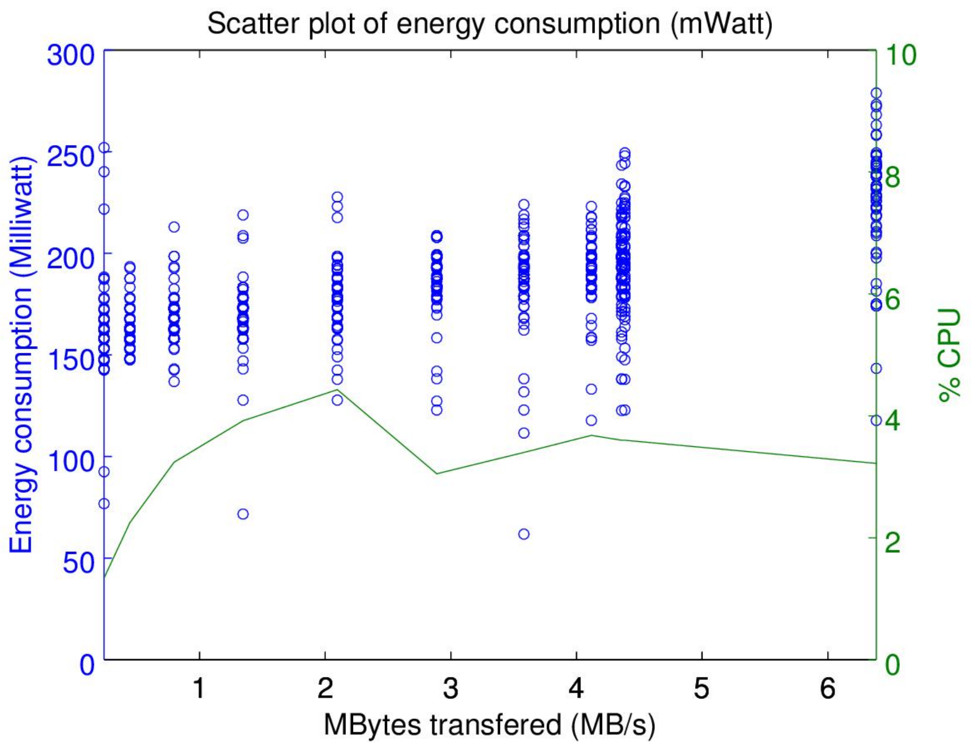

Figure 4.8. Scatter plot of WiFi energy consumption, in green the CPU load in $\%$

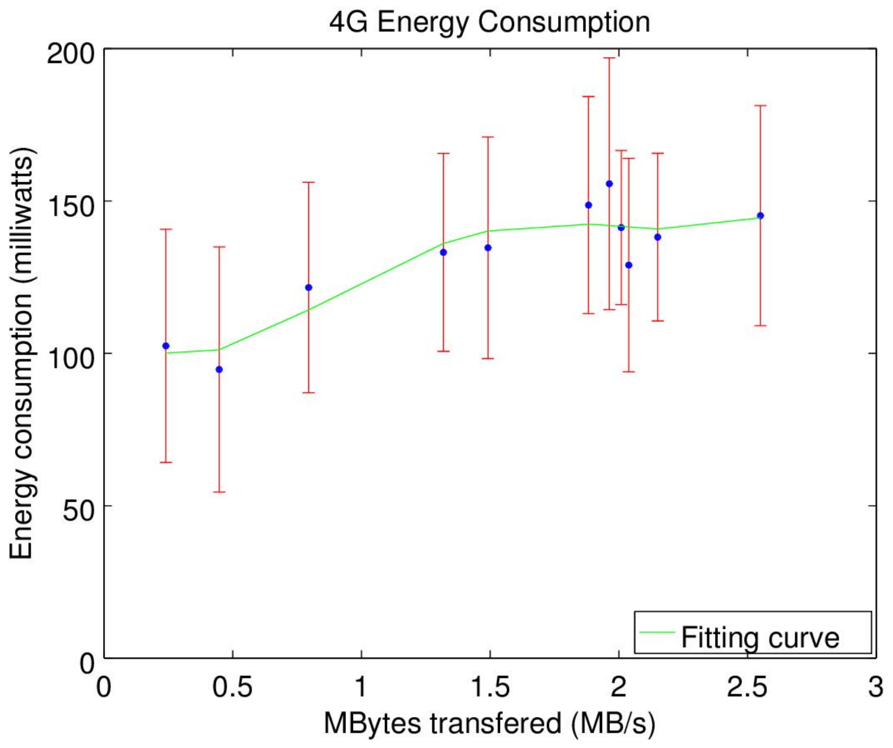

Figure 4.9. Power consumption for 4G together with its standard deviation

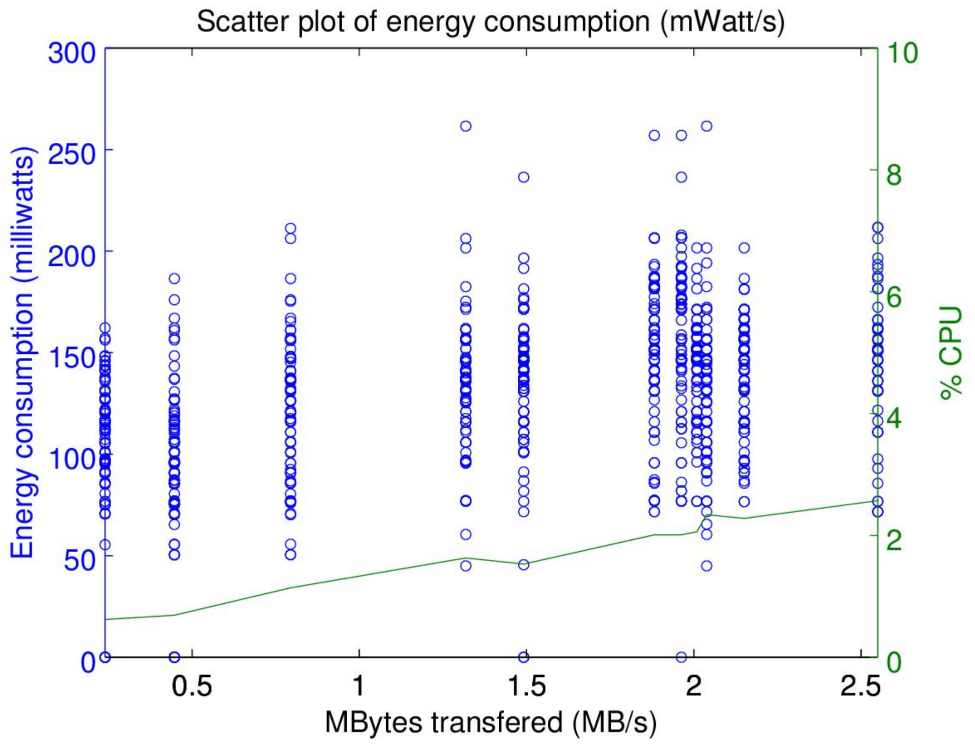

Figure 4.10. Scatter plot of 4G energy consumption, in green the CPU load in $\%$

curve is:

$$l_{radio}(s) = \begin{cases} +158.37 + 1.1811 \cdot 10^{-5} \cdot s^1 \\ -1.4722 \cdot 10^{-12} \cdot s^2 \\ +6.1454 \cdot 10^{-20} s^3 \\ +1.8794 \cdot 10^{-26} s^4, & \text{if using WiFi} \\ \\ +111.24 - 7.9499 \cdot 10^{-5} \cdot s^1 \\ +1.5999 \cdot 10^{-10} \cdot s^2 \\ -8.3738 \cdot 10^{-17} \cdot s^3 \\ +1.3748 \cdot 10^{-23} \cdot s^4, & \text{if using 4G} \end{cases} \tag{4.4}$$

For the fitting curve in Equation 4.4, we found a coefficient of determination of 0.99020 and 0.84887 for WiFi and 4G respectively. This curve can be found plotted at Figure 4.7 and 4.9. When evaluating l_{radio} in runtime, the current interface will be verified and the appropriate value will be used. An important detail: Although we plotted our graph in MBs for simplification, this equation accepts bytes as unit.

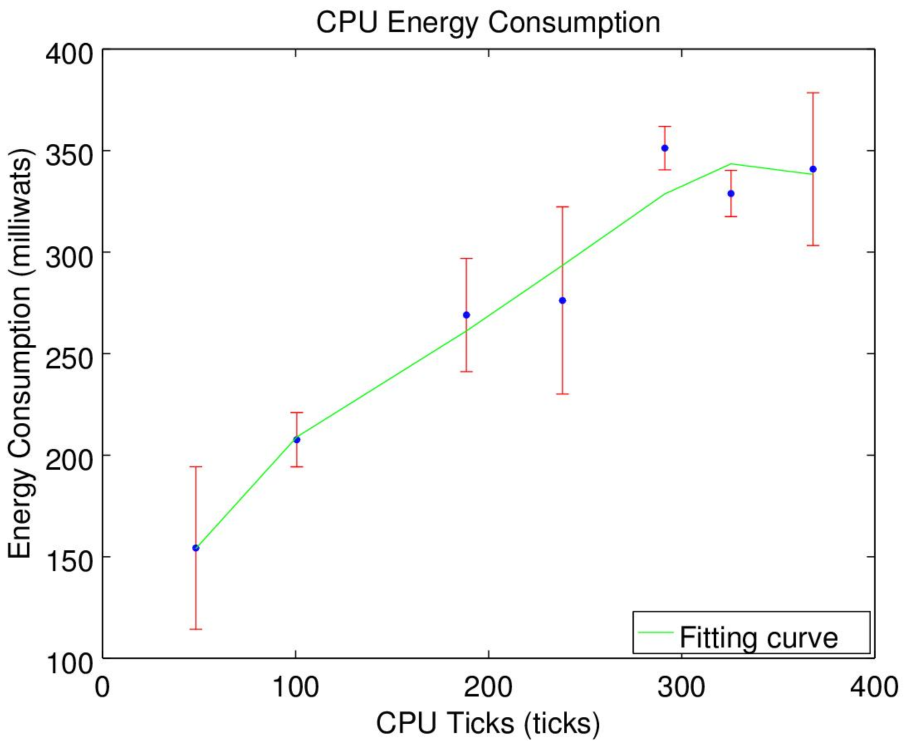

Figure 4.11. Power consumption for CPU together with its standard deviation

The same happens with CPU: As the application uses more resources, one can see the system consuming more energy.

In Figure 4.11, there is an upward trend reaching a plateau at the end as we achieve almost full load. The graph is represented in CPU ticks, which is normalized and abstracted by the Unix kernel as a substitute for Hertz. In this normalized value, a second will represent 100 CPU ticks. The fitting curve is also plotted at this graph and is defined as follows:

$$\begin{aligned} l_{cpu}(s) = & + 51.422 + 2.9076 \cdot s^1 \\ & + 0.019306 \cdot s^2 \\ & + 6.7841 \cdot 10^{-5} \cdot s^3 \\ & - 8.4491 \cdot 10^{-8} \cdot s^4 \end{aligned} \tag{4.5}$$

The fitting curve in Equation 4.5 has a coefficient of determination of 0.96605. We have now defined our two main equations that will be used by the *DecisionEngine* in section 4.5.

The new generation of mobile devices are equipped with powerful math coprocessors that can efficiently calculate complex math expressions. This does not hold true for all

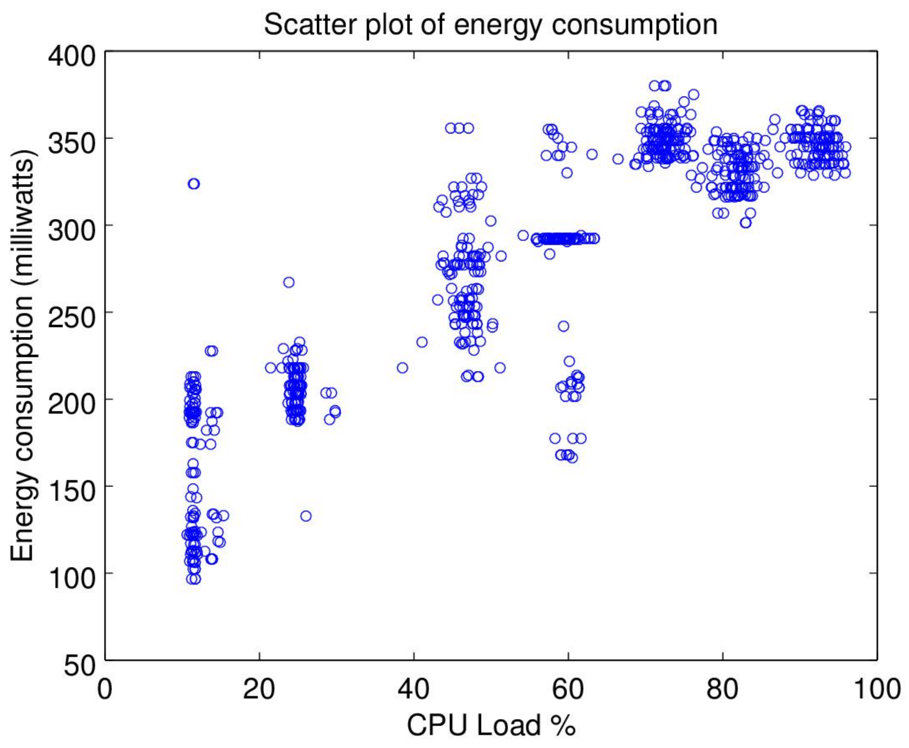

Figure 4.12. Scatter plot of CPU energy consumption

equipments, where the degree of the fitting function would make a considerable burden when performing the calculations. We chose a fourth degree fitting function to keep the equation small, while still maintaining accuracy. Making the function with two degrees results in a coefficient of determination of 0.95701 for CPU and 0.82454 for 4G as an example, slightly lowing accuracy.

4.4.5 Limitations

The energy model is chipset dependent and should be generated according to the desired mobile target. Equation 4.4 shows different output for WiFi and 4G and expects no interface changes within the method's execution. If this happens, the real bandwidth and energy consumption will be different from the prediction. This is not a serious limitation, provided that the Offloading candidates are not extensive methods with big execution time (i.e. with minutes as time unit). A discussion about the Offloading candidates may be found in section 3.2.6. The framework will work better if used in smaller self-contained sub functions. This diminishes the time window, which the user may change from WiFi to 4G or vice-versa.

4.5 DecisionEngine

The brain behind the Offloading framework is the *DecisionEngine*. The intelligence and decision making of the framework lie within this module. The Offloading framework is very dependent on context. There are situations where offloading a method will result in more usage of resources, having the optimal decision as local execution [Saarinen et al., 2012]. These bad decisions may accumulate, yielding in poor performance in comparison to the unmodified application. For this reason the decision engine is a crucial part of this framework.

This module implements the decision engine and stitches the remaining parts of the framework together, using the components inside the Offloading library to aid in this task. We chose to detail it as last, since every other module will be used for different purposes:

- *OffloadingManager*:
  Used as main access entity, storing references to all other modules. Also stores the constant α and the server URL and port.
- *BandwidthManager*:
  Will be used to estimate transfer time of the serialized objects and to store past perceived bandwidths according to location.
- *ExecutionManager*:
  Used to estimate execution time, result size, CPU ticks and byte volume, based on previous iterations. This module will also be useful when evaluating the assessment value and storing past executions.
- *EnergyModule*:
  This module is used to calculate instant energy consumption based on CPU ticks and bytes transfered. It is widely used in the equations of internal parts of the decision.

The proposed decision engine models the offloading problem using utility functions for energy and time, designed to work together. Each utility is weighted according to α. This builds a trade-off scenario between execution time and energy consumption, applied at tasks that may consume less energy when offloaded, but may take longer due to the delay when transferring its arguments and results.

A constant α, $0 \leq \alpha \leq 1$ sets the decision. A smaller α will make the decision tend towards saving energy, a bigger alpha will improve responsiveness.

The main equations for the boolean decision are presented below:

$$L(M) = \alpha \cdot tm_l(M) + (1 - \alpha) \cdot en_l(M) \tag{4.6}$$

$$R(M) = \alpha \cdot tm_r(M) + (1 - \alpha) \cdot en_r(M) \tag{4.7}$$

Where M is the Offloading candidate method, $L(M)$ is the estimated utility of local execution, and $R(M)$ is the utility of remote execution. The utility functions assess the time (tm) and energy (en) of the execution of M. Those functions will be detailed in further subsections. The decision chooses the alternative with the smallest execution cost. That means that the framework will offload the method M, when:

$$R(M) < L(M). \tag{4.8}$$

4.5.1 Modeling time

As mentioned before, we use the *ExecutionManager* from Section 4.3 to predict execution time from methods given their arguments. This black-box approach offers very low overhead, and uses attributes that are easy to monitor from the user-level side. We use the Akima spline interpolator to achieve this prediction. The local equation for time is as follows:

$$tm_l(M) = Interpolate(Assess(M), \text{"Execution-Time"}) \tag{4.9}$$

Equation 4.9 will simply get an educated guess according to previous runs as described in Section 4.3. The second argument passed as string, "Execution-Time", is necessary to specify the attribute being retrieved. Note that the function *Assess* is already defined in the same section previously referenced.

The same approach is used for the remote execution time, gathering the execution time from the server and interpolating this value. On the remote platform additional uploading and downloading of the method's serialized arguments is taken into account. The location aware historical empirical data is used from past executions and with each run, the bandwidth is calculated and added to this database as explained in Section 4.2. When on WiFi, bandwidth values from the same access point's *SSID* address are used. When on mobile data, the tower's LAC/CID (unique tower identifier) is recorded, together with its signal-to-noise ratio (SNR) to later calculate expected bandwidth values.

It is not possible to interpolate values when not enough data is available. To circumvent this problem, in such cases the system will randomly pick a choice, with an offload probability of 50%, in order to generate data until it can make a valid decision. This is the discovery phase. It will of course depend on network state: If Internet connectivity is not present, the decision will always be not to offload.

Finally, the remote execution time follows Equation 4.10:

$$tm_r(M) = Interpolate(Assess(M), \text{"Execution-Time"}) + trf(M.args) \tag{4.10}$$

$$trf(o) = size(o)/bandwidth \tag{4.11}$$

Equation 4.11 represents the calculation of the transfer time of serialized object o, in this case the method's arguments.

4.5.2 Modeling energy

To model energy, the module *EnergyModule* is used. We only monitor two main cellphone components: CPU and Radio. From these components we extract the CPU ticks and bytes transferred during the execution of the method. We then feed the *EnergyModule* with these parameters and receive the instant energy consumption per component. We multiply this by the method execution time to get its average energy consumption for the execution.

In the *DecisionEngine* we define the function $e : \mathbb{R} \rightarrow \mathbb{R}$, receiving as input the number of bytes or the number of CPU ticks spent at the method, outputting the method's consumption in milliwatts. The equations $en_l(M)$ and $en_r(M)$ can be defined as:

$$\begin{aligned} en_l(M) = e_{cpu}(&Interpolate(Assess(M), \text{"CPU-Ticks"})) \\ &+ e_{radio}(Interpolate(Assess(M), \text{"Bytes"})) \end{aligned} \tag{4.12}$$

$$en_r(M) = e_{radio}(size(M.args)) + e_{radio}(Interpolate(Assess(M), \text{"Result-Size"})) \tag{4.13}$$

The mapping function e, will consist basically in two parts: CPU (e_{cpu}) and radio (e_{radio}). They both use the equations l_{cpu} and l_{radio} from Section 4.4. In Equation 4.12, we add the energy consumption of the CPU and we also add the consumption of the radio component. This happens because the local execution may exchange bytes as well. In Equation 4.13, we add the number of bytes from the arguments of method M, together with the number of bytes of the returned object. All these attributes are retrieved through the interpolating technique describe in Section 4.3.

$$e_{cpu}(M) = l_{cpu}(M.ticks/M.runtime) \cdot M.runtime \tag{4.14}$$

$$e_{radio}(bytes) = l_{radio}(bandwidth) \cdot trf(bytes) \tag{4.15}$$

As one may see, the attributes are divided by the total time spent in the operation, that being the time spent transferring bytes and the execution time respectively. We input this to our function obtained from the distribution, we get the instant energy consumed in milliwatts and multiply this by the operation's total time again. This then successfully describes the average method energy consumption.

4.5.3 Algorithms involved in the decision

With the problem modeling done, we can now proceed to describe the concrete implementation. The algorithms make use of the methods defined in previous sections. *ShouldOffload* implements the method that decides whether Offloading is the optimal choice at the moment. *OffloadMethod* implements the logic of offloading a method. *UpdateInstrumentation* is charged of updating information about the method after a local execution.

The Algorithm 7, ShouldOffload, receives as input the method signature and a map with the arguments, the static fields and the *this* object. We recall that the algorithm gets called within a method execution, which was modified by the post-compiler in Section 3.2.

The algorithm returns true or false, indicating whether to offload or not. It should return true if the method is suitable to be offloaded at the moment, and false otherwise. In line 3, it first checks if the remote platform is available. If it is not available we cannot offload. False is returned directly if this is the case. In line 6, a second check is made, assuring that we have enough data to interpolate the method. In some cases, we do not have the exact assessment value at our past executions, or we do not have enough data to build our spline. We return a fair random boolean response [2] in this case. In line 17 we calculate the required time to transfer the map with arguments and the *this* object. We also account for the time spent on serializing the map. Line 21 fetches the interpolated value for execution time. Line 25 calculates the local energy consumption for the CPU and radio. One may notice that some values are divided by 1000. This is necessary to bring all measurements to the same unit: seconds. In line 29 we then calculate the remote energy consumption, taking into account uploading the map and downloading the response. Finally we mimic Equation 4.8 by returning the boolean answer.

In line 26 we generalized the bandwidth by assuming the worst case scenario: the minimum bandwidth. This is a simplification in our implementation, since we did not want to separately monitor incoming and outgoing bytes for the specific case of the local execution.

[2] 50% of probability for each answer.

Algorithm 7 Main decision algorithm. Outputs a boolean decision according to the current context

```
1:  function SHOULDOFFLOAD(M, Map)
2:      $assessment \leftarrow Assess(M)$
3:      if not $RemotePlatformAvailable()$ then
4:          return false
5:      end if
6:      if not $CanInterpolateLocalAndRemote()$ then
7:          return $GetRandomResponse()$
8:      end if
9:      $bw_{up} \leftarrow GetUploadBandwidth()$
10:     $bw_{down} \leftarrow GetDownloadBandwidth()$
11:     $minBW \leftarrow Minimum(bw_{up}, bw_{down})$
12:     $serializedObj \leftarrow SerializeArguments(Map)$
13:     $cpuTicks \leftarrow local.Interpolate(assessment,$ 'CPU-Ticks'$)$
14:     $bytesLocal \leftarrow local.Interpolate(assessment,$ 'Bytes'$)$
15:     $resultSize \leftarrow remote.Interpolate(assessment,$ 'Result-Size'$)$
16:
17:     $uploadAndDownloadArguments \leftarrow Size(serializedObj) \ / \ bw_{up} + \hookleftarrow$
18:                                          $resultSize \ / \ bw_{down} + \hookleftarrow$
19:                                          $TimeToSerialize(Map)$
20:
21:     $t_l \leftarrow local.Interpolate(assessment,$ 'Execution-Time'$)$
22:     $t_r \leftarrow remote.Interpolate(assessment,$ 'Execution-Time'$) + \hookleftarrow$
23:                             $uploadAndDownloadArguments$
24:
25:     $energyCPU_{local} \leftarrow l_{cpu}(cpuTicks \ / \ (t_l \ / \ 1000)) \cdot t_l / 1000$
26:     $energyRadio_{local} \leftarrow l_{radio}(minBW \cdot 1000) \cdot bytesLocal \ / \ (minBW \cdot 1000)$
27:     $e_l \leftarrow energyCPU + energyRadio$
28:
29:     $e_r \leftarrow l_{radio}(bw_{up} \cdot 1000) \cdot Size(serializedObj) \ / \ (bw_{up} \cdot 1000) + \hookleftarrow$
30:             $l_{radio}(bw_{down} \cdot 1000) \cdot resultSize \ / \ (bw_{down} \cdot 1000)$
31:     $u_l \leftarrow t_l \cdot \alpha + e_l \cdot (1 - \alpha)$
32:     $u_r \leftarrow t_r \cdot \alpha + e_r \cdot (1 - \alpha)$
33:     return $u_r < u_l$
34: end function
```

Another interesting fact, is how the bandwidth is stored. It is not the bandwidth per se, but simply a value that describes the amount of bytes that can be exchanged per millisecond. This is very handy in these equations, as one can simply divide by the size to get an estimation of transfer time. It is used directly in l_{radio} multiplied by 1000, since we want the nominal value in seconds.

The fact that the framework is location-aware, fosters favourable conditions for the converging of real bandwidth values. The precision of the predictions are calculated in the next chapter.

The next algorithm is the *OffloadMethod*, which will do the heavy lifting of the framework. It communicates with the remote platform and executes the method remotely, returning the result. It also updates the statistics with the *ExecutionManager* and *BandwidthManager*.

Algorithm 8 Offloads the method, calling the remote platform, updating the pertinent modules and returning the result

```
function OFFLOADMETHOD(M, Map)
    $s \leftarrow Serialize(Map)$
    $(res, uploadBandwidth, \hookleftarrow$
            $downloadBandwidth) \leftarrow UploadToRemotePlatform(Map)$
    if $res$ is NULL then
        Throw Exception
    end if

    $executionTime \leftarrow res.get($"t"$)$
    $result \leftarrow res.get($"r"$)$
    $remote.UpdateAssessment(M, [executionTime, Size(result)])$
    $UpdateUploadBandwith(uploadBandwidth)$
    $UpdateDownloadBandwith(downloadBandwidth)$
    return result
end function
```

The Algorithm 8 is called when the client decides to offload. It receives a Java Map with the arguments and the *this* object. The algorithm basically needs to keep track of various attributes of the remote execution such as execution time, bytes exchanged and result size. It then updates the respective Offloading library modules to allow their estimations to converge to the new values and returns the result.

Line 2 serializes the Map. Line 3 uploads the Map to the remote platform and receives several variables: the result, the perceived upload bandwidth and the perceived download bandwidth. If the result is null, we throw an exception that needs to be treated. A null result would mean a communication failure or a server-side exception. If an exception is thrown at this point, the framework proceeds to execute the method locally. This can be seen handled in Section 3.2 by the Algorithm 2. Line 9 retrieves the result object and the remote execution time from the result Map. This is done according to Algorithm 4. Line 11 and the ones that follow, update the bandwidth and the assessment value with the new result size and remote execution time. Finally the result is returned.

A communication failure may mean an unforeseen problem with the network or a timeout issued by the connection. The default values for the HTTP connection are set for 5 seconds for connection timeout (maximum time for starting the connection) and 5 minutes for the socket timeout (maximum time with the connection idle). The developer may modify this according to the application. As explained above, if any of these errors occur, the execution continues locally.

At last, we define the algorithm *UpdateInstrumentation*. It updates the respective modules with the values supplied for the local execution.

Algorithm 9 Updates the module *ExecutionManager* with the local execution metrics

```
1: function UPDATEINSTRUMENTATION(M, [ executionTime, cpuTicks, bytesExchanged ])
2:     $local.UpdateAssessment(M, [executionTime, cpuTicks, bytesExchanged])$
3: end function
```

Algorithm 9 simply updates the local execution store with the metrics received. It updates the values of execution time, CPU ticks and bytes exchanged by the local execution.

4.5.4 Limitations

The decision engine described in this section references all modules and it is therefore susceptible to all limitations described previously.

Besides that, there is an architectural limitation that is worth discussing and should be addressed in the future. The framework avoids with all means any extra network calls in order to maintain low overhead incurred on energy. We base our bandwidth estimations solely on observing exchanged bytes along time while offloading the method. Because of this, a phenomenon may happen, altering our results and resulting in bad decision making.

This phenomenon happens when the first requests, while in discovery time, present high latency and return lower bandwidth than normally perceived. That may happen in case of network latency or if there is sudden lag on the server. Since our calculations rely on bandwidth estimation, the decision making will be greatly influenced by this low bandwidth value. If all subsequent calls to *ShouldOffload* return false, deciding on always executing code locally, the bandwidth will never be updated and all following calculations will continue using the wrong bandwidth value.

We can mitigate this by adding a certain probability to the decision, favouring the option outputted from our calculations with probability $P = 0.9$ as an example. We may only do this in case we sense that bandwidth was not updated for some time.

Another option is to simply update the bandwidth with an extra ping request, again according to some time threshold of staleness. Our implementation uses this technique, sending a dummy request to update bandwidth if it was not updated for more than two minutes.

4.5.5 Conclusion

The Offloading library is a key component of ULOOF, containing the methods and processes necessary for enabling Offloading, being shipped with the application bundle.

It consists of five modules, being *OffloadingManager*, *BandwidthManager*, *ExecutionManager*, *EnergyModule* and *DecisionEngine*. The library models the Offloading problem in utility functions, employing online location-aware equations. It uses indirect user location to adjust its latency estimations, behaving differently according to the interface used. Our proposal makes use of a black-box approach to predict method execution time.

The proposal is now concluded, with all modules defined. The implementation of the post-compiler, the server-side and the framework summed to around 9000 lines of code and were developed in Java.

We follow now on experimenting the framework as a whole to evaluate its performance and behaviour.

Chapter 5

Experiments and evaluation

This chapter compares a version of the unmodified application and a version using our framework. But first we justify the choice of not making a direct comparison with other existing frameworks.

5.1 Position with respect to related work

We would like to stress that this project is a user-level plug-and-play general use Offloading framework. It attempts to be the least intrusive to the application's ecosystem, not forcing any development constraints nor limits to the application's creator. It also does not require any special Android runtime or application data in advance, may that be its energy profile or method's debug information. This is a novel approach, trading the availability of extra profiling resources and better accuracy in predictions for an easy and realistic integration with the application. Despite its motto, this framework still achieves a transparent and considerable execution time and energy gains.

We have made no direct comparison between this framework and other works. A real comparison with other frameworks in the literature in the matters of energy and execution time is not obvious. The results are very dependent on application, remote platform and even bandwidth. One type of application can perform very well in execution time when using some framework policy, while still performing worse when considering energy consumption. The opposite may also happen, when one framework optimizes energy consumption at the expense of execution time. Some other frameworks will work very well under high-rate connections, but will fail otherwise because of lack of fault tolerance. Moreover, employing a fast/slow remote platform will change the results completely. Therefore one-on-one comparisons may become tendentious and senseless in some cases.

Besides stating that most works do not provide enough information for a complete implementation, they are incompatible for a fair comparison with our work. For CloneCloud and ThinkAir, a modified runtime is used. This means that the framework has access to stack traces, deep process information, method stack and variable values. The comparison with these frameworks would be unfair, as this work has no access to these parameters. For Cuckoo, no decision engine is currently available, and offloading happens whenever possible. This is the same as the "always offload" scenario, which is already covered in our experiments. For COSMOS and AIOLOS, they have no energy model, focusing only in time. Additionally, the application has to be completely modified to work in their systems. Finally, the focus of this work is the decision engine, not only the underlying framework. The models are extremely coupled with the parameters available, making the implementation of a new model very cumbersome.

5.2 Experimental Setup

In order to validate our framework, we have created two Android applications that are able to capture a reasonable variety of scenarios for testing. Additionally, we chose a pre-defined set of inputs that would also stress the framework in different ways, building a benchmark that we could test under different circumstances. This selected workload comprises of a mix of input parameters that result in executions of short ($duration < 400ms$), medium ($400ms < duration \leq 3000ms$) and long ($3000ms < duration$) duration.

5.2.1 Remote Platform

The used setup for the remote platform is described in Section 3.3: A local machine runs Android x86 in Virtualbox. A reverse tunnel with a cloud node enables outside access (e.g. from the internet) to it. On Android x86 we run our server application, which listens to the port 8080 for commands. The local machine is a 64-bit Ubuntu 14.04 linux, geared with an Intel i7-4500U processor with four cores at 1.80GHz. It has 8GB of memory.

5.2.2 Mobile Platform

The mobile device used for the tests was a Samsung Galaxy S5, which is powered by a Snapdragon 801 processor having a 2.5 GHz quad-core CPU and 2GB RAM. We can see that we are almost facing a technology convergence regarding computing power between the platforms. However, depending on context, the Snapdragon chipset can aggressively downscale the CPU speed and voltage to save battery, leading to energy savings at the cost of a reduced performance.

5.2.3 Android Applications

Two Android applications were created in order to evaluate the decision engine. We aimed at creating different classes of applications in order to assess the framework under various conditions. For each application, the framework was included in the project and one method was annotated as Offloading candidate. To evaluate the framework, we create specific test batches per application, which are detailed bellow individually.

5.2.3.1 City route

This is a proof-of-concept application which simulates a navigation application. The application finds the best route between two points. A city is represented by an unweighted graph, where vertices are crossroads and edges are streets. The graph was taken from SNAP [SNAP Pennsylvania, 2016], the Standford Large Network Dataset Collection. We trimmed some nodes and added the node distances between a chosen starting point and the end points. The distance information is used by our custom assessment class, which outputs an assessment between two given points. We used the normal process described in Section 4.3 to set a custom assessment class for the method. After editing the graph, the maximum distance between two points is 139 hops away and the file is 2.1MB. The graph has 120000 edges.

The Offloading candidate method receives two points and tries to find the best route between these. This is done with a breath-first search algorithm. The method returns the number of hops between the points. The graph is assumed to be available both at the device and remote platform. A predefined set of end nodes are chosen, varying execution runtime in such way that always offloading or never offloading will not be the optimal solution. The end node choice was random, representing [1.1%, 1.2%, 1.3%, 1.4%, 1.5%, 1.6%, 1.7%, 1.8%, 1.9%, 1%, 2%, 3%, 4%, 5%, 10%, 15%, 20%, 25%, 30%, 33%, 35%, 45%, 50%, 60%, 70%, 80%, 90%, 100%] of the graph's diameter. The absolute number of hops of these values taking the best route are: [18, 19, 20, 21, 22, 23, 28, 33, 37, 51, 62, 70, 77, 83, 87, 89, 100, 105, 112, 117, 119, 120, 121, 126, 131, 132, 133, 134, 136, 138, 139]. Starting from 105 hops, the volume of nodes is high, taking up to 5 seconds to find the optimal route in our test mobile device. The asymptotic complexity of the algorithm is classified as $O(V+E)$, where V is the number of nodes (crossroads), E is the number of edges (streets).

One round consists of calculating the distance of each end node in the set. The test batch executes one round twenty times.

The assessment method of the Offloading candidate outputs the smaller distance between the two points given. This information is already available in the graph.

5.2.3.2 Fibonacci

An application for calculating the Fibonacci numbers is created for the purpose of representing exponential execution time. The implementation is a naive version of the algorithm without *memoization*. The Fibonacci numbers to be computed are chosen beforehand to provide a mixed workload and stress our decision engine. The set of numbers $N := 5, 10, 18, 20, 22, 25, 30, 35, 40, 41$ is used as test batch. Notice that most numbers can be computed in less than a second, while starting from 30 it will take around 35 seconds to be computed. The asymptotic complexity of the algorithm is classified as $O(2^n)$.

The Offloading candidate method receives a number n and calculates the Fibonacci number using the process described above. One test round consists of calculating each number N_i once. The test batch executes one round fifteen times.

The assessment method of the Offloading candidate simply outputs the Fibonacci number.

5.3 Sensitivity test for α

As mentioned in Section 3, the constant α defines the priority between favouring energy consumption or execution time in the Offloading decision. This section analyzes the effect of α depending on the application.

5.3.1 Methodology

A sensitivity analysis was performed to evaluate alpha. It consists of analysing α according to three scenarios: (R) always remote execution, (L) always local execution and (D) decision engine. In (D), our decision engine is used with varying values of α. To acquire the data, experiments are made with some of the applications developed in a semi-controlled environment. We created a semi-controlled environment by setting both mobile and remote platform to lie in the same network.

This batch of tests aims at simulating the execution of an application, summing the execution time and energy consumption in all three scenarios. Smaller values are better, meaning it has used less resources. Scenario (D) is the only one affected by α, since its decision depends directly on it. $\alpha = 0$ will prefer saving energy, $\alpha = 1$ will conversely save time.

5.3.2 Evaluating α

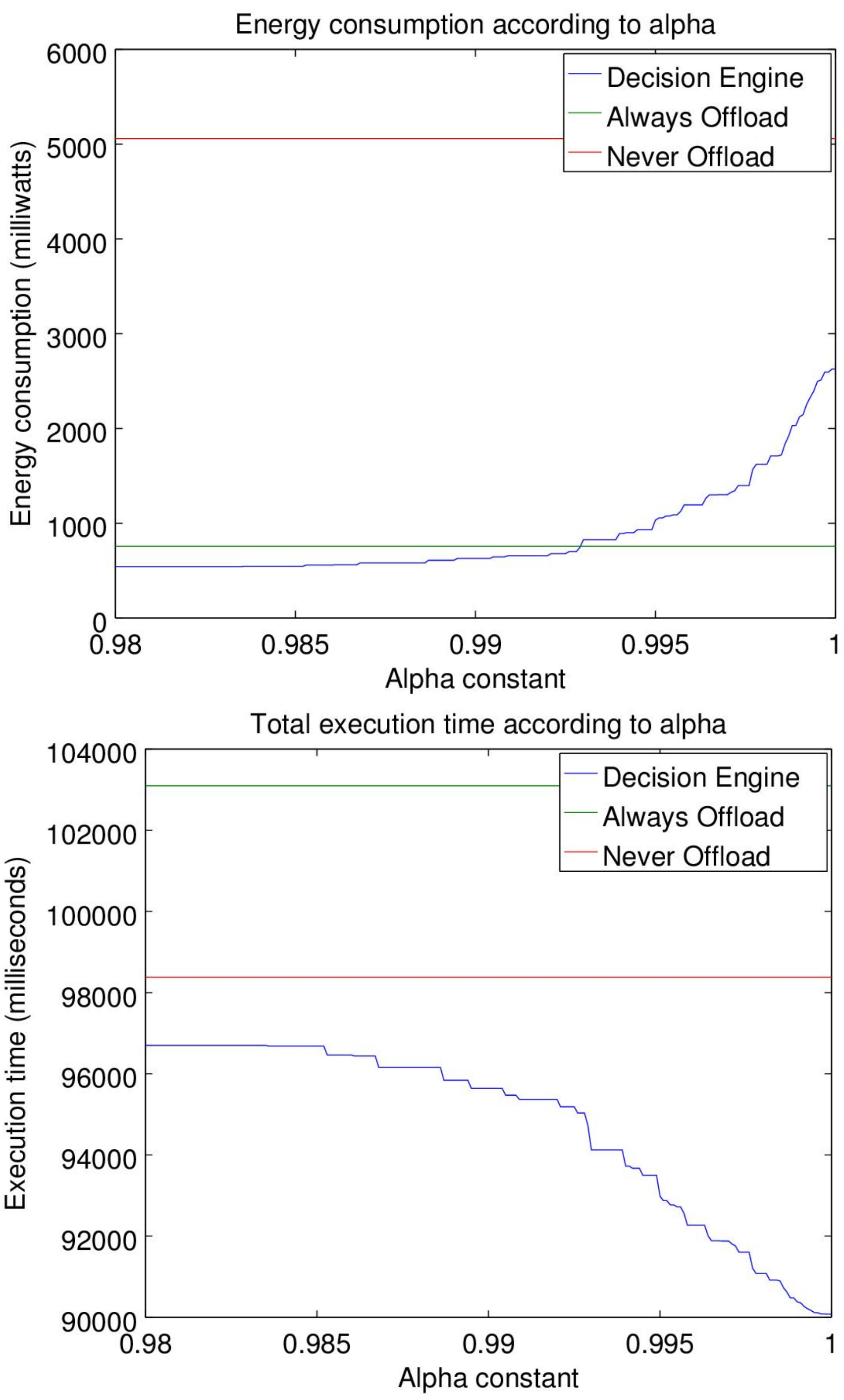

Figure 5.1. Total energy and time in a batch of tests for City route application 5.2.3.1 with low latency link

Figure 5.1 shows the results for the city route application performed in a low latency scenario, where the remote platform was in the same network as the mobile. All plots start from a given range, in this case 0.98. Before that (D) will not change its decision because of the unnormalized values for time and energy.

As one can see, time-wise (D) performs better than both scenarios with any α value. One may imagine that curve (D) should always lie between (R) and (L), but this will depend on the application's workload, the remote platform's performance, and bandwidth. The chosen test batches mix executions with different runtime characteristics, rendering the strict decision of (R) and (L) a poor choice. Low runtime executions are faster and inexpensive to be computed locally, while long lasting executions should be run remotely.

When setting $\alpha = 1$ we have lower execution time, but spend more energy. The sweet spot, where we would get the lowest execution time and still perform energetically as good as scenario (R) is where both lines from scenario (R) and (D) cross in the energy plot: $\alpha = 0.993$. The same test was made in a scenario with latency, but this showed similar results.

Another similar test in our semi-controlled environment was performed using the Fibonacci application. Figure 5.2 shows total resource sum after the complete test. The graph shows how the curve is now changed in comparison to the city route application. Now the value of α does make a big difference in the choice, deciding whether the system performs better than both scenarios or not. The sweet spot seems to lie in $\alpha = 0.995$, where both time and energy would be lower than both scenarios.

By changing the algorithm class, we see a slight difference in the plots. In this case, the burden of always contacting the remote platform, transferring the arguments, executing the code remotely and returning the result is not worth time-wise. The overhead involved makes it a poor choice in some cases. This is why the decision engine plays an important role. In this particular choice, the mobile device performs as good as the remote platform. This is possibly due to technology convergence and optimizations made by the Android system in mathematical calculations.

The last sensitivity test was performed using a modified version of the Fibonacci application. Its method was modified to receive additionally to the Fibonacci number an independent byte array. This creates a big size argument which has to be transferred through the network. The method ignores this byte array in its calculations, created only to increase the arguments' transfer size. Figure 5.3 reflects how this changes the performance of the framework. The burden of having to transfer the argument impacts greatly the time plot, performing time-wise worse than scenario (L) when favouring energy. This happens because of the network lag involved in transferring the arguments changing only when shifting α towards 1. The constant α is now crucial to the decision and should be adjusted according to user's wishes: energy or time. The sweet spot is now not obvious. A better analysis should be performed to find which α value would deliver the highest gain for both attributes.

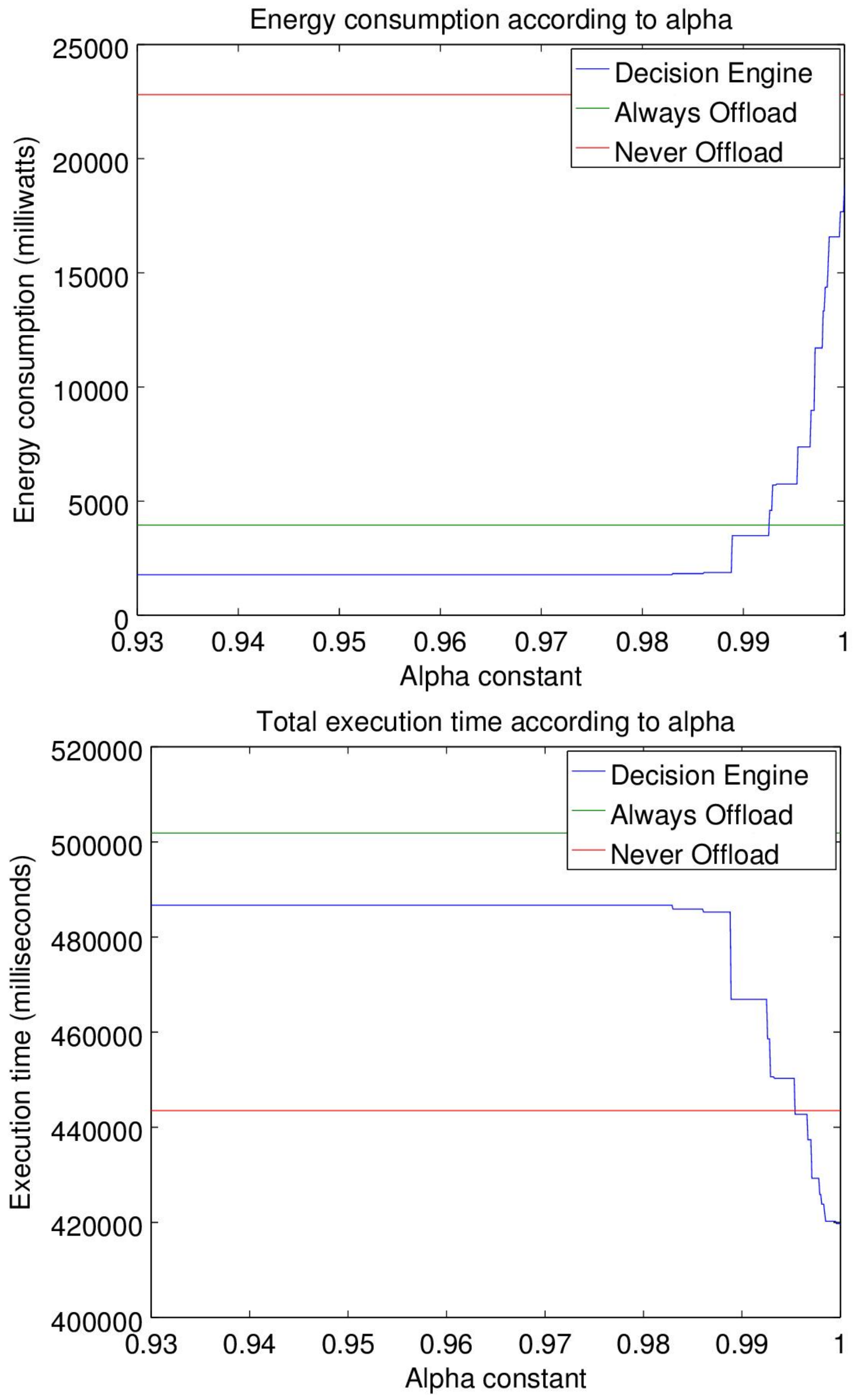

Figure 5.2. Total energy and time in a batch of tests for Fibonacci application 5.2.3.2

Although one can not generally tell which α value is best suited for every situation, we can predict how it will behave according to the scenario. Based on the observations of the results, we believe that the choice of alpha will depend on the performance of the remote platform in comparison to the mobile device, and as well on the bandwidth available.

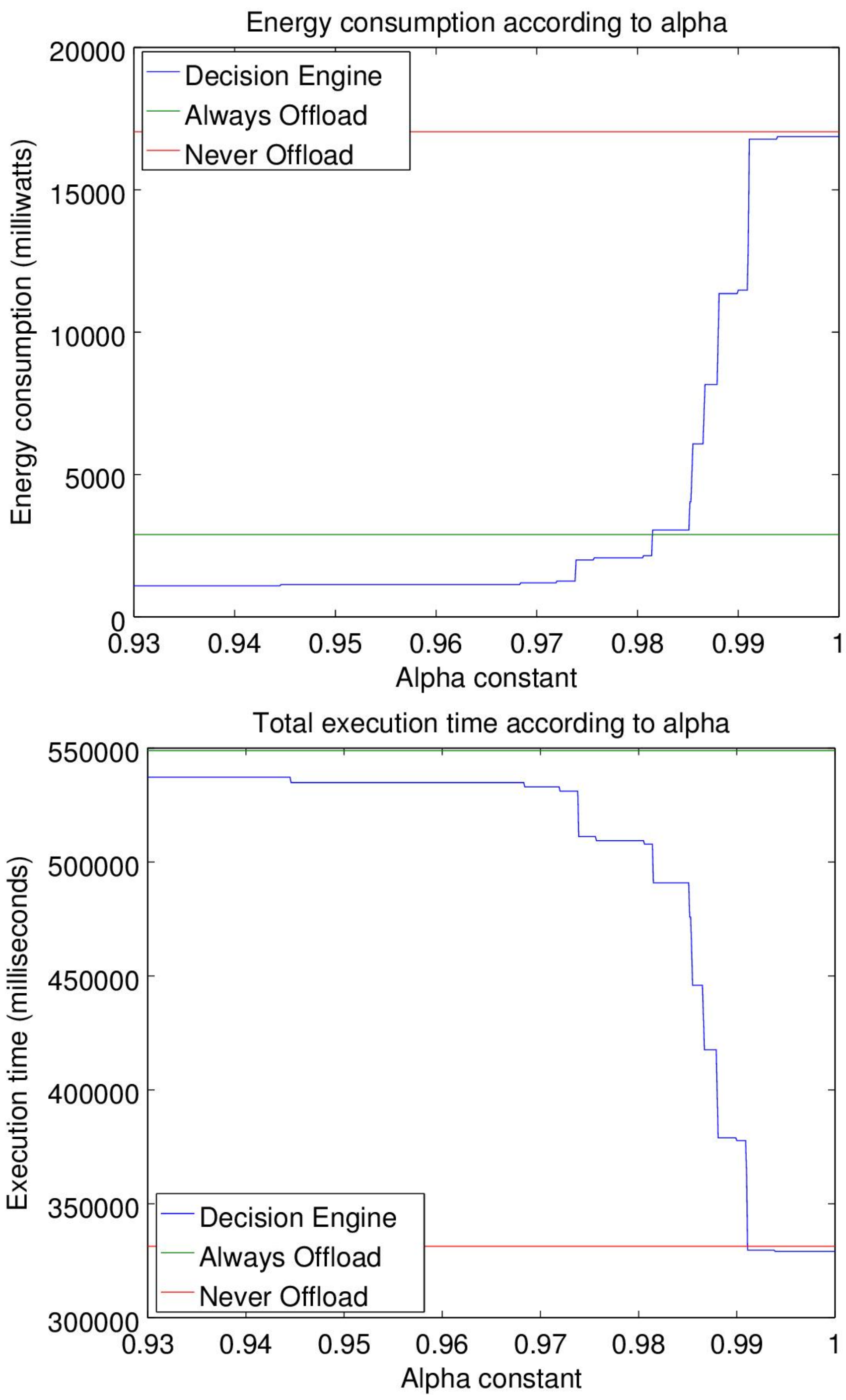

Figure 5.3. Total energy and time in a batch of tests for modified Fibonacci application with big size arguments

More precisely, two characteristics affect directly the curves of (R) (L) and (D): the network bandwidth and remote execution time. Network latency, arguments and result sizes also affect the curves, but they can be included in the bandwidth as they also change transfer time. Local execution time can be conversely covered by changing the remote execution time. Thus, with these characteristics, four scenarios are possible, which should generate different results:

- **Faster remote platform, high bandwidth:** Having a faster remote platform and high bandwidth makes offloading a very good choice. Therefore (L) curve will always be above (R) in both energy and time. We have $(D) \leq (R)$ at all times, independent of α.
- **Faster remote platform, low bandwidth:** The outcome will depend greatly on the mobile execution time. If a great part of the method execution times are smaller than the time taken to contact the remote platform, (R) is expected to stay above (L) in the time plot. Otherwise it will be on the contrary $(R) < (L)$. (D) in this case is expected to start between (R) and (L), converging to $min((R), (L))$ as it is tending to 1. The energy curves will again behave according to the executions, normally having $(R) < (L)$. (D) starts bellow $min((R), (L))$ finishing in the middle.
- **Same execution time, high bandwidth:** The execution time curves will be $(R) > (L)$, starting with (D) in between with $\alpha = 0$. As we shift α to 1, (D) will tend to (L). In this case, because of the Offloading overhead, we have low probability of performing better than (L). That means that when $\alpha = 1$, $(D) = \theta((L))$. For energy, $(R) < (L)$ and $(D) \leq (R)$ when $\alpha = 0$. (D) will end between (R) and (L) as α goes to 1.
- **Same execution time, low bandwidth:** For execution time, we have a similar setup from the previous scenario. The gap between (R) and (L) is bigger because of the transfer time. (D) this turn will start between (R) and (L), and will be converging to (L) when $\alpha = 1$. With energy we also have again a similar scenario with $(R) < (L)$ depending on the executions. We will have $(D) = \theta(min((R), (L)))$ when $\alpha = 0$, but it will be $(D) = \theta(max(((R), (L)))$ when $\alpha = 1$. As we can see the choices of α are sharper in this scenario. (D) will most probably not perform better than (R) and (L) at the same time in both plots.

5.4 Accuracy of prediction of execution time

In order to assess how accurate our execution time prediction is, we have executed the annotated Offloading candidate from the City route application with its workload. The tests were made again in a slightly modified semi-controlled environment, where the mobile had WiFi connectivity at all times and the remote platform was on the internet. We call it semi-controlled, because the mobile is still subject of various changes and interferences such as background jobs, network latency and CPU frequency changes. Because the backend was located in the cloud it also faced latency. We stored the prediction of each execution time (local and remote) and the method was executed both locally and remotely, returning their real execution time right after.

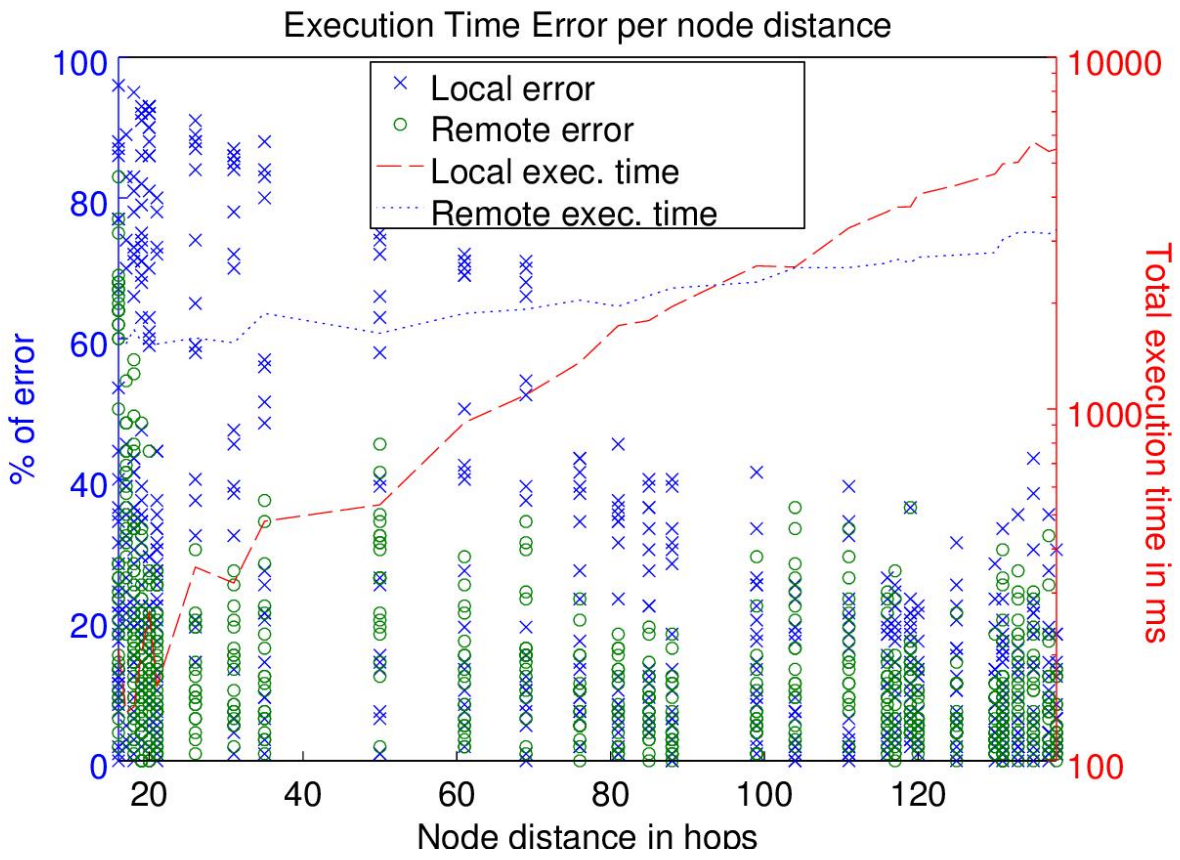

Figure 5.4. Scatter plot of relative prediction error together with real average execution time per hop

In Figure 5.4 we may find the relative error of the estimation in comparison to the real value of the method's execution time. In the plot, the points represent the relative prediction error (left Y-axis) and the curves represent the total execution time (right Y-axis). We can see that with small execution time, the relative error does float across the Y axis, reaching big figures sometimes. As execution time grows, the relative error is low with more precise predictions. Our application shares the CPU with other processes, meaning that our black-box approach may be less accurate when the execution time is smaller than 1 second. It leaves limited legroom for eventual lags and variations of historical executions. But this intrinsic prediction problem is less relevant when we look at the real error in milliseconds. Some milliseconds of lag may still pass unperceived by humans.

One may also notice that the spread of the error function tends to zero as execution time rises. Local prediction errors also had much higher variance, with a standard deviation starting from 137.34% when the local execution time is 207 milliseconds and reaching as low as 9.85% when the average runtime is 4658 milliseconds. On the other hand we have the standard deviation of the remote platform prediction error, which starts with 26.18% for 1658 milliseconds of execution time and drops down to 4.51% with an averaged runtime of 3028 milliseconds. The remote platform seems to have a steadier runtime, explained by being a dedicated server running in Android x86. The predictions are also much more consistent on the remote platform when looking at fast executions. In fact, they also become more precise as more instructions are being computed and the variation incurred by the system becomes less representative.

This explains once more the instability perceived on the Android platform for executions that are fast ($< 300ms$) and how our black-box approach works better starting from methods that run for longer than some milliseconds. Starting from 1000 milliseconds the error is relatively small, figuring 14.88% in average for the local executions.

Looking from an execution time point of view, we see a clear region being formed, representing when the application will profit from offloading. The two lines of execution time meet when they are both around 1150 milliseconds. Starting from this point we can mark this region as a time-wise interesting situation for offloading. It means that the offset incurred by latency and serialization plus the remote execution time are smaller or equal than the local execution time. As we can see, the local execution time grows exponentially, while the remote platform manages well the curve steepness, trending a linear growth.

Table 5.1. Average time of real executions and their respective predictions clustered by group

G	**Local (ms)**	tm_l (ms)	**Rem. (ms)**	tm_r (ms)
1	252.36	$159.51\ (36,79\%)$	1601.03	$1375.43\ (14.09\%)$
2	1472.09	$1390.97\ (5,51\%)$	2015.68	$2006.51\ (0.45\%)$
3	4286.49	$4257.82\ (0.66\%)$	2817.21	$2668.28\ (5.28\%)$

Table 5.2. Standard deviation of averages by group

G	σ **Local**	$\sigma\ tm_l$	σ **Rem.**	$\sigma\ tm_r$
1	281.90	224.34	304.76	380.27
2	564.49	519.37	228.14	189.85
3	1185.60	1184.43	390.14	339.46

Trying to identify the regions where the Offloading framework will work better, we have divided the executions in three groups: G_1, G_2 and G_3, where $G_1 := \{e \mid e.hops \leq 50\}$,

$G_2 := \{e \mid 50 < e.hops \leq 90\}$ and $G_3 := \{e \mid 90 < e.hops\}$. In Table 5.1 one may find the average local and remote real execution times and the average of tm_l, tm_r, the functions of execution time prediction by group. In Table 5.2 one may find their respective standard deviation σ. In G_1 we have small execution time, evidencing higher prediction error as discussed before. The tm_r prediction does a better job in this group, due to the higher stability of Android x86. In this group, we recognize that all executions should be local, as we have high overhead in comparison with execution time. One may notice high oscillation in this group by looking the considerable values at Table 5.2. In G_2 we perceive a more accurate prediction in both platforms, having less representative oscillations between execution times. The gap between both functions is very similar (around 150 milliseconds). We have in this group mixed Offloading decisions, depending on the execution time and energy consumption. The initial time offset incurred by the overhead is now not so relevant, and some executions will be performed remotely. Finally, in G_3 the remote execution outruns the local value, maintaining a low average error. We may notice that the error on the remote platform has slightly risen, but this can depend on executions and does not strictly represent a trend. The executions in group $G3$ are in this case the representative elements, where offloading is the optimal choice.

It is worth noting that the analysis conducted in this section is easily generalizable to whatever application, where the boundaries between execution time regimes G_1, G_2 and G_3 can differ as a function of the application.

5.5 Location-awareness

Previous experiments used some sort of semi-controlled environment. In this section, we perform a free experiment to fully test the location-awareness and testify the behaviour of ULOOF.

The main goal of this section's experiments is to test whether the decision engine works under real life circumstances stressing our location-aware model. The experiments assess the differences between an application with and without the Offloading framework. A version of the applications including the framework and the decision engine of this project were used.

We have tested the framework with two applications, each with a slight different scenario, specified in their respective section.

There are many setups in which we can experiment, configuring the alpha in many ways or shutting down the decision engine, always offloading the candidate. We choose three main scenarios to draft our results: setting α to 0, setting α to 1 and the unmodified version of the application. This can capture the engine when optimizing execution time or energy consumption in comparison to the unmodified application. The sweet spot for α mentioned in Section 5.3 varies according to latency, method argument and remote execution time. Therefore, one would have to create an algorithm to analyse the outcome of the equations and adjust α at each round. This was left for future work, and we chose to set it at both extremes and experiment each part separately.

5.5.1 City route application

An experiment with the city route application was made, comparing the unmodified version and the version with the framework. The version with the framework came in two flavours: One configured with $\alpha = 0$, favouring energy and another one configured with $\alpha = 1$ favouring time. To test the framework's capacity to work under bandwidth variations, a remote platform setup described in Section 3.3 was used. The test scenario starts with a WiFi connection, continuing with a drive around the city of Belo Horizonte in a predefined route. The drive takes around 25-30 minutes to be completed spanning a part of the city and finishing where it started. This was made in order to assess the behaviour of the system under stress of high latency connections as well as variations in bandwidth along the way.

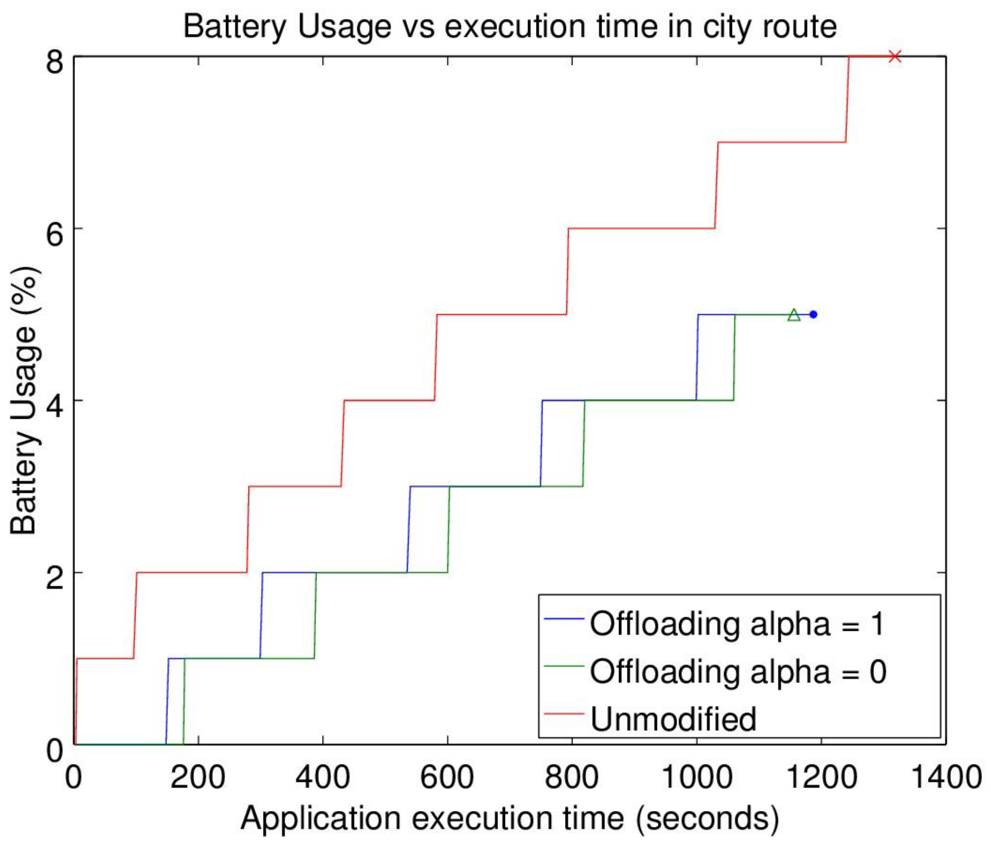

Figure 5.5. Battery usage comparison of city route application in a high latency scenario

As shown in Figures 5.5 and 5.6, both versions with the framework perform in energy and execution time better than the unmodified version. The curves have different length on the x-axis, purposely finishing before the end of the graph. In this case, it means that the application has finished at that point, having no more data to be shown. Hence, this represents a lower execution time. A dot, a cross and a triangle represent the end of the curve for $\alpha = 1$, $\alpha = 0$ and unmodified respectively.

As analysed in Figure 5.1 before this real experiment, our energy and time footprint should be smaller than the unmodified version. The results depicted here confirm our findings.

The version with $\alpha = 0$ does spend less energy than the other versions as expected. As a curious fact, $\alpha = 1$ took overall thirty seconds longer than $\alpha = 0$, which would contradict this work if this was a completely controlled environment. But they were made in different test batches, suffering from variations in the network, CPU frequency and even maintenance background services in the mobile, which can alter the method runtime. Moreover, thirty seconds represent only 2.6% of the test runtime. The results in Section 5.1, which were made in a more controlled environment, also predict the improvement of only 6 seconds in overall runtime in case of choosing $\alpha = 1$.

In this experiment, the remote platform was not a very good option all the times, being faster than the mobile platform but expensive due to its high latency. That means that only long lasting executions would be eligible to offload. The decision engine offloaded up to 27.6% of executions when $\alpha = 0$ and 29.13% when $\alpha = 1$. This can explain the little difference shown between the two options in Figure 5.1. The average runtime of the methods offloaded is around 2.5 seconds, which is quite above the total average of 1.5 seconds.

The results shown in this experiment fall within the expected values and correspond to the simulated part of our work.

Since the time and energy improvements of the framework were not significant in the experiment with an uncontrolled environment, we decided to rerun the experiment on a network with more predictable latency. Thus, the mobile was connected to a WiFi network at all times, and the server was located in the same local network as the mobile. The results, shown in Figure 5.6, prove our initial claim about latency. With a stable low-latency connection we can now see an improvement of 80% in battery and 47.64% in execution time. This difference can be attributed to latency: in this semi-controlled environment we offloaded 62.41% and 72.95% for $\alpha = 0$ and $\alpha = 1$ respectively. This is is around 100% more in both cases in comparison to the previous uncontrolled environment.

One of the causes of the poor improvements is the significant variations on the network. We can see bandwidth variation according to the towers connected to the cellphone during the experiment in Figure 5.7. As expected, upload and download bandwidth present difference,

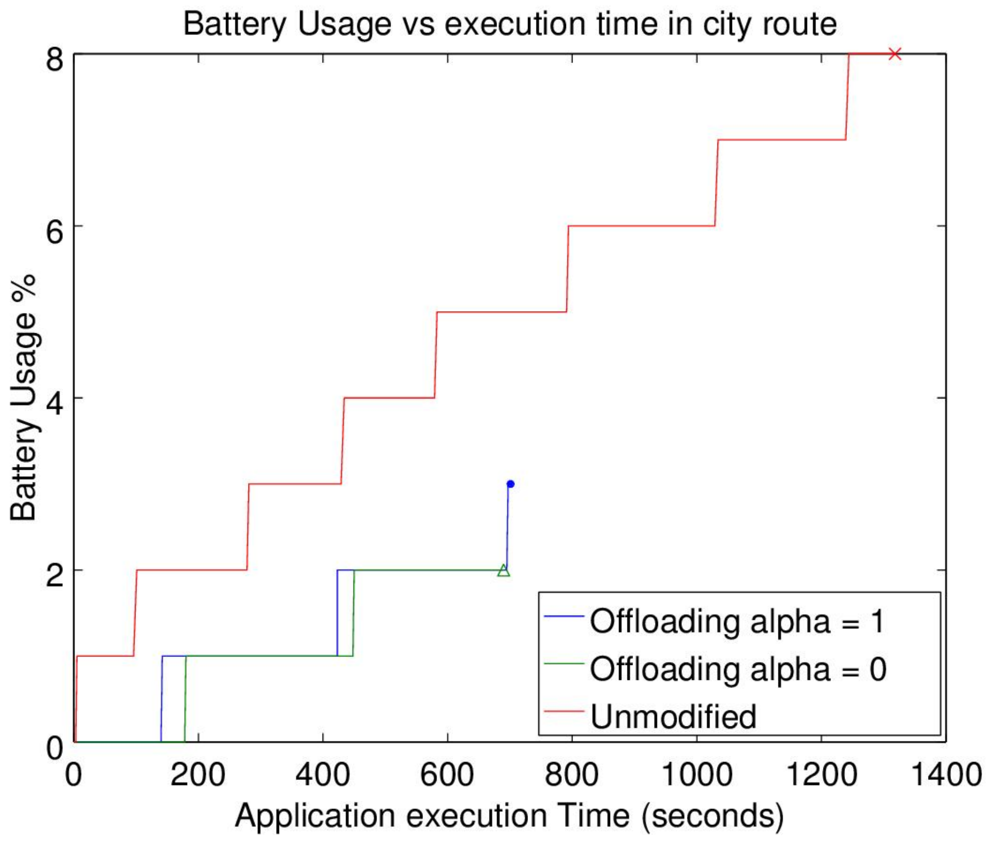

Figure 5.6. Battery usage comparison of city route application in a semi-controlled environment on Wifi

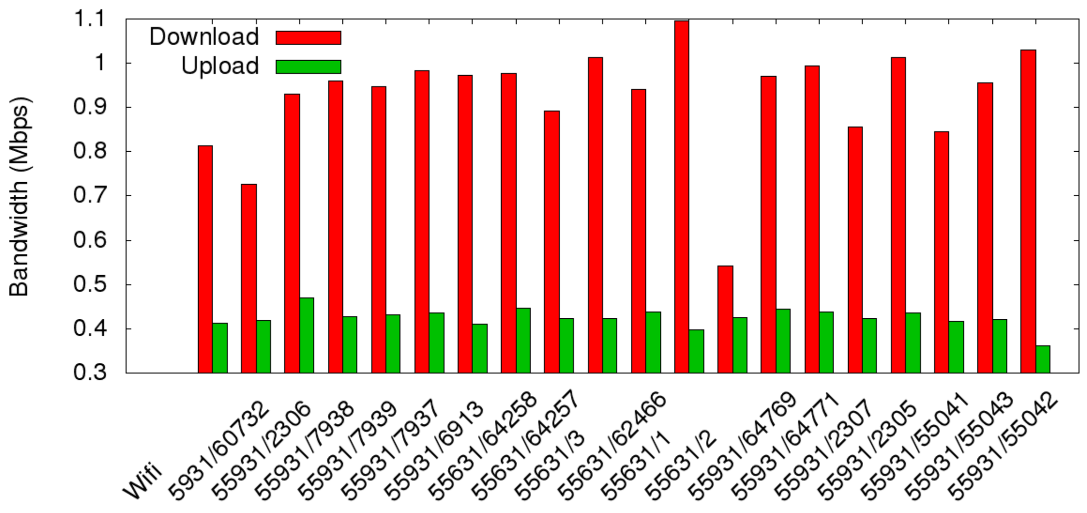

Figure 5.7. Bandwidth by cellphone tower and Wifi in city route application

accounting of an average of 47.10% of delta. This is accounted for in the system and treated accordingly in its calculations. One may find the LAC/CID from the towers on the x-axis.

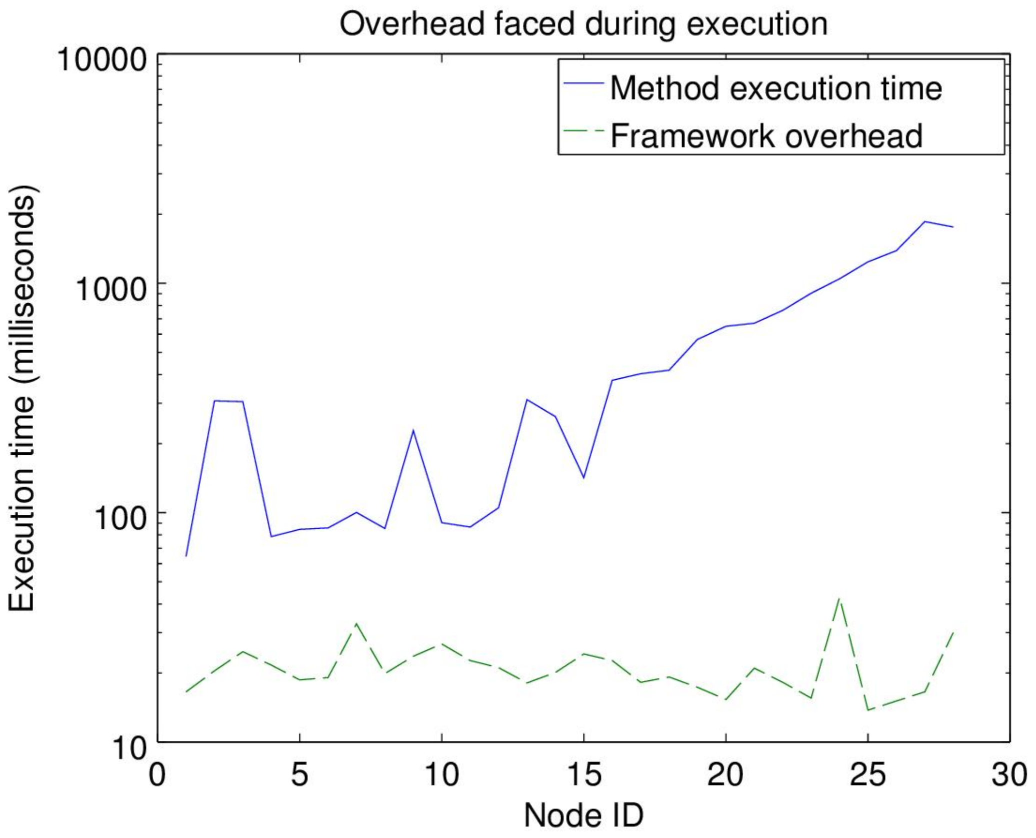

Figure 5.8. Overhead caused by the offloading framework

The overhead caused by the framework is shown in Figure 5.8. It is kept under 70 milliseconds at all times. Although this overhead may be significant for methods that run for less than hundreds of milliseconds, it is worth noticing that the average execution time of the methods is 513.47 milliseconds and on 60% of executions from this workload the overhead was maintained under 10%.

5.5.2 Fibonacci numbers

A similar experiment was made by using the application that generates the Fibonacci Numbers. We have used the same route and the same characteristics from the previous section, with a slight modification. In this setup, the experiment was made while the remote platform had a considerable higher latency, due to the changing of the WiFi broadband provider from the server. Curiously our results indicate that the plain execution time of the remote platform [1] was higher than the mobile. This may happen due to JIT optimizations performed by the relying Android system or even the use of GPU to perform mathematical calculations. The

[1]Without accounting for latency

same happened in the sensitivity test presented in the previous section, although one can not notice this solely by analysing the graph showed, since the delta between both platforms may be consisted of several attributes like workload and bandwidth. Another remarkable attribute is the low battery usage in comparison to the previous experiment, although similar overall execution time was necessary to perform the task.

Figure 5.9 shows the difference between the two scenarios, changing decisions according to α. The two experiments were made again in different test batches, offloading 15.89% of the executions when α was set to save energy, and 1.32% when α is set to 1, favouring time.

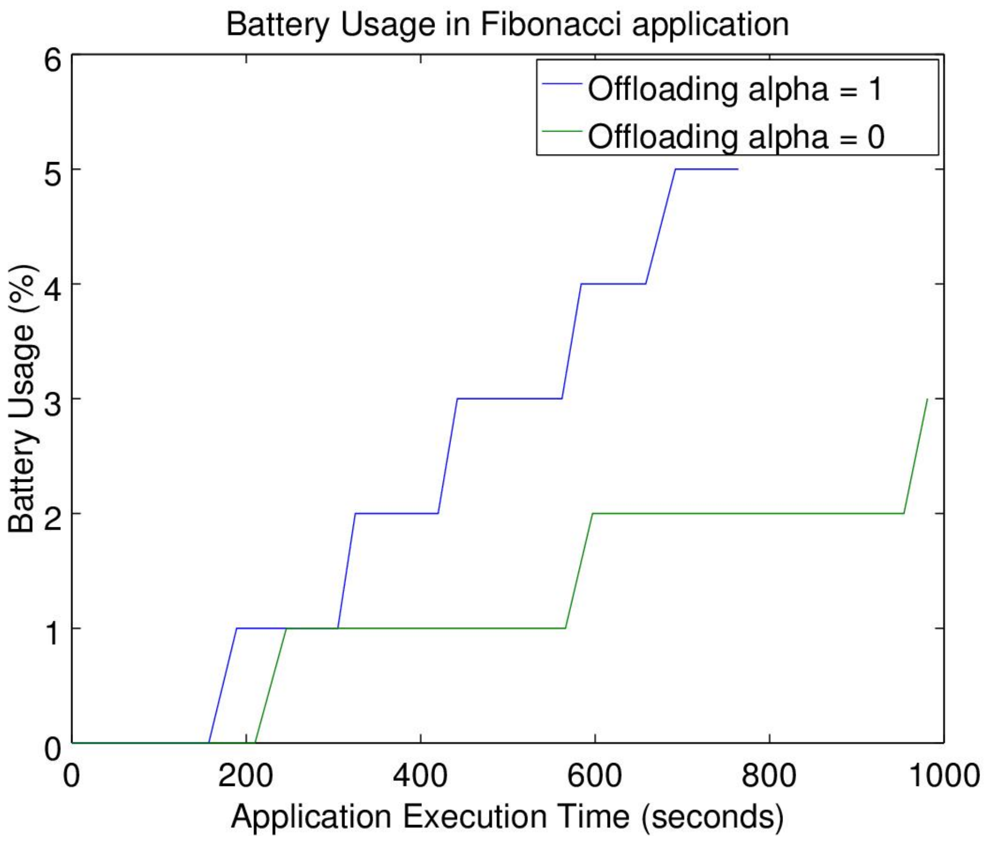

Figure 5.9. Battery usage comparison of Fibonacci application

Having the remote platform performing worse than the local device, and using the application within a real world scenario with high-latency and varying bandwidth will limit the amount of resources that offloading can save. That means that the utopian situation, where the environment presents low latency links and extremely fast remote platforms may not always exist. Our decision engine will nevertheless still do its best to profit from offloading.

In this case there exists a trade-off and definite choice between energy and time. The version with $\alpha = 1$ is quite close to the scenario "never offload", therefore we chose not to plot it in the graph.

5.6 Conclusion

We have concluded the chapter with experiments and evaluations from the proposed framework. The chapter included the development of two applications: city route and the Fibonacci, to evaluate our framework. An appropriate workload for each application was chosen to test the adaptive ability of the work.

We conducted a sensitivity analysis, based on altering the α scaling constant and observing the behaviour according to three scenarios: The Decision Engine, Always Offload and Never Offload. It was identified that two main characteristics affect directly the curves: bandwidth and remote execution time

Later on, the accuracy of our execution time predictions were assessed by running our decision engine and comparing the estimations with the real values measured. We noticed that the framework showed more stability and accuracy in its predictions starting from method executions that last around 1000 milliseconds. Three groups were formed from this analysis G_1, G_2 and G_3, each representing increasing probability of offloading.

Our framework was tested freely in a dynamic scenario with mobility, having three versions of the same application as comparison: An unmodified version, and two versions with the framework, where $\alpha = 0$ and $\alpha = 1$. For the city route, we faced an overall improvement of 37.5% of battery consumption and around 15% of execution time improvement. In a semi-controlled scenario for the same application, we have around 80% less battery consumption comparing the version with $\alpha = 0$ and the unmodified application, and around 47% improvement of execution time.

We observed an overhead of maximum 70 milliseconds for each execution in the city route application. The average runtime was 513.47 milliseconds, totaling 60% of executions with less than 10% of overhead for the workload described.

In conclusion, we attested that our framework can improve both execution time and energy consumption in a highly dynamic scenario, incurring low overhead and offering the developer easy and realistic integration in applications.

Chapter 6

Conclusion

In this work we presented ULOOF, a novel approach for a transparent cloud-assisted Offloading framework. We first analysed related works, recognizing shortcomings and limitations. We conceptualized a proposal that is easy to be adopted and integrated in existing applications; incurs little relative overhead on energy consumption and execution time; provides precise estimations in its models and adapts itself quickly in regards to user mobility, having deep in its concept the support for latency fluctuation.

We contributed to the community with a remote cloud platform that is able to reproduce the mobile device environment and execute code remotely. The remote platform also introduces a minimal protocol for information exchange with the client. We developed a post-compiler that is able to manipulate bytecode and insert arbitrary entry-points in order to intercept method calls and direct them to our framework. The result of the post-compiler is a modified Android application, enabled for Offloading and ready to be installed.

The implementation also consists of an Offloading library that is embedded in the application bundle, aiming for method instrumentation, dynamic evaluation of the Offloading decision and communication with the remote platform. The library provides numerous methods, enabling us to wrap the Offloading candidates with a proxy method, which calls our decision engine and maintains the necessary framework lifecycle. As a relevant strong point, the library incurs low impact at the application, spawning no background processes for the monitoring and decision-making method, thus yielding low footprint and overhead. Our models are online and self-adapting, being created from scratch to be location-aware and arrange latency according to user location. We created an energy model, derived from real device data that can successfully capture the method's energy utilization. We identified a trade-off between energy consumption and execution time, which are two main attributes when optimizing a method. Our models incorporate this trade-off through a scaling constant α, which merges two equations, one for each attribute. The framework works in a plug-and-play fashion and is easy to be imported and used by the application.

We experimentally evaluated our framework through two applications: The city route, for calculating a route between two points in a city; and an application to calculate the Fibonacci numbers. We diagnosed several scenarios in which our scaling constant α can be fitted and analysed the outcome under various conditions. We discovered the existence of a "sweet spot": the α setup that maximizes resource gains. We analysed the accuracy of our models by evaluating the relative error among several executions. Finally we run the applications under full mobility condition in three scenarios: $\alpha = 0$, $\alpha = 1$ and unmodified application. In comparison to the unmodified version we achieve 40% energy consumption improvement and around 15% of execution time improvement for the city route application. We repeat the test with a stable network connection and attest an improvement of 80% and 47.64% for energy consumption and execution time respectively.

The framework was tested and evaluated by two applications that were developed by us. We recognize the need of testing the framework in other scenarios and with other applications, preferably developed by a third-party. We do not claim to have exhausted all situations and we plan on improving this analysis. Other measurements like intrusiveness are judged subjective and may be assessed differently according to other points of view. To be able to fully attest this, the framework may need to be tested by several developers in a real world scenario.

6.1 Future work

For future work, we have several improvements in mind.

- We plan to develop a dynamic algorithm for adjusting α at each iteration, converging at the sweet spot mentioned earlier to maximize energy and execution time gains.

- We will make the bandwidth estimation more realistic for LTE: We are aware that although the Shannon equation may give us accurate bandwidth predictions, the LTE bandwidth curve resembles a staircase [Scheme, 2009] having its data rate changing according to modulation and channel frequency. One challenge is to acquire the current modulation used while still keeping the framework on user-level.

- Currently the framework does not transfer back the environment from the remote platform, meaning that changes made in the remote platform's environment will not be replicated back. This is supported by our model but was not implemented due to time constraints. We will finish the implementation in the future.

- Isolation for executions on the remote platform is still an open challenge. Because we replicate the mobile state and do not employ any sandboxing [1], parallel executions may affect each other. We plan to solve this issue.
- Java supports callbacks by allowing interfaces to be passed as arguments. Our framework does not support this, remaining as a challenge and an open issue.
- The Offloading candidates are annotated manually by the developer. This may not be strictly done this way and can be automated. We aim at developing a tool to identify these candidates without developer interference, enabling our framework to work on packaged existing applications, broadening its usage.
- We did not attempt to employ any security or authentication on the communication with the server. For future work we will create an authentication protocol, together with the needed security measures to make data exchange untamperable.
- We would like to optimize object transfer size for state replication. In the current implementation the complete *this* object, together with all static fields are being transferred. This can be selectively lowered with the help of static analysis.
- The *BandwidthManager* and the *ExecutionManager* accumulate data to profit from past executions. Our current implementation does not foresee any eviction policy so far. It means that the data may grow too big for the device itself. An optimal eviction policy must be employed in these modules to avoid reaching this point.

[1] sandbox is a technique and a security mechanism for separating running programs

Bibliography

Akima, H. (1970). A new method of interpolation and smooth curve fitting based on local procedures. *J. ACM*, 17(4):589--602. ISSN 0004-5411.

Android AIDL (2016). Android aidl. `http://developer.android.com/guide/components/aidl.html`. Accessed: 2016-04-01.

Android AVD (2015). Android avd manager. `http://developer.android.com/tools/help/avd-manager.html`. Accessed: 2015-06-28.

Android OS (2016). Android operating system. `https://www.android.com`. Accessed: 2016-02-03.

Apache HttpComponents (2016). Apache httpcomponents. `https://hc.apache.org/`. Accessed: 2016-02-03.

Apple iOS (2016). Apple ios. `http://www.apple.com/ios/`. Accessed: 2016-02-03.

Autossh (2016). Autossh. `http://www.harding.motd.ca/autossh/`. Accessed: 2016-02-03.

Baer, J.-L. and Schwab, B. (1977). A comparison of tree-balancing algorithms. *Communications of the ACM*, 20(5):322--330.

Balan, R. K., Satyanarayanan, M., Park, S. Y., and Okoshi, T. (2003). Tactics-based remote execution for mobile computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 273--286, New York, NY, USA. ACM.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164--177. ISSN 0163-5980.

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41--41, Berkeley, CA, USA. USENIX Association.

Carroll, A. and Heiser, G. (2010). An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, volume 14. Boston, MA.

Chen, D., Messer, A., Milojicic, D., and Dwarkadas, S. (2003). Garbage collector assisted memory offloading for memory-constrained devices. In *Mobile Computing Systems and Applications, 2003. Proceedings. Fifth IEEE Workshop on*, pages 54--63. IEEE.

Chen, E. and Itoh, M. (2010). Virtual smartphone over ip. In *World of Wireless Mobile and Multimedia Networks (WoWMoM), 2010 IEEE International Symposium on a*, pages 1–6.

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. (2011). Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301--314, New York, NY, USA. ACM.

Cidon, A., London, T. M., Katti, S., Kozyrakis, C., and Rosenblum, M. (2011). Mars: Adaptive remote execution for multi-threaded mobile devices. In *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, MobiHeld '11, pages 1:1--1:6, New York, NY, USA. ACM.

Cignetti, T. L., Komarov, K., and Ellis, C. S. (2000). Energy estimation tools for the palm. In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '00, pages 96--103, New York, NY, USA. ACM.

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010). Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49--62, New York, NY, USA. ACM.

dalvik (2015). DalvikVM. `https://github.com/android/platform_dalvik`. Accessed: 2015-05-20.

Esteves, R., McCool, M., and Lemieux, C. (2011). Real options for mobile communication management. In *GLOBECOM Workshops (GC Wkshps), 2011 IEEE*, pages 1241–1246.

Genymotion (2015). Genymotion. `https://www.genymotion.com/`. Accessed: 2016-01-28.

Ha, K., Pillai, P., Richter, W., Abe, Y., and Satyanarayanan, M. (2013). Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 153--166, New York, NY, USA. ACM.

Ickin, S., Wac, K., Fiedler, M., Janowski, L., Hong, J.-H., and Dey, A. (2012). Factors influencing quality of experience of commonly used mobile applications. *Communications Magazine, IEEE*, 50(4):48–56. ISSN 0163-6804.

Imagemagick Toolkit (2016). Imagemagick toolkit. `http://www.imagemagick.org/script/index.php`. Accessed: 2016-02-03.

Index, C. V. N. (2016). Cisco visual networking index: Global mobile data traffic forecast update, 2015-2020. *White Paper, February*.

Java RMI (2016). Java rmi whitepaper. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html`. Accessed: 2016-04-01.

Kemp, R., Palmer, N., Kielmann, T., and Bal, H. (2012). Cuckoo: A computation offloading framework for smartphones. In Gris, M. and Yang, G., editors, *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–79. Springer Berlin Heidelberg.

Kosta, S., Aucinas, A., Hui, P., Mortier, R., and Zhang, X. (2012). Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945--953. IEEE.

Kristensen, M. (2010). Scavenger: Transparent development of efficient cyber foraging applications. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 217–226.

Kryo (2016). Kryo. `https://github.com/EsotericSoftware/kryo`. Accessed: 2016-02-03.

Kwon, Y., Lee, S., Yi, H., Kwon, D., Yang, S., Chun, B.-G., Huang, L., Maniatis, P., Naik, M., and Paek, Y. (2013). Mantis: Automatic performance prediction for smartphone applications. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 297--308, San Jose, CA. USENIX.

Kwon, Y.-W. and Tilevich, E. (2012). Energy-efficient and fault-tolerant distributed mobile execution. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 586–595. ISSN 1063-6927.

Li, Z., Wang, C., and Xu, R. (2001). Computation offloading to save energy on handheld devices: A partition scheme. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01, pages 238--246, New York, NY, USA. ACM.

Lopez-Perez, D., Valcarce, A., de la Roche, G., and Zhang, J. (2009). OFDMA femtocells: A roadmap on interference avoidance. *IEEE Communications Magazine*, 47(9):41–48. ISSN 0163-6804.

Miettinen, A. P. and Nurminen, J. K. (2010). Energy efficiency of mobile clients in cloud computing. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 4--4, Berkeley, CA, USA. USENIX Association.

MobileWebApps (2015). Why mobile web apps are slow. `http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/`. Accessed: 2016-01-27.

NanoHTTP (2016). Nanohttp. `https://github.com/NanoHttpd/nanohttpd`. Accessed: 2016-02-03.

O'reilly, T. (2005). What is web 2.0: Design patterns and business models for the next generation of software. *Communications & strategies*, 1(1):17.

OSGi (2016). OSGi. `https://www.osgi.org/`. Accessed: 2016-02-03.

Pamboris, A. (2014). *Mobile Code Offloading for Multiple Resources*. PhD thesis, Imperial College London.

Papazoglou, M. P. (2003). Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3--12. IEEE.

Perrucci, G. P., Fitzek, F. H., and Widmer, J. (2011). Survey on energy consumption entities on the smartphone platform. In *Vehicular Technology Conference (VTC Spring), 2011 IEEE 73rd*, pages 1--6. IEEE.

Raja Vallee-Rai, Laurie Hendren Vijay Sundaresan Patrick Lam, E. G. and Co, P. (1999). Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125--135.

Saarinen, A., Siekkinen, M., Xiao, Y., Nurminen, J. K., Kemppainen, M., and Hui, P. (2012). Can offloading save energy for popular apps? In *Proceedings of the Seventh ACM International Workshop on Mobility in the Evolving Internet Architecture*, pages 3--10. ACM.

Satyanarayanan, M. (2001). Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17. ISSN 1070-9916.

Scheme, B. T. (2009). LTE: the evolution of mobile broadband. *IEEE Communications magazine*, page 45.

Schlachter, F. (2013). No Moores Law for batteries. *Proceedings of the National Academy of Sciences*, 110(14):5273--5273.

Shannon, C. E. (2001). A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3--55. ISSN 1559-1662.

Shi, C., Habak, K., Pandurangan, P., Ammar, M., Naik, M., and Zegura, E. (2014). 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, pages 287--296, New York, NY, USA. ACM.

Shye, A., Scholbrock, B., and Memik, G. (2009). Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 168–178. ISSN 1072-4451.

Smartphone Market Share (2015). Smartphone os market share, 2015 q2. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. Accessed: 2016-02-29.

SNAP Pennsylvania (2016). SNAP - Pennsylvania road network. `https://snap.stanford.edu/data/roadNet-PA.html`. Accessed: 2016-03-01.

Stoica, I., Morris, R., Karger, D., Kaashoek, F. M., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, page 149/160, San Diego, California, United States. ACM New York, NY, USA, ACM New York, NY, USA.

Tilevich, E. and Smaragdakis, Y. (2008). NRMI: Natural and Efficient Middleware. *Parallel and Distributed Systems, IEEE Transactions on*, 19(2):174–187. ISSN 1045-9219.

Trepn Power Profile (2016). Trepn power profile. `https://developer.qualcomm.com/software/trepn-power-profiler`. Accessed: 2016-02-03.

Trepn Power Profile Forum (2016). Trepn power profile forum. `https://developer.qualcomm.com/blog/tips-using-trepn-profiler`. Accessed: 2016-02-03.

Vallee-Rai, R. and Hendren, L. J. (1998). Jimple: Simplifying java bytecode for analyses and transformations.

Verbelen, T., Simoens, P., Turck, F. D., and Dhoedt, B. (2012). Aiolos: Middleware for improving mobile application performance through cyber foraging. *Journal of Systems and Software*, 85(11):2629 – 2639. ISSN 0164-1212.

VMWare (2016). Vmware. `http://www.vmware.com`. Accessed: 2016-03-01.

Wu, H., Knottenbelt, W., and Wolter, K. (2015). Analysis of the energy-response time trade-off for mobile cloud offloading using combined metrics. In *Teletraffic Congress (ITC 27), 2015 27th International*, pages 134–142.

Yang, S., Kwon, Y., Cho, Y., Yi, H., Kwon, D., Youn, J., and Paek, Y. (2013). Fast dynamic execution offloading for efficient mobile cloud computing. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, pages 20–28.

Zhang, L., Tiwana, B., Dick, R., Qian, Z., Mao, Z., Wang, Z., and Yang, L. (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 105–114.