Improving the performance and usability of an offloading engine for Android mobile devices with application to a chess game

submitted by
Joan Martínez Ripoll

Master Thesis

Faculty IV - Electrical Engineering and Computer Science
Technical University of Berlin

Supervisors:
Prof. Dr.-Ing. Adam Wolisz
Prof. Dr. Katinka Wolter

09 October 2013
Abstract

Offloading computation from smart-phones to nearby resource-rich PCs (known as surrogates) has been recently re-discovered as a technique to address the limited processing power and battery lifetime of the mobile phones, enabling the opportunity to provide computation-intensive applications to the user without high costs.

Intensive computation applications implementing video games, optical character recognition, voice and image processing or 3-D modeling can be enhanced by offloading the heavy computational application parts to resourceful servers and receiving the results from these servers while reducing energy consumption and execution times at the smart-phone.

In this thesis, the different characteristics of the offloading systems will be studied, as well as the most relevant actual implemented offloading systems. As a result of the acquired knowledge, an offloading system will be implemented and presented: the offloading engine.

The offloading engine has a task execution time saving approach, offloads at the fine-grained granularity, and performs dynamic decisions that take into account the network conditions, the amount of data that needs to be transferred to the surrogate for the task's execution, and the task's input parameters. The consideration of the input parameters of the task is a feature that is relevant because, since different input parameters of a same task lead to different number of instructions to execute, the local or remote execution will be preferable.

The offloading engine is tested using 3G and Wi-Fi connections, linear costs methods, and a chess game. It is concluded that the offloading system can save significant amounts of time, saving up to 72 seconds in a mid range Android smart-phone when determining the 20 next moves of the chess game.
Contents

1 Introduction

2 Background and related work
   2.1 Characteristics of the offloading systems
      2.1.1 The application partitioning
      2.1.2 The offloading decision
      2.1.3 Serialization
      2.1.4 Interoperability
      2.1.5 Virtualization
      2.1.6 Mobility and fault tolerance
      2.1.7 Privacy and security
   2.2 Related offloading systems
      2.2.1 CloneCloud
      2.2.2 MAUI
      2.2.3 PANDA
      2.2.4 Cuckoo
      2.2.5 MACS
      2.2.6 Summary
   2.3 Fault tolerant systems
      2.3.1 Wait for the optimal moment to launch the local re-execution
      2.3.2 Check-pointing
      2.3.3 A monitoring technique based on a Markov Chain mode
## 3 The offloading engine

3.1 Objectives and motivation ........................................... 29
3.2 An overview of the engine ............................................ 30
3.3 System characteristics ............................................... 31
3.4 System architecture .................................................. 33
3.5 Obtaining the parameters involved in the offloading decision .................................................. 36
    3.5.1 The network and server availability ....................... 36
    3.5.2 RTT and bandwidth ........................................... 36
    3.5.3 Android and server estimations of the task execution time ........................................... 38
3.6 The Android side ................................................... 43
3.7 The server side .................................................... 44
    3.7.1 The communication with the server ...................... 44
    3.7.2 The server files .............................................. 44
    3.7.3 The server overload .......................................... 45
    3.7.4 The problem of two different language virtual machines ........................................... 45
3.8 The web interface ................................................... 46
    3.8.1 A system to upload new application parts ................ 46
    3.8.2 The automatic cost estimation ............................ 46

## 4 Testing

4.1 Hardware ............................................................. 47
4.2 The engine testing application ..................................... 48
    4.2.1 Comparing 3G and Wi-Fi connections ..................... 50
    4.2.2 Executing the methods ...................................... 53
4.3 The chess game .................................................... 55
    4.3.1 Easy difficulty .............................................. 58
    4.3.2 Normal difficulty ........................................... 59
    4.3.3 Hard difficulty .............................................. 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.4 Evaluation</td>
<td>61</td>
</tr>
<tr>
<td>5 Conclusions</td>
<td>63</td>
</tr>
<tr>
<td>5.1 The offloading engine further work</td>
<td>65</td>
</tr>
<tr>
<td>6 Appendix A: Saving time means saving energy?</td>
<td>66</td>
</tr>
<tr>
<td>7 Appendix B: How to use the Offloading Engine in an Android application</td>
<td>70</td>
</tr>
<tr>
<td>Bibliography</td>
<td>71</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Smart phones have nowadays a strong presence in the daily life of millions of people in the world. The popularity and usage of these devices from part of the population has constantly increased since their launch in the market. In fact, as shown in Figure 1.1, 620 millions of smart-phones were sold during 2012, and during 2011 and 2012 smart-phones sales overcame the PCs ones [Res11]. Even more, different smart-phone sales forecasts indicate that the growing tendency will remain [Cod13] [IDC13] and it is predicted that by 2015, mobile application development projects will outnumber native PC projects by a ratio of 4:1 [GHL12].

![Figure 1.1: PC and smartphone sales per year by Gartner [Res11].](image)

All of this, combined with the actual demand of these devices, has led companies and freelance developers to have a strong interest in creating applications for them. As a result, we have now a complete and huge market of applications that offer different functionalities. However, some of these applications use the device resources too intensively, consuming big amounts of battery and taking long times of executions in some of their activities.

Computation offloading has been discovered as a solution to increase the mobile systems' capabilities by migrating computation to more resourceful computers (i.e. servers) and receiving the required result from them. If a user wants to execute computation intensive applications, avoiding the smart-phone to perform the computation will free its resources and save battery and time, leading to a better user experience. In addition, this also provides a new concept and opportunity of creating content-rich and computation-intensive applications without losing portability.
Furthermore, smart-phone battery studies show that energy is and will remain being the primary bottleneck for mobile devices [Pow95], so improving the devices energy consumption with computation offloading techniques seems to be the best method to follow.

Applications like mobile games that require artificial intelligence, image treatment for purposes such as face recognition or optical character recognition, virus scanning, text files indexing and voice recognition are perfect candidates for computation offloading.

The aim of this thesis is to 1) study in detail the concepts related with the computation offloading topic as well as relevant systems that implement this technique in order to 2) create an offloading system based on the learned knowledge to test and evaluate the benefits of the offloading computation techniques in nowadays smart-phones.

To achieve the creation of the system, new ideas and new combinations of computation offloading concepts are closely treated to result in a simple offloading engine for Android devices that decides whether some parts of an application are preferable to be executed locally or in a server. The offloading engine is inspired on the previous work “Mobile devices computation offloading” [GM13], but several improvements present in the actual offloading engine lead both systems to have little in common.

For the testing and evaluation of the offloading system, different applications of it are performed, being the most important one a chess game.

Finally, another important aspect of the thesis and the resulting offloading system is that tasks that have different computation costs depending on some input parameters (e.g. the recursive algorithm for obtaining a number of the Fibonacci sequence) deserve a special attention. In other words, instead of focusing into tasks that always have expensive computation costs, a closer approach is dedicated to tasks that, depending on their input parameters, are worth to be executed locally or remotely. This aspect is considered as relevant during the thesis, hence, it is also considered by the offloading engine for the local or remote execution decision.
Chapter 2

Background and related work

2.1 Characteristics of the offloading systems

In this section, different aspects about the offloading systems, decisions that need to be taken into account when developing one, and closely related concepts will be exposed.

2.1.1 The application partitioning

The partitioning of an application is an important step in order to determine which parts of a program are susceptible of being offloaded.

The granularity

The first point to consider when partitioning a system is its granularity, i.e. the extent to which the system is broken down into small parts. There are two blocs to consider:

- **Coarse-grained** systems consist of few and large components that regard large sub-components.
- **Fine-grained** systems regard smaller components of which the larger ones are composed. This approach is considered to lead to large energy and time savings because only the sub-parts that benefit from remote execution are offloaded [CBC+10].

Different researches on developing infrastructures for offloading at various granularities have been done. Offloading may be performed at the levels of threads, methods, tasks, applications, or virtual machines [KLLB13].

The determination of the partitions

The second point to consider is the manner in which the partitions are determined: manually by the programmer or automatically.
CHAPTER 2. BACKGROUND AND RELATED WORK

- *Manual partitioning* is said to be efficient, as it is supposed that the programmer is aware of the program’s computation expensive parts and can determine the application’s partitioning in an advantageous way. Furthermore, no computation is required in order to determine the partitions. As a disadvantage, performing the partitioning can pose a burden to the programmer, especially if no offloading techniques were considered at the outset and a restructuring of the program is required. This can be considered as an important problem, because most of the actual mobile applications do not separate the computation expensive code sections from the others, which means that they cannot be used without further work in offloading systems that require a manual partitioning.

- *Automatic partitioning* is done by a partitioning mechanism that can use dynamic program analysis (such as dynamic profiling) and/or static program analysis to solve the matter. As a consequence, this system permits applications with any kind of code structure to be offloaded. However, the computation for determining the partitions automatically is very expensive. This causes the partitioning mechanism to be usually performed only once, since it undesirable to partition a program several times during execution due to the very high overhead for analyzing it [KLLB13]. Another possible drawback is that the resultant partition depends entirely on the partitioning mechanisms, which could not offer the best problem solution.

2.1.2 The offloading decision

The offloading decision can be static or dynamic.

A *static decision* requires the program to be partitioned during its development in a manual way. As an advantage, static decisions have low overhead during execution because the program does not have to decide where the program parts will be executed. As a disadvantage, this offloading decision method is only suitable when the parameters can be accurately predicted in advance.

For their part, *dynamic decisions* are able to adapt to different run-time conditions, such as changing network conditions. Prediction mechanisms based on the task’s execution time or its battery consumption are also often used to anticipate if offloading would be beneficiary or not. The disadvantage is that these predictions added to the monitoring of the run-time conditions can lead to a high overhead.

**Factors involved in a dynamic decision**

When offloading a computation expensive task, there is always a trade-off between the energy and time that is required for the local execution and the energy and time that is needed to offload. Offloading involves sending data, waiting for the task to be executed, and receiving the response data. Therefore, offloading all of the phone’s tasks to the cloud is not always recommended, since this can lead to valuable energy and time losses. Dynamic decisions aim to predict the best solution either from a time saving approach or an energy saving approach.

From a time perspective, being $I$ the number of instructions to be executed, $X_{local}$ the speed of the local execution in instructions per time, $X_{server}$ the speed of the remote execution in instructions per time, $D_s$ and $D_r$ the data to be sent and received respectively, and $B_s$ and $B_r$ the sending and reception bandwidths, offloading will save time if it is performed when:

$$\frac{I}{X_{local}} - \frac{I}{X_{server}} - \frac{D_s}{B_s} - \frac{D_r}{B_r} > 0$$

(2.1)
2.1. CHARACTERISTICS OF THE OFFLOADING SYSTEMS

From an energy perspective, being $P_{ic}$ the power consumed when executing locally, $P_i$ the power consumed when the device is idle, and $P_s$ and $P_r$ the power consumed for sending and receiving the data respectively, offloading will save energy if it is performed when:

$$\frac{P_{ic} * I_{X_{local}}}{X_{server}} - \frac{P_i * I_{X_{server}}}{B_s} - \frac{P_s * D_s}{B_r} - \frac{P_r * D_r}{B_r} > 0 \quad (2.2)$$

From this formulas it can be seen that, on the one hand, the most complex the task is, the most interesting offloading becomes and, on the other hand, having a small bandwidth and a high amount of data to transfer decreases the offloading attractiveness [LT11].

Depending on the approach of the offloading system and its preciseness, the variables are chosen and transformed into more or less accurate values that determine the outcome of the decision. As a final note of this section, some offloading systems involve too status information of the mobile device, such as remaining memory or battery, into the offloading decision.

2.1.3 Serialization

Serialization is the process of translating objects or data structures into a format that can be stored permitting to be later retrieved in the same or another computer. Serialization is applied when, for example, a file is saved or before sending information through a network.

In the offloading field, deciding and standardizing how and what information will be serialized before it is sent to the server is not irrelevant, as the server must receive the necessary information to process the offloaded task and receive it in a way that preserves the data types. For instance, if an object has to be serialized, all its data structure must be too serialized, which at the same time could involve serializing other nested objects.

Moreover, the serialization and de-serialization processes must be performed when sending the information to the server, and when receiving the answer from it.

2.1.4 Interoperability

Different types of mobile devices are able to connect and interact with one or many servers across a variety of different networks. Nowadays, the most relevant networks used in mobile devices are Wi-Fi, 3G, and LTE, being still Wi-Fi and 3G the most usual. It is very common in these devices to switch from one of this networks to another, which leads to network speed, latency and energy consumption changes that should be taken into account for the offloading decision. As an example, Android and iPhone mobile devices switch to 3G signal when there is no Wi-Fi available and, when using 3G, some are too able to switch back again to Wi-Fi if a known and registered Wi-Fi network is available. Due to the fact that 3G connection is typically slower [KLLB13] and consumes more power than Wi-Fi [LLY+13], the offloading benefits will vary, even causing offloading not to be worth.

Furthermore, the interactions (or non-interactions when this is not possible) between mobile devices and servers as well as the possible decision changes that are consequence of connection changes must be hidden from the user.
2.1.5 Virtualization

Virtualization is a very common used feature for offloading services, as it allows to run arbitrary task, originally designed for a smart-phone on one or more virtual machines, increasing utilization and flexibility and making offloading more feasible. This is the consequence of the virtualization ability to provide hardware and operating system independence, which permits the servers to offer computing as a service without having to modify the original codes of the application parts to offload. Nowadays, this independence solves the differences between the different smart-phone and server OS and their architectures (typically ARM architecture for the smart-phones and x86 for the servers). In addition, virtualization can also permit to perform advanced techniques such as sending program states or in-memory data structures from one machine to another and can too offer programs and their data to be isolated and protected at the servers.

Migration from mobile device to server can be done virtualizing the OS or using application-layer virtual machines like the Java virtual machine (VM), the Common Language Run-time VM or the Dalvik VM from the Android Platform. Nevertheless, application-layer virtual machines cannot be performed in all mobile devices’ OS. For instance, Apple’s iOS applies no application-layer Virtual machine but, in contrast, the Android OS applies it due to the previously mentioned Dalvik VM. In these cases, virtualizing the OS is the option to follow.

2.1.6 Mobility and fault tolerance

Mobile devices suffer frequent location changes and rely on wireless networks to connect with the servers. As network or server failures may happen, the offloading system must handle them in order to provide a reliable service that permits the system to continue the execution of the application.

When a server failure happens during the offloading of a task, another available server to offload the task can be used. However, if this is not possible or network failures happen, a common approach is to redirect the task to a local execution.

As it will be shown in the studied offloading systems for smart-phones of this thesis, local re-execution is a very used network failure solution. Two reasons cause it to be adequate. On the first hand, other existent network tolerance solutions require an excessive energy and time overhead for a smart-phone environment. On the other hand, when a mobile device loses its connection, there is no real guarantee that it will be recovered in a short period of time because, if this happened due to mobility reasons, an unpredictable time could pass until the network is recovered.

Nevertheless, other fault tolerance proposals are exposed in this thesis at the Section 2.3.

2.1.7 Privacy and security

In offloading systems, users’ data and programs are sent to servers which the users do not have under their control. Therefore, security and privacy settings depend on the IT management of the cloud server provider, resulting in different consequences regarding privacy and security that must be taken into account.

On the first hand, a bug or security loophole in the cloud server could lead to a breach of privacy. In fact, this gains a strong relevance when analyzing past events in the cloud services. For example, in March 2009, a Google bug caused to share documents without the knowledge of their owners’ [Kin09], or in July 2009 a Twitter breach permitted a hacker to acquire confidential documents [McM09].
On the other hand, cloud service providers work typically with different third-party vendors and the users have no guarantee on how these vendors safeguard data and determine its access rules.

To solve this issues, different studies to protect outsourced data have been proposed. The solutions include encrypting the data before storing it and stenography, a technique to transform the data in a way that operations can be performed without exposing them [KL10]. However, most of these solutions have limitations. Encryption keys could be too large and excessively increase the amount of data, efficient computation on encrypted data is still a topic of research [KLLB13] and finally, as most important, causing mobile devices to perform encryption or stenographic techniques before sending the data to the server would require an additional processing that would lead offloading to not be worth.

To conclude, transmitting the data is also a security concern. Depending on the amount of data to send and receive, encrypted communication channels can be considered.
2.2 Related offloading systems

A variety of offloading systems have been created. Depending on their objectives, different methods have been used for implementing them, leading each system to have its own particularities. In this section, the offloading systems that have been considered to be more related with the objectives of this thesis will be exposed. For each system, a summary, an analysis, and observations including what has been of inspiration for the implementing of the thesis’s offloading system will be described. Depending on the interest of the offloading system and the amount of information that it contributes, the explanations will be more or less detailed.
2.2. RELATED OFFLOADING SYSTEMS

2.2.1 CloneCloud

Summary

CloneCloud [CIM+11] is a system that improves execution times and energy of mobile applications by offloading the most indicated part of their execution to device clones running in a computational cloud.

A fundamental idea of the CloneCloud system is also a fundamental concept for offloading systems in general: as long as executing application parts on the cloud improves the overall performance (in terms of time, energy, reliability, etc.) of the application, sending the necessary files, data and code to the device clones on the cloud will still be worth it.

The system makes too a strong emphasis on indicating its capability to automatically transform unmodified applications by performing a partition of them at a fine granularity. This partitioning is performed by separating between parts that have the most computational expensive characteristics and the other ones. This way, at the same time, the computational expensive parts are also determined to be executed on the cloud while the other are determined to be executed locally at the mobile device.

CloneCloud indicates that application-layer virtual machines are widely used in mobile platforms, which avoids having to modify applications executables and permits migrating them to more resourceful computers. Therefore and because of the Dalvik VM, the prototype to implement the system is implemented for the Android OS.

The migration to the cloud is done at the thread granularity. Internal state of the Application-layer virtual machine is used to determine the necessary information for executing the thread remotely. This state is then packed and sent to the cloud, where it is placed at the same address space of the VM, simplifying the migration procedure.

Finally, the system offers the possibility to encrypt the data that is transmitted, but no security and privacy measures are specified for the files that are cloned at the cloud.

The partitioning mechanism

The CloneCloud paper shows a strong importance on the automatic partition mechanism of the system. In fact, it is the only modern studied offloading system, designed for actual smart-phones, that offers this feature.

The partitioning mechanism is executed only once per application, it does not need the source code of the application, and it does not require the programmer to write the application in a specific way.

A set of execution conditions (CPU speeds, network characteristics, and energy consumption) is given and, through static analysis combined with dynamic profiling, an application partition that also indicates local or remote execution for each part is determined. This configuration is said to optimize the energy consumption or total execution time of the application. The partitioning mechanism is run multiple times with different execution conditions, which results in a database of pre-determined partitions.

Finally, when the Application is running and the cloud resources and network are available, a partition from the database is picked and applied with some fast and small modifications.
Paper analysis and observations

Clone cloud proposes to continuously have a synchronized copy of the mobile device contents in the cloud in order to offload operations like file searches or virus scanning. However, little information is given about how the File-system synchronization between the device and the cloud is done and what energy consequences this has. In fact, this characteristic of the system is criticized by the Cuckoo framework [KPKB10], commenting that this is a costly process that can be avoided by temporary cloning only the service that the application needs to offload. Furthermore, most of the applications of the smart-phone market do not require a multitude of files from the device for their execution, which makes the concept of having a clone of the phone less interesting.

Moreover, for implementing the system, the Dalvik VM was modified in order to apply the dynamic profiling required for the partitioning mechanism. On a real scenario, only the modification of the Dalvik VM on the cloud clone could be applied.

Another characteristic of the system is that, to select the partition configuration from the database during run-time, only availability of the cloud resources and network link characteristics are considered. Therefore, the aim of this thesis to emphasize on the input parameters in order to take the offloading decision is not covered, as once the partitioning is established, a task will always be executed locally or remotely according to the part where it is located without further considerations.

To conclude, at the evaluation of the system, it is explained that the partitioning mechanism is performed at the server in order to avoid time and energy consumption at the mobile device. As it has previously been explained, the partitioning mechanism uses different execution sets and results in a database of pre-determined partitions. This system was of inspiration for the thesis’s offloading system when estimating execution times of the tasks. Although the generated information is completely different, the thesis’s system also performs different executions on the server to result in a database that is used for the offloading decision, which also avoids time and energy consumption on the smart-phone. The system is later discussed at the section 3.5.5.

The Android implementation of the system is also not available [Kem].
2.2. RELATED OFFLOADING SYSTEMS

2.2.2 MAUI

Summary

MAUI [CBC+10] is a computation offloading system that aims to save energy consumption of smart-phone applications by offloading at a fine-grained granularity (more precisely, by offloading methods).

Two versions of a smart-phone application are considered, one running locally on the smart-phone and another one running remotely on the server. Communication is possible between different instructions set architectures because MAUI benefits from a managed code environment by using the Common Language Runtime, an application-layer VM.

MAUI provides a programming environment to fulfill a requirement of the system regarding the partitioning of the applications: the programmers need to indicate in their code what methods are remoteable. Taking advantage of a feature from the CLR executables, this information is stored as a method attribute in the meta-data of the application compiled files. This way, once at run-time, MAUI uses reflection to identify these methods indicated as remoteable.

If the system decides to execute a method remotely, type safety combined with programming reflection are used to extract the program state that the method will need for the remote execution. More concretely, the referenced by the method in-memory data structures (e.g. the method parameters, static variables or current object variables) are serialized and sent to the surrogate using XML.

The offloading decision is dynamic and provides a global energy optimization that determines which methods marked as remoteable must be executed locally and which remotely. MAUI decides at run-time whether to offload or not the methods depending on:

- Wireless connectivity measures such as bandwidth and latency.
- Network costs for shipping the XML serialized program state extraction required for the method’s remote execution.
- CPU costs of the method given its input parameters.

A timeout mechanism is used to detect communication failures between the server and the smartphone. When a failure is detected, the system tries to find an alternative server, but if this is not possible or a network failure happens, the method is executed locally.
The MAUI profiler and the MAUI solver

These are two relevant pieces of the MAUI system that are interesting to mention.

The MAUI profiler is responsible to provide the MAUI solver with the information that the MAUI solver requires to take the offloading decision of each method. This information is:

- **Energy consumption characteristics of the device**
  This includes 3G and Wi-Fi usage consumption as well as a function to determine the energy consumption of the smart-phone according to the CPU cycles it executes. A hardware power meter that can provide fine-grained energy measurements is used to calculate both of this energy consumption metrics, resulting in concrete 3G and Wi-Fi consumption values and a simple linear model for obtaining the energy consumption depending on the CPU cycles. All of this information is obtained only once to be later stored at the smart-phone and retrieved each time the application is stared.

- **Running time and resource needs of the methods depending on their input parameters**
  More concretely, the CPU cycles required for the execution of the method, the state transfer energy requirements and the run-time duration. This information is determined through past invocations of the methods by continuously monitoring it. The continuous monitoring is important, since the changing nature of the input parameters leads to different values that need to be precisely measured for the offloading decision to be correct.

- **Network characteristics of the wireless environment**
  These are the bandwidth and latency. A 10KB data is sent to the server in order to obtain these values. When a method is offloaded, these values are re-calculated again. If no transfers happen during one minute, the 10Bb data is sent again to obtain more actualized estimations. A network change (e.g from Wi-Fi to 3G) implies too doing this process again.

For its part, the MAUI solver indicates what decision for each method of the application is better to perform using the previous described information of the MAUI profiler. The decisions of the MAUI solver are globally optimal, which means that the aim is to optimize the entire energy consumption of the program instead of the methods individually. Since the solver requires intensive computation, it is only located on the server and, while the application runs, it is periodically invoked. By doing so, the MAUI solver continuously learns from the past behavior of the program and also adapt to the changing conditions of the environment (e.g. network connection type). The module is invoked asynchronously to avoid pauses on the interactive performance of the application.

Paper analysis and observations

A **representative offloading system**:  

MAUI is an offloading system that is very representative of what this thesis aims to study. Even more, different features of the system have been of inspiration when implementing the thesis’s offloading system.

Sending a package to the server in order to obtain the RTT and the bandwidth, as well as using past executions behaviors to predict new ones, are simple ideas that not only work well when the system is applied into real applications, but also have computation needs that are not excessive for a sensitive battery environment like the smart-phone one.
Furthermore, the decision to continuously monitor the network conditions and the running time and resource needs of the methods depending on their input parameters is certainly a requirement, because changes of the used wireless network, as well as different task input parameters, are factors that have to be taken into account for the offloading decision.

**Asking the developer to mark the methods as remoteable:**

The MAUI paper proposes this to ease the programmers from thinking if a particular method makes sense to be offloaded or not. In other words, if the programmer doubts about a method being offloaded, this can be marked as remoteable and the system will decide if it is worth it or not.

At the same time, the system requires also the code of the methods marked as remoteable to follow different conditions like avoiding user interface interactions. These conditions are significantly enough to cause the programmer write the program in a determined way. Even more, the paper remarks that program code modifications and structure changes lead to performance benefits.

Therefore, a trade-off between the programmer’s involvement and the performance of the offloading system can be derived. The more the programmer gets involved and is conscious of the benefits of the computation offloading system, the better performance results are obtained. As a consequence, if the aim is to really save energy and execution times, bringing the developer to be aware of these facts and to follow some patterns can be considered as a good criterion, as in the end, he or she will also be satisfied with the results.

**Energy measurements:**

A hardware power meter is used by the MAUI profiler to determine the energy consumption characteristics of the device. Nowadays, most smartphone APIs do not offer fine-grained energy measurements, which forces offloading computation systems that focus on energy savings to use other methods. As a disadvantage, this makes it difficult to apply these offloading systems at the smartphone market, because no other energy measurements (like the MAUI’s hardware power meter) can be used. Either a database with all the smartphones’ batteries energy consumption characteristics should be provided or the APIs should improve the energy consumption measurements of the mobile devices.

**The need of waiting for the offloading decision to be taken at the server:**

Another observation of the system is that the MAUI solver always runs on the server, which implies that all the offloading decisions require a network communication.

From the energy saving consumption approach of the system, this is acceptable, because the smartphone does not consume energy while waiting for the decision. However, from a time saving point of view, it should be considered that the expended time for the communication can sometimes be significant if the network quality is poor; for instance, experiments performed in this thesis with 3G connection determine sometimes RTTs of 3000ms. As this time can be waited to conclude with a local execution decision, not all the potential offloading methods might benefit from this system.

**To conclude:**

The way to obtain the remoteable methods takes advantage of the CLR characteristic to permit adding meta-data information to the methods included at the compiled files of the application.
This makes MAUI compatible with smart-phones running under the Windows Phone OS. However, in order to augment the systems compatibility with other smart-phone OS, it should be studied if this remoteable method obtainment could be too applied on other application-layer VM or VM at the OS level.

This system is not available [Kem].
2.2. RELATED OFFLOADING SYSTEMS

2.2.3 PANDA

Summary

The PANDA [YOC08] project goal is to let mobile handsets fluently execute applications that usually run on powerful PCs. The mobile handsets offload tasks to more powerful PCs in order to solve their insufficient resource capacity when this is convenient.

PANDA proposes an interesting idea: Users with mobile handsets can search for new mobile services from the Internet, download them, and use them locally or remotely on the servers where they reside.

The developers of the project encourage cafes, airports, or in general service providers to use their desktop PCs to provide offloading tasks. In business terms, this can either improve their customer-friendliness or a high-level offloading service can be provided.

The system is designed to run under Java VM and the granularity is performed at the Java class level. In other words, each Java class file of the service is susceptible of being executed locally or remotely. The applications are monitored, so that information about CPU and memory costs is collected for each class. A partitioning module combines this information with the network connection bandwidth and a speedup factor between the device and the surrogates to decide which classes must be serialized and migrated to the surrogates. Also, to take the offloading decision, the partitioning module uses profiling techniques to determine:

- **Class dependencies**, including method calls or data usage of other classes, because this implies the migration of the dependent classes too.
- **Classes that handle user interactions** or directly access I/O devices, because they cannot be migrated.

Finally, the system also considers using different surrogates to execute different tasks at the same time.

Paper analysis and observations

The speed factor between surrogate and mobile device mentioned in the system paper was of inspiration when building the thesis offloading engine, as a speedup factor variable has been also used. The experiments performed with the system also show a factor that this thesis studies: the relevance of the input parameters of the task to offload when deciding to offload or not. Concretely, an image with a text is captured and the text must be recognized through OCR techniques. If the captured image has little text in it, local execution is preferable but, on the other hand, when the amount of text increases, offloading increases too the execution time benefits.

Although the system is available [Yan08], different reasons cause it to not be suitable for this thesis. On the first hand, the Java class granularity is not enough fine-grained for the aim of this thesis. Even more, this measure depends too much on how the programmer has written the application. For example, the whole application could be compiled in a single class, hence it would be not possible to offload because user interface interactions would be included. On the other hand, the PANDA project was developed during 2007 and the system is tested in a PDA running under Windows Mobile 2003 OS. In general terms, it can be considered to be based on old mobile system characteristics and not fully actualized for the current smart-phone market (which at that time was starting to emerge).
Finally, sending class files to different surrogates can be a disadvantage if they are too big.
2.2. RELATED OFFLOADING SYSTEMS

2.2.4 Cuckoo

Summary

Cuckoo [KPKB10] is a framework with an Eclipse integration that aims to save energy and execution times of Android smart-phone applications by offloading their computation intensive application parts to servers with the Java VM.

Cuckoo, as a framework, relies strongly on the developer in order to achieve the benefits of computation offloading. The programmer must write two versions of the computation intensive application components that can be offloaded, one for the local execution, and another one for the remote execution. This way, different libraries and algorithms implementations can be used for each version, and remote servers can benefit from full performance features such as parallel computation.

Cuckoo uses a builder responsible of compiling the remote code and bundle it in a .jar file to be later installed at the cloud resources. All code, local and remote, is kept together, so that the user always has a compatible remote implementation when the surrogates are not available.

The system always offloads a task if it is possible: when a remote resource is reachable, the task is offloaded and, if this is not possible, local execution is performed. Therefore, network failures are too solved.

Finally, no security for the remote services is provided: untrusted phones can access the remote resources and can too install any code onto the system.

Server and smart-phone communication: sending the remote resources

Ibis communication middle-ware [vNMW+05] is used for the communication between smart-phone and server. A communication channel must be first established, then, a particular method with particular arguments can call the server and receive the returned result.

It may happen, though, that the method is not available on the server. In this case, the .jar file containing the compiled code for the remote service is sent and installed at the server. This procedure adds the overhead of sending the .jar, but it occurs only once per server service unavailability and, from then on, many invocations can be done.

Furthermore, Cuckoo offers two methods for discovering the servers and send to them the compiled methods if necessary. One of them optimizes for the time speed and the other one for energy savings. The programmer may choose between the two of them:

- With early binding, time speed is optimized. Cuckoo tries to connect with a server and, if the remote resources are not installed, it sends and installs them when the Android application starts or when a connection is reestablished. This way, when the methods are invoked, they are already installed and can be called right away.

- With with late binding, energy is optimized. The resource discovery process is delayed until the method is invoked. This avoids energy overhead, because unnecessary connections and possible remote resource sending that might happen due to network disconnections are not performed.
Paper analysis and observations

The paper about the Cuckoo framework has a very technical character and explains several implementation features for building an offloading system for Android smart-phones.

As declared in the paper, Android platform is chosen because it provides an application model that fits well for computation offloading in contrast with other popular platforms such as the iPhone. This statement, added to the useful and detailed implementation and technical information about Android smart-phones, were relevant for deciding Android OS to be the OS of the offloading system of this thesis.

A special implementation feature to mention is the sending of the compiled methods as a .jar file to the servers. This procedure was of inspiration for building the offloading engine system to upload new application parts (Section 3.8.1).

This framework shows again the relation between the programmer involvement and the overall throughput of the offloading techniques, as it proposes the programmer to write two versions of the code in order to obtain the maximum benefit from offloading.

Beyond the technical information, Cuckoo differs from the objective of this thesis because, as all the tasks are executed remotely when possible, no offloading decisions are taken. This is also a debatable point of the system, as it is known that offloading always a task does not always save energy and execution time.

This system is too not available [Kem].
2.2. RELATED OFFLOADING SYSTEMS

2.2.5 MACS

Summary

MACS [KYK12] is a framework that has similar characteristics with Cuckoo: it offloads parts from Android applications to obtain the corresponding result, it needs the programmer to write a local and a remote version of the computation expensive application part, the server side runs under the Java VM, and it sends to the server the class compiled files of the remote services as a .jar file if they are not already installed.

The main difference is that Cuckoo offloads whenever it is possible whereas MACS takes the offloading decision dynamically. From each Android service, the following information is monitored and saved as meta-data of it:

- The memory consumption on the mobile device.
- The code size of the compiled code.
- The dependency information on other services, in terms of the extra data to be transferred through the network that this implies.

This information, in addition to the data transmission costs depending on the bandwidth, is used to calculate energy, time, and memory criteria that determine if offloading would be advantageous or not.

The Android interface definition language (AIDL) is used for the services to facilitate the communication during the application execution, as it is also combined with Remote Procedure Call (RPC).

Paper analysis and observations

The consideration of the memory costs of the applications for the offloading decision is particularity of the MACS offloading system. This information is used to avoid a service from consuming more memory than the available memory on the mobile device. Probably, this characteristic MACS is due to its willing to offload memory intensive applications such as face recognition on videos.

As an interesting data, the energy criterion obtains the Wi-Fi and processor phone consumption energy values using an Android application called "PowerTutor" [ZTQ+10]. This application was tried to obtain this information from the Android devices used for the testing of the thesis’s offloading system, but it is based on components knowledge and it was not able to recognize the smart-phones used for the testing.

Finally, the energy and time criteria of the Android services require an estimation on the number of instructions to be executed. This estimation is done based only on the code size. In the paper, it is mentioned that a better estimation could be done taking the input parameters into account, which is one of the points that this thesis treats.

The framework was not found on the Internet.
2.2.6 Summary

The two following tables summarize the most important points of the previous offloading systems.
### 2.2. RELATED OFFLOADING SYSTEMS

<table>
<thead>
<tr>
<th>System</th>
<th>Granularity of the partition</th>
<th>Partitioning determination</th>
<th>Serialized information to send to the server</th>
<th>Saving approach</th>
<th>Decision</th>
<th>Virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloneCloud</td>
<td>Fine G. Thread migration.</td>
<td>Automatic.</td>
<td>Internal state of the AL-VM used to determine the thread state. State packaged and sent.</td>
<td>Time and energy.</td>
<td>Dynamic.</td>
<td>AL-VM Dalvik running on both the server and the smart-phone.</td>
</tr>
<tr>
<td>Cuckoo</td>
<td>Fine G. Android services (application parts).</td>
<td>Manual.</td>
<td>Requests done to the server with the method name and its particular arguments. Ibis communication middleware used for the communication.</td>
<td>Time.</td>
<td>Almost static. Only local executions are performed when there is no network connection.</td>
<td>Server runs under the Java VM and the smart-phone under the Dalvik V. Therefore, no state can be sent and the remotable tasks are stateless.</td>
</tr>
<tr>
<td>MACS</td>
<td>Fine G. Android services (application parts).</td>
<td>Manual.</td>
<td>RPC communication between services written under the AIDL interface.</td>
<td>Time, energy and memory.</td>
<td>Dynamic.</td>
<td>Server runs under the Java VM and the smart-phone under the Dalvik V. Therefore, no state can be sent and the remotable tasks are stateless.</td>
</tr>
</tbody>
</table>

Table 2.1: Offloading systems overview 1.
<table>
<thead>
<tr>
<th>Network failure treatment</th>
<th>Programmer burden</th>
<th>OS for the implementation</th>
<th>Security and privacy</th>
<th>Year</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cuckoo</td>
<td>Local re-execution.</td>
<td>Android.</td>
<td>Jar files with the compiled code are stored at the server. Remote resources can be accessed by untrusted phones which can install any code onto the system.</td>
<td>2010.</td>
<td>No.</td>
</tr>
<tr>
<td>MACS</td>
<td>Local re-execution.</td>
<td>Android.</td>
<td>Jar files with the compiled code are stored at the server.</td>
<td>2012.</td>
<td>No.</td>
</tr>
</tbody>
</table>

Table 2.2: Offloading systems overview 2.
2.3 Fault tolerant systems

As described in Section 2.1.6, offloading systems must provide reliable services that are tolerant to network or server failures.

However, after analyzing the related offloading systems and as commented in [SLM+12] and [Kem], computation offloading papers take little attention to networking [Kem] and numerous offloading works assume reliable communication between the mobile device and the cloud [SLM+12].

As this is an important part of the offloading systems, a research out of the related studied offloading systems papers scope was done to accomplish the objective of learning new proposals for fault-tolerant systems.

In this section, fault tolerant proposals that were considered for the offloading system of the thesis will be exposed.

2.3.1 Wait for the optimal moment to launch the local re-execution

A concrete study to determine the appropriate moment for launching the local re-execution on smart-phone devices is done at [WWGM]. The paper exposes that, when the wireless connection is lost, the immediate local re-execution of the failed offloading task may not be cost-effective. It might happen that the wireless network recovers in a moderate time, permitting to resume the offloading task and still spending less time and energy than with an immediate locate re-execution.

A timeout until local re-execution launching is proposed. The timeout is obtained by multiplying a percentage with an estimated time containing the entire execution of an offloading task. Being $T_w$ the time to wait, $T_{se}$ the execution time of the task on the server, $T_{ds}$ and $T_{dr}$ the required time for sending and receiving the data of the offloaded task respectively, and $P$ the percentage, $T_w$ can be expressed as:

$$T_w = P \times (T_{se} + T_{ds} + T_{dr}) \quad (2.3)$$

The optimal timeout percentage is found through synthetically comparing its performance under different network conditions and it is determined to be 150%.

The conclusion is that waiting this time for the local re-execution can be an advantage, because the network might recover during the wait, permitting to try offloading again.
This system was first considered, but it was found that, from a time saving point of view, it is optimal only in certain cases. Between network connection and disconnection, a recovery time $T_{rec}$ is needed. Tasks such that $T_w < T_{rec}$ will always wait pointlessly, because they will always be re-executed locally. Observations about this issue were carried out:

*Firstly*, $T_{rec}$ can be very variable, because when a smart-phone loses its connection, there is no real guarantee that it will be recovered in a short period of time. This happens due to the mobile nature of the smart-phones and the network contracts that the user might have. For a clearer understanding of this variability, four recovery scenarios are described as it follows depending on the smart-phone’s user contracts and mobility:

- **Fast recovery 1**: The user has a 3G contract. When the user leaves a Wi-Fi area, 3G is enabled.
- **Fast recovery 2**: The user moves around a Wi-Fi coverage area, losing and recovering the connection.
- **Slow recovery 1**: The user is in an unreachable network area, such as the basement of a building, the interior of a tunnel or a subway.
- **Slow recovery 2**: The user has no 3G contract and moves from one place to another. Until a new Wi-Fi access point is reached, the network will not be recovered.

*Secondly*, even if the network recovery is fast, $T_{rec}$ is not negligible. For instance, experiments satisfying the two previously described fast recovery cases under the Android devices used in this thesis (Section Hardware) showed $T_{rec}$ values that fluctuated between 5 and 10 seconds.

*Consequently*, this method is interesting if:

1. It is known beforehand that $T_w$ will mostly require more than $T_{rec}$ seconds. For instance, this can happen if it is known beforehand that the task is always computational expensive (i.e. $T_{se}$ values always oscillate between several seconds) and the users are as described in fast recovery 1 and fast recovery 2.

2. The offloading system has an energy-saving approach, because causing the device to wait will not cause to consume a disproportionate amount of extra energy.

On the one hand, this thesis does not dismiss the time-saving offloading approach. On the other hand, tasks requiring different data transfers and with very variable execution times depending on the input parameters are also a studied subject, causing suppositions about $T_w$ to not be suitable. As a consequence, this method was discarded.
2.3.2 Check-pointing

A model to express the performance of fault tolerant systems based on check-pointing is proposed at [OWYZ08]. The offloading system must periodically checkpoint the execution state of the offloaded tasks on both, the mobile device and the server. Concretely, the check-pointed data generated in the server side is transferred after a given interval of time onto the mobile device. This way, if a failure happens, the mobile device can load the last check-pointed data and recover the application execution from this point.

The state transition of this fault-tolerant offloading system is shown in Figure 2.1, and will be described as it follows.

![Figure 2.1: State Transitions of Fault-Tolerant Offloading Systems [OWYZ08].](image)

$S_{NE}$ represents the non-offloading execution state, i.e. a state where the execution is performed at the mobile device. $S_{OE}$ represents the offloading execution state, i.e. a state where the execution is performed at the server. $S_{CP}$ is the state where both, the mobile device and the server, take a snapshot of the application’s execution and the check-pointing data is saved in the mobile device. Finally, $S_{FR}$ is the state of the system where a failure occurred and it is being recovered.

The transitions between $S_{NE}$ and $S_{OE}$ determine an offloading execution: an offloading begin event and an offloading execution end event. In a theoretical non-failure scenario, only this events would be necessary. The mobile device would ask for an offloading service to start, and an answer from the server would be received.

Transitions to $S_{CP}$ are used for writing and obtaining the check-pointed data. If a failure in state SCP occurs, a new checkpoint operation until a checkpoint is successfully created is done.

Finally, transitions to $S_{FR}$ occur due to mobile device or server failures (such as shutdown) or server non-reachability due to the mobile device’s movement. If this happens, a failure recovery operation is performed by using the last check-point data and, once this is performed, the execution state transits to $S_{NE}$.

Mobile device and the server failures are considered Poisson processes with rate $\beta$ and $\gamma$ respectively. Due to the fact that failure recovery operations are carried out in the mobile device, a rate of failures in the state $S_{FR}$ is also considered and it is too $\beta$. 

25
The model is tested with a π calculator Java application. Places of Decimals as input of the π calculator are given in order to augment the computational cost of the application. As a result, it is discovered that the computation time for the offloading process using the check-pointing method adds, regardless of the decimals to obtain, an average overhead of around 25 seconds to the non check-pointing one. Besides the example, the proposed tolerant failure model is said to add a constant overhead that must be taken into account, hence the system is recommendable for tasks such that always have notorious benefits due to offloading (i.e tasks such that, as a consequence of a high benefit due to offloading, cause this overhead to be non significant).

As already mentioned in the previous section, this thesis aims to study tasks of variable execution times, which caused this method to be discarded.
2.3. FAULT TOLERANT SYSTEMS

2.3.3 A monitoring technique based on a Markov Chain mode

A monitoring technique based on a Markov chain model is proposed for predicting failure occurrences at [PYCL11]. The paper focuses on predicting the availability of mobile devices to be used as resources in mobile cloud computing. The motivation of this approach relies on the rapid expansion of the smart-phone market, a reality that has increased the interest of considering mobile devices as resources for large scale distributed processing.

The proposed technique aims to solve the fault problems that occur because of the mobile devices’ volatility: unstable wireless connections problems, location changes, and the availability of the resources to be used for offloading a task are predicted to solve the issue.

More in detail, three resource states are considered depending on the ability to complete an operation on the cloud:

- *Stable state*, representing an operation-available and no-fault state.
- *Unstable state*, representing an operation-available and possible fault state.
- *Disabled state*, representing an operation-unavailable state due to faults or network disconnection.

Modeling the patterns of the mobile devices’ past states, the system makes a prediction of the future ones. Probabilities of transition from one state at a time $i$ to another one at a time $j$ are considered. The previous states are used to estimate the transition probabilities and the previous states information is collected from data of the same day a week ago. Finally, failure occurrence is predicted using the most probable future state of each monitoring interval.

The system, though, is criticized at [WWGM], saying that the next state of a given resource is hardly related to previous states so that, to some degree, the accuracy of the monitoring based prediction is difficult to be verified.

Although this approach differs from the thesis’s one, it shows a different and interesting approach that could be considered for future improvements of the implemented offloading system or for other works.
Chapter 3

The offloading engine

The offloading system based on the studied knowledge is here presented: the offloading engine (OE).

The first Section of this chapter indicates the motivations and objectives that led to the implementing of the OE.

Section 3.2 exposes a general overview of the engine.

Section 3.3 explains the system characteristics of the engine, focusing on the decisions that were taken and why. Explanations about the different considered offloading concepts that were discarded are too treated here.

Section 3.4 describes the system architecture of the OE, showing the different possible work-flows of the system.

Section 3.5 contains detailed explanations on how the parameters that are involved in the offloading decisions are obtained, as well as factors that difficult this process.

Sections 3.6 and 3.7 give a detailed and technical view of the different functionalities performed by the OE at the smart-phone and the surrogate side respectively.

Finally, a web-interface that has been built for other programmers to user the OE is presented in Section 3.8.
3.1 Objectives and motivation

The main purpose when building the offloading engine has been to test and evaluate the benefits of the offloading computation techniques in nowadays smart-phones, focusing on tasks such that have different computation costs depending on their input parameters. This computation variability is a factor to take into account, since considering it can improve the system capabilities to decide when it is cost effective to execute a task locally or remotely.

In the second place, creating an offloading system from scratch also permits to understand in a clearer and detailed way the fundamentals and functioning of an offloading system.

As previously described, the related found and studied offloading systems are either not available or they have a different approach. This resulted not only in a need to implement the offloading engine, but also raised the motivation for a third objective: the intention of giving the engine a profit, enabling other developers to use the engine and test it for their applications.

Finally, although the offloading engine is a simple offloading system, it has been implemented in an ambitious and scalable way, permitting future improvements to be easily performed.
3.2 An overview of the engine

The main aim of the engine is to test how much the offloading computation techniques can help nowadays with the latest smart-phones. To accomplish this purpose, an engine for Android mobile devices has been built considering that the target applications to be improved are applications such that require intensive computation in some of their parts. It is important to remark that the engine is focused also on improving tasks such that have different computational costs depending on their input.

The engine relies on the possibility of the application parts to be executed either locally in the smart-phone or in a server. To perform this work, the code of the application parts to offload is copied both in the mobile device and in the server. This way, in case it is decided to offload, the required input parameters and data are sent (through 3G or Wi-Fi, LTE or whichever connection the mobile device has) to the server, causing the smart-phone to wait for the subsequent response.

In order to decide whether it is worth it or not to offload in a given situation, the engine estimates a time prediction for both the local and the remote execution. The lowest time estimation determines where the execution will be held.

A web interface has been created so that other developers can try the engine and test its performance with their applications. Nevertheless, the engine requires the code of the application parts susceptible of being offloaded to be previously defined by the developer. Once this is done, these can be uploaded to the server through a web-interface. No source code is needed, only compiled files. Another labour of the web-interface is to run a system to automatically estimate execution times of the uploaded parts depending on their inputs.

Although the engine was initially thought to send and receive small amounts of data from mobile to server, it has finally also been adapted to send bigger data transfers, offering more versatility and augmenting its possibilities of being incorporated in more applications. Games that require artificial intelligence (e.g. table games or strategy games) or applications performing voice recognition, image face recognition or optical character recognition, are good examples of applications that could improve their efficiency due to the OE.

The engine is meant to be compatible with all Android devices using a version equal or higher to 1.5 (API 3). This offers an extended rank of Android devices compatibility, as the current Android version is 4.3 (API 18). At the server side, an Apache Tomcat web-server has been used on purpose, because its servlets are written in Java and executed by the Java Virtual Machine, which enables a code compatibility with Android due to the Dalvik VM. This however, will be later discussed at Section 3.7.4, since not all application codes have resulted to be compatible.
3.3 System characteristics

A time saving approach

As described previously when analyzing the related offloading systems, the APIs of today’s smartphones do not offer fine-grained energy measurements [KYK12][CBC+10]. Due to the difficulties found to obtain the energy consumption of the Android devices when running a determined part of an application, a time saving approach has been considered, so that the engine improves the task’s execution times. Nevertheless, experiments performed for this thesis show that saving time can also save energy. An explanation on how improving the execution times affects the energy consumption of the smart-phone can be found at the Appendix A of the thesis.

Fine-grain granularity

The offloading engine has been designed to offload at the fine-grained granularity, as this is the best granularity to save big amounts of time and energy because only the sub-parts of an application that benefit from remote execution are considered as potential to be offloaded [CBC+10]

Manual partitioning

As in MAUI, Cuckoo and MACS offloading systems, the OE relies on the programmer to determine the potential offloading parts of the application. When assuming that the programmer knows what parts of the application are the most indicated to be offloaded this can be considered as an advantageous solution, since it brings to an offloading partitioning scheme that permits the OE to decrease the execution times of the tasks in a very efficient way and without dedicating any previous computation to solve this matter. However, this could be also considered as a disadvantage, because it augments the ease of use of the OE. Nevertheless, the decision was taken due to the willingness to prioritize execution time improvements above all.

Dynamic decisions

The decision to offload the tasks or not is dynamic because the engine must adapt its decisions depending on:

- \( N_a \) and \( S_a \): network and server availability respectively.
- \( RTT \): the RTT to the server.
- \( B_s \): network bandwidth quality for the data to be sent in bytes/ms.
- \( D_s \): data size to be sent in bytes.
- \( E_{ta}, E_{ts} \): estimation of the task’s execution time given some input parameters on the Android and the server side respectively.

Having these values, the offloading decision can be taken:

\[
\text{if } (N_a \text{ and } S_a) \text{ offloading} = E_{ta} > RTT + E_{ts} + \frac{D_s}{B}
\]  

(3.1)

Notice that the data to be received is not included as a parameter of the offloading decision, this has been considered as further work.
Serialization

The server only receives strings to identify and process the tasks. Therefore, the programmer must take into account that all the task parameters must be serialized to a string and, once at the server, de-serialized and parsed again to the corresponding type. For this reason, if object instances such as vectors, lists, or user-defined classes are required as parameters, they must too be serialized and later, at the server side, parsed to retrieve their original form. When necessary, the same system must too be applied for the server response. Automatic processes for the parameters serialization could have been performed but, since no automatic parsing can be deduced for the user-defined classes, this idea was discarded.

Two virtual machines that interpret applications written with the Java Programming Language

Both Android applications and Apache Tomcat servlets are written with the Java programming language. On the one hand, Android applications run under the Dalvik VM, a specialized virtual machine for Android OS that compiles Java classes (compiled with a Java compiler) into Dalvik executables. On the other hand, Apache Tomcat runs under the Java VM. Taking advantage from this fact, the purpose was to use the same application part code on both the server and the mobile device.

Fault-tolerance

Once the different fault-tolerance proposals for network and server failures were studied, local re-execution was decided to be the most adequate. When a failure happens while trying to offload, the task is immediately redirected to be locally executed, which offers fault-tolerance without adding time and energy overheads. This causes local re-execution to be suitable for the system, as the OE is designed to deal with very variable execution time tasks. Explanations about why the other studied fault-tolerance systems were discarded were previously described when analyzing each of them at Section 2.3.

Finally, as a support for this decision being optimal, MAUI, MACS, and Cuckoo, which are studied offloading systems that also deal with variable execution time tasks, achieve fault-tolerance using local re-execution as well.

Privacy and security

Privacy and security are a concern, because data and developers’ application parts are sent and stored at the server. In order to offer security and privacy, https is used for the data transmission and only compiled classes are stored at the server. Although https requires time for the encryption and decryption of the data and encrypting data leads to an increase of the data amount, the time impact consequences have resulted to be very small and cause this option to be preferable. Moreover, the developer can add, modify, and delete application parts to the server using the web interface, but user identification is required so that this information is protected. No data stored at the server is encrypted nor have stenography techniques been used, as this has not been considered a priority for the purposes of the OE.
3.4 System architecture

The system architecture of the offloading engine is here described through diagrams.

A summarization of the engine behaviour when reaching a potential offloading task is shown in Figure 3.1, and the corresponding work flows are shown in Figure 3.2. The parameters in the flow chart are introduced as below:

- $T_{re}$: Estimated time required for executing the task remotely.
- $T_{dt}$: Estimated time required for the data transmission. RTT is used for this estimation plus an estimation of the time that the sending of the task’s input data will require.
- $T_{lc}$: Estimated time required for executing the task locally.

![Figure 3.1: Offloading decision and task execution flowchart.](image-url)
Only tasks with small data results are suitable for the engine, as big data responses have been considered as further work, hence no estimated time for transmitting big data responses is considered.
3.4. SYSTEM ARCHITECTURE

The offloading decision is only considered when the mobile device has Internet connection. In this case, the previously described parameters are used to estimate local and remote execution times. Then, the engine decides whether to execute remotely or locally considering the lowest execution time as the best option.

However, in the offloading scenario, network failures and server failures might happen: the connection can be lost while offloading, the server could not be reachable, or the server could turn down during the processing of the task. In these cases, the failures are handled redirecting the execution to the mobile device, enabling the system to continue executing the task.
CHAPTER 3. THE OFFLOADING ENGINE

3.5 Obtaining the parameters involved in the offloading decision

In this section technical details on how the parameters of the offloading engine are obtained and why will be explained. For some of the parameters, Android OS features and its functioning, the Android API, and experimentation results will also be exposed if they were relevant to determine their obtainment.

3.5.1 The network and server availability

For the network availability, three possible connection types are considered: 3G, Wi-Fi and none.

Android broadcast receivers [dev13a] are used for constantly monitoring the network availability and the type of the connection. Each time a connection type is determined, a network type change event is triggered by the broadcast receiver.

Note that the connection type is determined not only when the application using the OE is started, but also when due to mobility reasons, the smart-phone changes from one network type to another. This, as mentioned in [CBC+10], is relevant because, since different connection types lead to different RTTs and bandwidth values, these values must be actualized each time the network type change event is triggered.

The server is initially considered to be not available until a RTT is successfully obtained. However, due to server failures, the server can be set to unavailable again.

3.5.2 RTT and bandwidth

To obtain the RTT, ten requests which require no extra data transfers apart from sending a short string are done to the server. The server responds the requests also with a short string and then the RTTs are obtained to be later averaged. Once this value is obtained, a small package is imminently sent to the server in order to obtain the bandwidth. Both of these methods are inspired in [CBC+10].

Although this is the best system that has been found, RTT values can be sometimes very unstable. Two relevant factors must be taken into account for understanding such instability.

**The Android power save mode:**

Power-save mode (PSM) is an option that the users from some Android smart-phone can enable to save power consumption. Different options can be selected, including the behavior of the Wi-Fi connection, which is of interest. However, the PSM option (and thereby the Wi-Fi connection behavior options) can change significantly depending on the Android version.

Some versions disable the Wi-Fi radio when the screen is turned off. Turning off the screen is another preference that the user can define by determining a period of time where no interaction with the smart-phone has happened. If this is the case and during the offloading of a task the screen turns off, the Wi-Fi connection will be temporarily disabled and the RTT will continue increasing until the screen is enabled again.
3.5. **OBTAINING THE PARAMETERS INVOLVED IN THE OFFLOADING DECISION**

Other versions, put the Wi-Fi to sleep each 100ms [CBC+10] waking up just to check if the AP has any incoming data destined for the phone. If still no data has been sent by the server, then the radio will go back to sleep for another 100ms. As a consequence, the RTT can be too affected.

Finally, other versions simply disable all the connections when the battery reaches a certain minimum.

Since not all Android smart-phones offer the PSM option, different Android versions including it have different Wi-Fi connection behaviors, and this is a feature that the user can enable or not, no simple predictions can be done. Therefore, the described RTT obtainment of the OE was decided to remain as valid. Nevertheless, it is important to know the existence of this Android option and, if no interferences of the RTT due to the PSM are desired, this option should be disabled.

**The Instability of the 3G Connection:**

A typical 3G(UMTS) telecommunication technology switches between three states [GSS11]

- **IDLE**: The idle state. It consumes very little power. This is the default state when no network connection is needed.
- **DCH**: The Full-power state. It consumes notable battery energy. This state transmits data at around 7.1Mbits/sec to and from the device.
- **FACH**: The half-power state, an intermediate state were the power consumption is about the 50 percent of the DCH one. In this state, the device only transmits user data through shared low-speed channels that are typically less than 15kbps.

As shown in Figure 3.3, delays and tail times occur between each state transition.

![Figure 3.3: The three current states of a mobile device on a typical 3G network [GSS11].](image)

Changing from IDLE or FACH to DCH, i.e. incrementing the data transmission speed, adds a delay because control messages are exchanged between the device and the network. This delay,
which is around 1.5 or 2 seconds depending on the promotion, is noticeable by the user because the application is not responsive during this time. Once at DCH, the design is meant to avoid this state for too long because it consumes notable battery power and monopolizes channels that may be needed for other network users. Therefore, after 5 seconds of no data transfers, the state is changed to FACH. Finally, when no data is transferred at FACH during 12 seconds, the state is set back to IDLE. It can be noticed that this design shows a trade-off between

1. Long tail times that enlarge the consume battery power but enable quicker user input responses and
2. Shorter tail times that conserve battery power at the cost of slower response times.

As a consequence of the 3G design, precise data transmission speeds and RTTs are difficult to estimate in a precise way. When data is transmitted to obtain the bandwidth and RTT variables, no suppositions on occurred delays can be done, since other applications could have led the 3G state to be DCH.

Moreover, since ten small data transfers containing a short string are included in the server request for obtaining the RTT as an average of the obtained times and the package sent for obtaining the bandwidth is immediately after sent, it can be concluded that the values that the OE obtains are more close to the 3G DCH performance state. This leads to optimistic estimations which, in the worst case, could lead to time estimation errors reaching values around 1.5 seconds. Nevertheless, permitting the usage of 3G while assuming this risk is a better choice than not considering it, because for tasks such that the benefit of offloading is strong (e.g big data transfers are not required and notable execution times improvements are achieved), these penalties will still not damage the overall benefit to offload.

At Section 4.2.1, an application is executed to test how this issue can affect in a real scenario.

3.5.3 Android and server estimations of the task execution time

During this whole section, when referring to the estimated time of a task, it will also be assumed that this refers to a task with a given set of input parameters.

Three variables are used to calculate the estimated execution times of the tasks:

- $S_s$: A server speed in Instructions per millisecond
- $I$: A cost in number of instructions of a given task with a given set of input parameters.
- $S_f$: A speedup factor between the server and the mobile device. In other words, how many times the Android smart-phone is slower than the server.

Once this variables are determined, $E_{ts}$, the estimated execution time for a task to be resolved by the server, and $E_{ta}$, the estimated execution time for a task to be resolved by the Android device, can be obtained as:

$$E_{ts} = I \ast S_s$$  \hspace{1cm} (3.2)

$$E_{ta} = E_{ts} \ast S_f$$  \hspace{1cm} (3.3)

In the following sections, an explanation on how these variables are obtained is exposed.
3.5. **OBTAINING THE PARAMETERS INVOLVED IN THE OFFLOADING DECISION**

The server speed, $S_s$

As the actual system consists only of one server, this value is constant. $S_s$ has been determined through experimentation by executing on the server a Java byte code implementing a loop to later divide the number of Java byte code instruction by the time it took to solve the task. The server has been proved to suffer no overload and have a constant throughput. A different scenario is commented at Section 3.7.3.

The cost in number of instructions of a given task, $I$

The OE includes two options for obtaining the estimated number of instructions of a given task with a given set of input parameters:

- **Option 1:** The programmer provides a function capable to inform which is the cost in number of instructions of a task for a given set of input parameters. This can be very precise and effective if the function is well defined. However, sometimes this can be very difficult or impossible. In these cases, the second option can be used.

- **Option 2:** The automatic estimation cost system can be used. Firstly, the programmer provides a set of input parameters that are executed at the server. Using the $S_s$, the costs in number of instructions are determined and result in a database of inputs and costs. The web interface is used for this initial step to avoid energy and time consumption on the smart-phone. Secondly, this database must be copied at the Android application. Thirdly, each time that an estimated task cost needs to be determined, the most similar inputs are searched and weighted to obtain this value. Finally, once the task has been completed a new entry with the real execution values is added to the database or averaged with the existent ones. As a consequence, the more the application is used, the most precise the estimations are.

On the one hand, the idea to constantly monitor past executions in order to predict future ones is inspired in the MAUI system, where this method is said to be cost-effective because it is not prohibitively expensive and it provides acceptable estimations. On the other hand, the concept of running execution sets at the server side to provide a cost database that is of help for the offloading decision is inspired on the Clone-cloud partitioning mechanism, as this mechanism also runs different execution sets at the server to results in a database used for the offloading decision of each application part.

The automatic cost system has been implemented by a collaborator of the OE and is explained more in detail at [Gri13].
The speedup factor, $S_f$

The speedup factor $S_f$ between the smart-phone and the server. In order to understand the correct obtainment of this variable, previous Android OS features must be described:

**Different CPU frequencies and the Android governor:** Android devices run at different CPU frequencies. The Android governor is responsible of controlling how the CPU raises and lowers its frequencies [Xd12], in other words, the Android governor tells the CPU exactly what to do in a given situation. The governor orders are based in response to the demands that the user is placing in the device, i.e the computation that the active applications used by the user need. The importance of the governors relies on their impact to save energy consumption but also bring fluidity to the interface when it is needed. Different governors exist and can be selected depending on the smart-phone device [Xu13]

**Android Processes and processes’ resources:** An Android process by default has a single main thread and represents an application. Android processes are given more or less resources depending on their priority. It is not necessary to understand all the different priorities, but only that for reclaiming resources, the process must be at its maximum priority, the so-called critical priority. Figure 3.4 shows the priorities of the Android processes, more information about this can be found at [dev13b]. The important fact to extract is that for a process to have a critical priority,

![Android Processes Priorities](image)

Figure 3.4: Android processes priorities.

it must be active, which means that it must be hosting applications that are currently interacting with the user.

Once these concepts have been briefly described, it can be concluded that, for obtaining a precise speedup factor:

1. The CPU frequency, determined by the Android governor, must be working at the same speed that it would work if the task was executed locally.

2. The process containing the OE, which is actually the main process of the application to be enhanced by the OE, has the same resources as it would when the task was executed locally. Since the applications that the OE aims to improve are supposed to interact with the user, this priority must be the critical priority.
3.5. OBTAINING THE PARAMETERS INVOLVED IN THE OFFLOADING DECISION

The first time the application starts, the first speedup factor:

When the application using the OE starts for the first time, no speedup factor is still determined. The OE waits until a potential task to be offloaded arrives, which means that the application and its UI have already been started before and that the application is the current application interacting with the user. Therefore, it can be assumed that the process has had time enough to obtain the corresponding resources, i.e. the resources of a critical process.

At this point, a one million loop method is launched, forcing the smart-phone to compute and the governor to determine a CPU frequency that is representative for the execution of a computation expensive task. The cost in instructions $I$ of this method is known beforehand and, once the method ends, the overall required time in millisecond $T$ is obtained. This way, the Android speed $A_s$ in instructions per millisecond is determined to be:

$$A_s = \frac{I}{T} \quad (3.4)$$

As a result, being $S_s$ the server speed in instructions per second, the first speedup factor $S_{f\text{Loops}}$ is obtained as:

$$S_{f\text{Loops}} = \frac{S_s}{A_s} \quad (3.5)$$

Different speedup factors for different tasks:

Since not all the instructions require the same number or cycles to be executed, not all the instructions require the same time to be executed and, as a consequence, not all the speedup factors are the same for all the tasks. To solve this issue, each application part that is susceptible of being offloaded has its own speedup factor, $S_{f\text{Api}}$. Nevertheless, the $S_{f\text{Loops}}$ is used initially as an approximate speed factor for all the application parts $S_{f\text{Api}} = S_{f\text{Loops}}$. But, once a representative local execution of the application part is run, this speed factor value is updated with the corresponding one, which is also obtained following the same logic that was previously exposed for the $S_{f\text{Loops}}$ calculation.

A representative local execution:

The term representative local execution is used because, since different input parameters lead to different computation costs, not all the executions of a task will cause the Android governor to determine a high CPU frequency, thus not all the executions can be considered for the $S_{f\text{Api}}$ calculation. For instance, running few loops on the loop method would not be representative, because it is not cost-expensive. However, when running millions of loops, the method becomes cost-expensive, and it can be considered for the $S_{f\text{Api}}$ calculation. Through experimentation, a minimum task execution time $Min_{et}$ to solve this issue has been determined to be 20ms.

Adapting to possible smart-phone performance changes:

In order to consider possible performance reductions of the mobile phone due to an augment of installed applications that run at the same time or ageing in general, the $S_{f\text{Api}}$ are always calculated after a representative local execution. This could be considered as not very relevant but, since it can be done with no further cost, it was decided to be performed. The new $S_{f\text{Api}}$ are averaged with the old ones, and a maximum of 20 $S_{f\text{Api}}$ representing the last 20 representative executions is considered, i.e executions with execution times bigger than $Min_{et}$. Note that this also happens when the $S_{f\text{Api}}$ is refined for the first time, updating the approximate $S_{f\text{Loops}}$ value with the own.
Storing an retrieving the speedup factors:

The $S_f_{App}$ are stored before the application containing the OE is closed and are retrieved again when it is started. This way, the $S_f_{Loops}$ needs only to be calculated once, and all the $S_f_{App}$ are too stored. The Android Shared preferences [dev13a] have been used for this matter.
3.6 The Android side

On the Android side, the engine only requires some files to be included in the Android Application. The files are the following:

- **Engine.java**: This file is the core of the offloading engine. It contains all the functionalities needed to obtain the values of the relevant parameters that affect the offloading process, keep them updated, load and store the persistent ones, and decide in a given situation whether it is worth it or not to start an offloading process. This file shouldn’t be modified by the programmer who wants to use the engine in his application.

- **Algorithms.java**: Here is where all the potential offloading application parts are called. Their code can be also written here, but it is recommended that the application parts are compiled and packaged in a .jar file to be called by this file. If the programmer decides to provide a function to predict the cost in instructions of a task for a given set of input parameters, this must be written here. However, this is not obligatory and, when not written, the OE will use the automatic estimation cost.

- **DataBaseHelper.java**: This file is required for using the automatic estimation cost.
3.7 The server side

3.7.1 The communication with the server

The communication with the server is based on the basic query/response structure. In order to query the server (an Apache Tomcat web server), the serialized string containing the identification required data of the task to be offloaded is used to build an HTTP request. Once the task is completed by the server, the query is answered with XML formatted data, which is later parsed by the engine to obtain the result. Figure 1.1 shows a visual representation of the communication with the server.

![Communication with the server diagram]

Figure 3.5: The communication with the server.

3.7.2 The server files

Concerning the files at the server, the most relevant ones are the following:

- *Algorithms.java*: This file is very similar to the Algorithms.java file of the Android device. This file is responsible of calling the application parts to execute. The difference, though, is that here all the application parts from all the different Android applications that use the OE are called, whereas in the Android device only the ones that are contained in the smart-phone installed applications are. Another difference is that no functions to determine estimated execution times are present, as all the estimation process is held at the Android side.

- *RunAlgs.java*: The Java servlet that receives the queries and calls the corresponding algorithms in Algorithms.java.

- *UploadFile.java* and *UpdAndCompAlgs.java*: These files are used at the web-interface to upload to the server new application parts susceptible of being offloaded. These are .jar files containing the compiled codes of the application parts that are called by the Algorithms.java file when necessary.

- *GenerateDB.java* and *isDbReady.java*: These files are used at the web-interface for the automatic execution time estimation of the application parts that can be offloaded given a certain input.
The other files provide utilities for the implementation of the system or are not relevant for understanding the system.

### 3.7.3 The server overload

Although using a single server has been enough for the testing purposes of this thesis, an increase of popularity of the offloading system would lead to an excessive computation load for the server that would decrease its execution efficiency.

As a consequence, the whole system would be damaged, because the computation ability of the server, which is now treated as a static constant, is very relevant for the execution time estimations, hence in this scenario the offloading decisions would be determined erroneously.

However, the system has been design in a scalable way and would permit solving this situation with an enlargement to use multiple servers. This would solve the computation load if the cloud service could manage to offer a minimum execution time performance (nowadays, some cloud vendors can guarantee a minimum level of server speed performance [KLLB13]) and, at the same time, it would also be an interesting option because it would permit decreasing the connection RTT when being able to connect to servers that are closer to the mobile device. Moreover, using multiple servers would also permit to solve a server failure by re-executing the task to be offloaded in another server.

### 3.7.4 The problem of two different language virtual machines

Initially, the most relevant factor to use an Apache Tomcat web server was to benefit from the fact that its servlets are written with the Java programming language, like in the Android applications. The decision was taken to permit uploading the same application part compiled code for the smart-phones to the server, maintaining the privacy of the original source code and avoiding the programmer to write two different versions of the same application part. Although this is possible for some Android applications (like the ones used for testing the engine), differences between the Java Virtual Machine and the Dalvik VM entail the incompatibly of others.

On the first hand, Dalvik does not execute Java Byte-code. Instead it compiles Java classes into Dalvik executables (".dex") that use an alternative instruction set. For this reason, any application part using Dalvik libraries (also known as Android core libraries) will not be interpreted in a Java Virtual Machine, and so, it cannot be uploaded to the Apache Tomcat server.

On the other hand and as a remarkable problem, due to a research of the Android Applications that require intensive computing, it has been discovered that many of them use Android NDK, a tool to use native code on the applications that is recommended for self-contained, CPU-intensive operations that don’t allocate much memory [dev13a]. Application parts using native code or libraries that require it will, yet being apparently a good target for the OE, unfortunately also not be able to be uploaded to the Apache Tomcat Server. It is important to mention too, that some Android libraries designed for computational efficiency processing use previously compiled native code for the Android ARM Architecture (e.g. OpenCV), which makes it unfeasible to profit them in a regular server.

A good solution for this problem would be to virtualize the Android OS or the Dalvik VM in a computer that could still use its whole potential and resources.
3.8 The web interface

To avoid the costs of sending the application parts susceptible of being offloaded to the server as well as the costs of calculating the initial data base of the automatic cost system, an web interface was created: https://www.mi.fu-berlin.de/offload/

If the developers want to try the engine and test its performance with their applications, the web-interface must be used. In the appendix 2 of this thesis, a user guide to apply the OE in an Android application is exposed with all the necessary steps required for this matter.

3.8.1 A system to upload new application parts

Inspired on the Cuckoo and Macs offloading systems, the programmer must compile the application parts susceptible of being offloading and pack them in a .jar file that will be installed on the server. No source code is needed, only the class files.

The Java Archive (JAR) file is suitable for this purpose because the format bundles class files and auxiliary resources associated with applets and applications into a single archive file [Ora13].

Using the web-interface, the programmer must simply upload the .jar file and select the methods inside the .jar that must be uploaded. Then, an automatic process installs the new methods to the server (more concretely, the .jar is stored and the methods to offload are linked with the Algorithms.class java file). Once this is done, the OE installed on the Android device will already be able to offload the tasks.

This process is guided, describes the steps to follow, and uses different screens to inform about the progress done.

If a new method that was previously added is uploaded, then the new definition of the method will override the last one. A single user is considered now and a user/password identification is required.

The system, however, is meant to be able of being improved in a future, enabling the possibility to have more users. With this expansion, each user would manage its own application parts, which would be separated from the other user ones and would only be accessible by those knowing the user identification.

3.8.2 The automatic cost estimation

Another labor of the web-interface is to run an initial process step for the system to automatically estimate execution times of the new uploaded application parts depending on their input parameters. For this task, the user must supply a file containing execution sets for the methods. As a result, the database containing the related instruction cost for each of the sets is returned. This file must then be added to the Android application using the OE. Like the method upload process, this process is also guided and uses different screens to inform about the progress done. More detailed information about the automatic cost estimation functioning can be found at [Gri13].
Chapter 4

Testing

4.1 Hardware

Two mobile devices have been used for the testing of the offloading engine:

- **Samsung Galaxy Young (832 MHz).**
- **Samsung Galaxy Nexus (1.2 GHz Dual Core).**

The CPU frequencies are the maximum frequencies that the mobile devices can have.

The used server has the following technical details:

- **Processor:** 4 cores, Intel Xeon CPU E5649 2.53 GHz.
- **Main memory:** 7786 MB.
- **Java version:** 1.6.
- **Apache Tomcat 6.**

The server was located 8km away from the smart-phones during the testing.
4.2 The engine testing application

This application was developed in order to show the general functioning of the OE and perform some basic testing. This is a simple application with limitations like parameter type and limited number of parameters that the OE does not have. The application, as an installable Android .apk file, can be found at https://www.mi.fu-berlin.de/offload/downloads

Figure 4.1 shows an overview of the different layouts of the application:

![Figure 4.1: The engine testing overview.](image)

**The main screen:** Situated in the middle, the main screen is used to enter the input parameters of the methods, select a method to execute, or check the engine parameters that are being used for the offloading decision.

**The execution data screen:** The screen on the left is the resulting screen of executing a method from the main screen. The estimated execution times and the real execution times are shown for both the local and the offloading execution. The server execution time for solving the task is also shown, although this time is included in the offloading execution time. As a reminder, note that the offloading engine will take the decisions to offload or not depending on which estimated option has the lowest execution time.

**The engine parameters:** This screen shows all the relevant parameters for the engine to decide if a task should be offloaded or not. The different speed factors for each of the methods are shown, as well as the RTT, the bandwidth, and the network and server availability information.

Although several methods can be run in this application, only two will be used for this testing:

- **Do some loops:** This method takes the first parameter as input and performs as many loops as indicated. When the method finishes, the output "Done" is resulted. This algorithm has a $O(n)$ complexity.

- **Sort array:** This method takes the first parameter as input. First, it creates an array of doubles of the size defined by the input parameter and then, it fills it with random values. Finally, it sorts the array using the selection sort algorithm and the output "Done" is resulted when the method finishes. This algorithm has a $O(n^2)$ complexity.
4.2. THE ENGINE TESTING APPLICATION

Functions to estimate the cost were written for both methods, hence the automatic cost estimation system was not used in the testing of this section.
4.2.1 Comparing 3G and Wi-Fi connections

In this testing, the 3G and Wi-Fi connection performances are compared. As described previously in Section 3.5.3, it is known that 3G connections suffer delays depending on the elapsed time interval between the last data transmission sent and the new one.

This testing was done with the Android Galaxy Young device by executing the *Do some loops* method with different iterations to resolve. In order to test the connection differences between Wi-Fi and 3G, high numbers of iterations were introduced as input parameters to force the OE to decide offloading. As a remarkable fact, to test the 3G performance all the executions were performed in a short period of time except for three of them (60, 84 and 108 millions of iterations), where some time was waited. The RTTs determined by the OE for each connection were 352 ms for the 3G and 80 ms for the Wi-Fi.

Figure 4.2 shows the Wi-Fi results and Figure 4.3 the 3G ones. Two charts are included for each Figure. The upper charts indicate the execution times of:

- **Local execution**: The local execution times for solving the task when no internet connection is available.
- **Offloading estimation**: The estimated by the OE times for the whole process regarding the offloading of the task.
- **Offloading real**: The real times that the whole process to offload the task took.

The charts below show a more detailed view of the real offloading executions, separating between the time that was spent for the communication and the time that the server required for solving the task.
4.2. THE ENGINE TESTING APPLICATION

Figure 4.2: The Wi-fi communication time results.

It is clear, that the performance differences are very significant. Wi-Fi is very stable whereas 3G, on the contrary, is very unstable and confirms the expected delays.

The following conclusions when using 3G connection can be deduced:

- If the application using the OE communicates often with the server, the predicted RTT will be valid. Otherwise and as studied previously, penalizations up to 2 seconds can happen.

- If the application using the OE does not always communicate often with the server but the benefit due to offloading is very high, it is still worth it to offload. Although the predictions could be wrong, if the application tends to save several seconds due to offloading considering 3G for offloading will still be positive and improve the overall execution time.

Moreover, other applications from the mobile device could be also communicating through 3G, avoiding the delays on the application where the OE is included.
The OE was determined to not discriminate between 3G and Wi-Fi connections for the offloading decisions because the fact that 3G RTTs are much higher than the Wi-Fi RTTs (about 4 times more) was considered to be a sufficiently remarkable factor that affects the offloading decision. However, a future improvement could let the programmer add extra information to be considered by OE when the established connection was 3G. For instance and when connected to a 3G network, the programmer could decide if his application will benefit from offloading by answering the following questions:

1. Will the application communicate often with the server?

2. Do the potential tasks to offload often benefit in a notable way (in the order of seconds) from offloading?
4.2. THE ENGINE TESTING APPLICATION

4.2.2 Executing the methods

Figure 4.4 and Figure 4.5 show, for each smart-phone, the estimated and the real execution times of each method. For clarity, the local execution and estimated values that would happen if no offloading was performed have not been excessively enlarged once they intersect with the offloading ones. The lines on the charts have the same meaning as the one explained in the previous testing. As a reminder, note that the OE will always execute the option with the lowest estimation.

Figure 4.4: The execution times of the loop method.

Figure 4.5: The execution times of the Sort array method.
Several observations can be deduced from this charts:

1. The server execution time to resolve a task increments very slowly compared to the smartphones’ one. Therefore, the offloading times have all very similar values, resulting in an almost horizontal line.

2. As a consequence of 1, the most significant factor for the offloading decision is the RTT, as it is much bigger than the server execution time.

3. As a consequence of 2, even if the server was infinitely fast, offloading cannot improve performance for tasks such that can be resolved in less than the RTT time.

4. Since the Galaxy Young device is slower, it also offloads tasks earlier, which makes sense.

5. In general, the OE takes the correct decision and, even if the estimations are not very precise (like in the sort array case) in overall, offloading will save time.
4.3 The chess game

An Android Chess game was decided to be adapted in order to use the OE. A chess game is an excellent candidate for offloading since the cost for the game to choose the next move can be very high and the only data to be transmitted is a representation of the piece positions. In fact, the potential benefit of using offloading techniques for improving a chess game is mentioned at [LT11], [DLNW11], and [CBC+10].

The application, as an installable Android .apk file, can also be found at https://www.mi.fu-berlin.de/offload/downloads

Figure 4.6 shows an overview of the different layouts of the application and game options.

The different screens are explained as it follows:

**Game:** The game screen, situated in the middle, is the interface used when playing to the chess game. The user must only drag and drop the white pieces, and the artificial intelligence (AI) of the game moves the black pieces. When pressing the setting button of the Android device, a tab will appear showing different options to choose.

**Last execution data:** Once a chess move has been done, this screen can be open through the options tab of the Game screen. The estimated execution times and the real execution times are shown for both the local and the offloading execution. The server execution time for solving the task is also shown, although this time is included in the offloading execution time. As a reminder, note that the offloading engine will take the decisions choosing the execution time with the lowest estimated value.
**The engine parameters:** This screen shows all the relevant parameters for the engine to decide if a task should be offloaded or not: the speed factor, the RTT, and the connection availability and type. The bandwidth is not shown because no big data transfers are done, but only small strings containing the chess board state.

**New game options:** Each time the user selects to start a new game at the main screen, three different difficulties are offered: easy, normal and hard.

**Different difficulties lead to different execution times:**

The application part that is in charge of determining the next move of the AI performs the following steps:

Firstly, it obtains all the possible moves given the board situation.

Secondly, for each possible move, an evaluation is given. The evaluation is an integer determining how good or bad the movement is for the black pieces.

For obtaining the evaluation, a number of future moves $n$ is considered. The moves are implemented in a tree structure and each node of the tree is also given an evaluation. The future moves $n$ is very relevant to determine the evaluations:

- Before $n$ is reached, all the possible move sequences starting from the current state are given an evaluation. For efficiency, the alpha-beta pruning algorithm is implemented [Wik13a]. This algorithm prunes a tree part if at least one possibility has been found that proves all the next moves of the tree node to be worse than a previously examined one.
- If the number of moves $n$ is reached, a coarse evaluation is given to the tree node. This evaluation has a fast execution time cost.

The number of next moves is actually determined by the difficulty level of the game. Higher difficulty levels have higher $n$ values, because this leads to better evaluations and to better choices of the AI when moving the black pieces.

Finally, the moves are sorted depending on their evaluation and one of the best moves is taken. More in detail, if there is a clearly best move it will be chosen, but if there are more, one of them will be chosen randomly. This is performed to provide variety to the chess games, since even when the user takes the same moves, the AI will not.

Relevant observations can be taken from this process. On the first hand, higher difficulty levels should lead to longer execution times for the AI, because higher values of $n$ will lead to more next movement predictions. On the other hand, the alpha beta algorithm will prune the tree depending on the pieces positions in a concrete moment of the game. As a consequence, no previously estimations can be done in an easy and non costly way.

Therefore, the chess game was considered to be a very interesting scenario to test the Automatic cost estimation system. The parameters provided to the automatic system were 1) The difficulty of the game and 2) An integer representing the board state, i.e. the positions of the pieces in a concrete moment of the game.

As it is explained in [Gri13], to calculate the estimated costs, the automatic cost estimation system needs an integer identifier for each set of execution parameters. This identifier is used to obtain its related cost from the database if the entry exists and to identify similar execution sets. For the chess game to use this system, the identifiers were performed by combining 1) and 2) into a single
4.3. THE CHESS GAME

integer and precisely in this order so that the system could detect games with the same difficulty.

The initial database of the game, generated by the web-interface, had 5 games played on each
difficulty level. Then, the game was installed on both Android devices and different users played
the game until 20 more games had been played for each difficulty level.

Five different tests for each difficulty level were performed with 20 AI moves and will be described
as it follows. The rate of successful decisions of the OE, the execution time saved or lost due to
correct or wrong offloading decisions, and the overall time saved because of the inclusion of the
OE in the chess game application are shown as it follows for each game difficulty. The conclusions
of all the tests can be found at the end of this section.
4.3.1 Easy difficulty

The easy difficult mode is meant for people who are learning to play chess. The AI is limited to simply move the pieces in a correct way and avoid the king piece to be killed if this is possible. This leads to very low computation costs and at the same time, to very irregular behavior of the AI. As a consequence, the results of the tests indicated that it was never worth to offload.

Due to the difficulty of playing two games with the same pieces moves, Figure 4.7 shows the estimated and real time execution values of a single game in easy mode. It must be taken into account, though, that if the five games would have been averaged, the real local execution time line would have been smoother, but always under the offloading time line.

Figure 4.7 shows the execution times of the Galaxy Nexus on the left and the Galaxy Young on the right.

![Figure 4.7: Easy game difficulty execution times](image)

The estimated offloading and real offloading values nearly match, impeding the clear visibility of the offloading estimation lines.

Since the OE will offload or not depending on which option has the lowest estimated execution value, the following information can be deduced:

**Galaxy Young and Galaxy Nexus**

Rate of successful decisions: 100%
Execution time saved due to correct offloading decisions: 0 ms
Execution time lost due to wrong offloading decisions: 0 ms
Overall time saved: 0 ms
4.3. THE CHESS GAME

4.3.2 Normal difficulty

The normal difficulty mode is meant for people who have played several times to chess games. The AI in this mode is capable of choosing good moves, offering a challenge to the user.

Five games with the same game movements were averaged in order to obtain the data for Figure 4.8. The sequence of movements that were performed is shown at the end of this section, at the Table 4.10.

As previously commented, the chess game does not respond necessarily always with the same moves, therefore, only games where the sequence was as described in the 4.10 Table were considered for achieving the data.

Since the automatic cost estimation system is based on previous estimations, the monitoring of the games moves while performing the tests was disabled. Otherwise, as all the moves are the same, the predictions would have been very precise and not realistic.

![Figure 4.8: Medium game difficulty execution times(GN left and GY right)](image_url)

Since the OE will offload or not depending on which option has the lowest estimated execution value, the following information can be determined:

**Galaxy Nexus**

Rate of successful decisions: 85%
Execution time saved due to correct offloading decisions: 14322.4 ms
Execution time lost due to wrong offloading decisions: 45.2 ms
Overall time saved: 14277.2 ms

**Galaxy Young**

Rate of successful decisions: 95%
Execution time saved due to correct offloading decisions: 52070.9 ms
Execution time lost due to wrong offloading decisions: 28.1 ms
Overall time saved: 52042.8 ms
4.3.3 Hard difficulty

The hard difficulty mode is meant for chess game professional players. The AI in this mode is capable of choosing nearly perfect moves. Five games with the same game movements were averaged in order to obtain the data for Figure 4.9. The sequence of movements that were performed is shown at the end of this section, at the Table 4.10.

The monitoring of the games moves by the automatic estimation cost system was too disabled for performing the tests of this difficulty.

Figure 4.9: Hard game difficulty execution times (GN left and GY right)

**Galaxy Nexus**

Rate of successful decisions: 100%
Execution time saved due to correct offloading decisions: 24016.9 ms
Execution time lost due to wrong offloading decisions: 0 ms
Overall time saved: 24016.9 ms

**Galaxy Young**

Rate of successful decisions: 100%
Execution time saved due to correct offloading decisions: 76613.3 ms
Execution time lost due to wrong offloading decisions: 0 ms
Overall time saved: 76613.3 ms
4.3. THE CHESS GAME

4.3.4 Evaluation

The overall evaluation is positive. Although the estimated values are not precise, twenty five previous monitored games result to be enough for the automatic cost estimation system to obtain execution time estimations that lead to a high correct decisions rate: a rate of 96.6%. This way, the OE is capable of taking different correct decisions depending on the input parameters, which in this case, are the game difficulties and the board state.

More in detail and referring to the automatic cost estimation system, it can be observed from the normal and hard difficulty figures Figure 4.8 and Figure 4.9, that the estimations tend to be good at the beginning of the game and get worse as the game advances. This is due to the fact that the first moves of a chess game tend to be the same, which causes the system to have accurate values because the board states were already executed in the past and monitored.

On the other side, once the game moves increase, the automatic cost estimation has less probabilities of finding the executions sets in its database. As a consequence, the system estimates the cost by searching other board states with the same difficulty and averages them. These estimations are not precise, but since the average of the medium and hard difficulty modes is to offload, and for easy difficulty mode to not offload, the outcome tends to be mostly successful. Even more, when the decision is wrong, the time penalties are very low compared to the times savings of the correct ones.

Moreover, a tendency that can be observed at the medium and hard difficulty modes is that, as the number of pieces decreases, the execution times also decrease. This is probably consequence of a decrease of possibilities to evaluate by the tree of moves. If this was the case, adding the number of remaining pieces to the integer identifier of the execution set would improve the estimation calculations once the game is in an advanced state.

To conclude, the benefits for offloading a chess game with a minimal artificial intelligence are very high. The Android Galaxy Nexus, a high-end smartphone, can save up to 24 seconds within twenty moves. For its part, the Galaxy Young, a medium-end smartphone, can have an even more remarkable execution time saving of 76 seconds. For a regular player of the chess game, this would save big amounts of time in a whole day. A side effect of this time savings is that, for both devices, the game fluency and overall experience is really improved when using the OE, as the user does not notice the time required for the computation of the next move.
**CHAPTER 4. TESTING**

Game moves for hard and normal game difficulties

The game moves are described with the algebraic notation [Wik13b]

![Figure 4.10: Game moves for hard and normal game difficulties](image)

<table>
<thead>
<tr>
<th>Move</th>
<th>Player (White)</th>
<th>IA (Black)</th>
<th>Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e2, e1</td>
<td>d7, d5</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>d2, d3</td>
<td>e7, e6</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>d1, h5</td>
<td>g8, f6</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>h5, g5</td>
<td>d5, e4</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>d3, e1</td>
<td>f6, e1</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>g5, d6</td>
<td>e8, d8</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>f2, f5</td>
<td>e1, f6</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>c1, g5</td>
<td>b7, h6</td>
<td>27</td>
</tr>
<tr>
<td>9</td>
<td>g5, h4</td>
<td>b8, e6</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>b1, a3</td>
<td>f8, b4</td>
<td>27</td>
</tr>
<tr>
<td>11</td>
<td>e2, e3</td>
<td>b4, d6</td>
<td>27</td>
</tr>
<tr>
<td>12</td>
<td>h4, f6</td>
<td>g7, f6</td>
<td>25</td>
</tr>
<tr>
<td>13</td>
<td>a1, a4</td>
<td>d8, d7</td>
<td>25</td>
</tr>
<tr>
<td>14</td>
<td>g1, e2</td>
<td>b8, d8</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>f3, f1</td>
<td>e7, f8</td>
<td>25</td>
</tr>
<tr>
<td>16</td>
<td>g2, g3</td>
<td>d6, a3</td>
<td>24</td>
</tr>
<tr>
<td>17</td>
<td>h2, a3</td>
<td>d8, d1</td>
<td>22</td>
</tr>
<tr>
<td>18</td>
<td>e1, d1</td>
<td>f8, g8</td>
<td>21</td>
</tr>
<tr>
<td>19</td>
<td>e2, d4</td>
<td>c5, d7</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td>d4, e6</td>
<td>d7, e6</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Move</th>
<th>Player (White)</th>
<th>IA (Black)</th>
<th>Pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e2, e1</td>
<td>d7, d5</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>d2, d3</td>
<td>e7, E5</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>d1, h5</td>
<td>b8, c6</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>c1, g5</td>
<td>g8, f6</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>G1, F3</td>
<td>F6, H5</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>G2, G1</td>
<td>H5, F6</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>B1, G5</td>
<td>D5, E4</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>C3, D5</td>
<td>D8, D5</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>C2, e4</td>
<td>F8, B4</td>
<td>29</td>
</tr>
<tr>
<td>10</td>
<td>F3, D2</td>
<td>D5, D4</td>
<td>29</td>
</tr>
<tr>
<td>11</td>
<td>a2, a3</td>
<td>D4, B2</td>
<td>28</td>
</tr>
<tr>
<td>12</td>
<td>A1, b1</td>
<td>B2, B1</td>
<td>27</td>
</tr>
<tr>
<td>13</td>
<td>E1, E2</td>
<td>B1, D3</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>E2, E1</td>
<td>D3, A3</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>F1, H3</td>
<td>A3, H3</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>G5, H4</td>
<td>H3, H4</td>
<td>23</td>
</tr>
<tr>
<td>17</td>
<td>E1, E2</td>
<td>B4, D2</td>
<td>22</td>
</tr>
<tr>
<td>18</td>
<td>F2, F1</td>
<td>H8, F8</td>
<td>22</td>
</tr>
<tr>
<td>19</td>
<td>H1, G1</td>
<td>H1, H2</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td>G1, G2</td>
<td>H2, F1</td>
<td>21</td>
</tr>
</tbody>
</table>

Figure 4.10: Game moves for hard and normal game difficulties
Chapter 5

Conclusions

Computation offloading techniques can save significant amounts of time and energy in nowadays smart-phones. Several implemented offloading systems have tested and made evidence of this fact, as well as the offloading system developed in this thesis, which has proved to be able of saving up to 72 seconds in a mid range Android smart-phone when determining the 20 next moves of a chess game.

However, the decision to always offload a task is not always advantageous, since this can sometimes harm the overall performance of the application. Network conditions, computation speeds of both the smart-phone device and the surrogate, and the total amount of instructions that the task will execute depending on its input parameters, are factors that must be taken into account for the offloading decision, since they directly affect the local and remote execution times of the task, causing one or another option to be preferable.

Dynamic offloading decisions are presented as a solution to this issue due to their ability of adapting to these different run-time conditions. Nevertheless, offloading papers as well as the development and testing of this thesis’s offloading system show difficulties and challenges on obtaining and predicting them.

Firstly, smart-phones run at different CPU frequencies in order to save battery consumption, thus, the computation speed of the mobile device must be determined at a precise moment where it is assured that the mobile device is running at its maximum speed.

Secondly, due to the mobile nature of the smart-phones, network connection quality is constantly changing and network failures happen often. This causes the offloaded tasks to not be completed. Local re-execution is the most cost-effective and general solution for solving this issue, since other solutions are too expensive for the smart-phone environment or not suitable for tasks with variable execution times.

Thirdly, another instability factor when predicting data transfer times is that 3G connections can delay the communication up to two seconds. This time is sufficiently significant to be taken into account by the offloading system, since depending on the character of the potential tasks to offload, only when a Wi-Fi connection is established the offloading option should be considered.
The last difficulty and challenge when referring to the factors that affect the dynamic decision is the prediction of the number of instructions of a task given a determined input. In the offloading system of this thesis, two options are offered: to write a function that determines the number of instructions from the input parameters, or to use an automatic cost estimation system based on past executions of the task.

Finally, it has been discovered that the code structure of the application and the contained methods in each application parts are very relevant if it is desired to obtain the maximum benefit of the computation offloading techniques. For instance, if a user interaction is contained in a computation expensive method, the method will not be able to be offloaded. If the aim is clearly to save energy consumption and execution time, bringing the programmer to be aware of these facts and to follow some patterns is a good criterion, since once the programmer is conscious of the benefits of the computation offloading techniques, he or she will also be satisfied.
5.1. The offloading engine further work

The overall results of the offloading engine have been positive, however, there is still space to improve.

In order to permit the OE to offload application parts using native code or libraries that require it, either a virtualization of the Dalvik VM should be done in the server, or the programmer should write another version compatible with the server architecture. This is considered the most important future work, since several computation expensive Android applications contain native code.

No big data transfers are considered to be pulled. In order to add this functionality, a new parameter for the offloading engine decision containing the size of the data to receive could be added. As an alternative option, a rate between the data sent and the data that will be received could also be included. For instance, this would be interesting when sending images to the server that have to be fetched back again with the same size or a bigger or smaller size.

To upload new application parts to the server, the OE uses a web-interface. This way, the costs of sending the application parts susceptible of being offloaded are avoided. However, if the system was expanded with more servers, a similar system as the proposed by the Cuckoo framework could be performed: in case that the surrogate is available but the application part is not installed, the smart-phone sends the .jar file containing the compiled classes of the task. In this case, the automatic cost estimation would not have an initial database, but would rather build it monitoring the past executions directly from the mobile device and having a learning phase that is now not required.

Finally, the impossibility of measuring the energy consumption of an Android device in a fine-grained way with the Android API has led to a saving time approach of the offloading engine (OE) and to determine a study on how saving execution times affect the energy consumption. The Android coarse-grained energy measurement of Android is a problem that other offloading systems such as MAUI have also faced. Exposing it to the API providers, in addition to the energy savings that can be achieved through offloading could be considered, since nowadays energy savings in mobile devices is a studied topic.
Chapter 6

Appendix A: Saving time means saving energy?

In this appendix, the energy that a smart-phone consumes as a consequence of saving time due to computation offloading techniques is exposed.

As exposed during the factors involved in a dynamic decision (Section 2.1.2) a saving time approach or a saving energy approach can be considered for the offloading decision.

Being $X_{local}$ the speed of the local execution in instructions per time unit, $X_{server}$ the speed of the remote execution in instructions per time unit, $D_s$ and $D_r$ the data to be sent and received respectively, $B_s$ and $B_r$ the sending and reception bandwidths, $P_{le}$ the power consumed for the local execution, $P_i$ the power consumed when the device is idle, and $P_s$ and $P_r$ the power consumed for sending and receiving the data respectively, offloading will save energy if it is performed when:

$$\frac{P_{le} \ast I}{X_{local}} - \frac{P_i \ast I}{X_{server}} - \frac{P_s \ast D_s}{B_s} - \frac{P_r \ast D_r}{B_r} > 0 \quad (6.1)$$

To study how saving time affects the energy consumption, two scenarios will be considered:

1. Scenario 1, a general scenario with notable computation where nearly no data needs to be transmitted.

2. Scenario 2, a notable computation with variable data transfers scenario.
Scenario 1

It is considered that the data to transmit is not significant and that the computation is notable. A RTT for transferring a minimum data for the communication and the corresponding energy that this requires would also happen but, since a notable computation is assumed, these values are too small to be significant.

Therefore, in this scenario offloading will save energy if performed when:

\[ \frac{P_{lc} \times I}{X_{local}} - \frac{P_i \times I}{X_{server}} > 0 \]  

(6.2)

Since it is known that \( F > 1 \) and that \( P_i < P_c \), in this scenario it can be considered that saving time means saving energy.

Scenario 2

In this scenario, a computation expensive task is assumed. The aim is to discover how a computation expensive task changes from being enhanced due to offloading to the contrary depending on the data amount that needs to be transferred.

The following variables are defined: \( C \) as the number of instructions to execute, \( F \) as the speed relation between the surrogate and the mobile device, \( S \) as the server speed in instructions per millisecond, \( M \) as the mobile device’s speed in instructions per millisecond, and \( B \) as the bandwidth in bytes per millisecond. The server speed \( S \) is considered to be the speed of the server used in this thesis, and the other values are obtained from executing the loop method described in Section 4.2 for a 20 million input using a Wi-Fi connection. This method is selected because it is known from its resulting Java Byte-code that it performs 5 instructions per loop. Therefore, the values of these variables are determined as:

\( C = 100 \times 10^6 \) instructions  
\( F = 33.6 \)  
\( S = 66666666 \) instructions per ms  
\( M = S/F = 148.41 \) instructions per ms  
\( B = 500 \) bytes/ms

The following variables are also defined: \( P_c \) as the power in Watts consumed when executing locally, \( P_i \) as the power in Watts consumed when the device is idle, and \( P_{tr} \) as the power in Watts consumed when sending data with a Wi-Fi connection. The values of these variables are from the MACS offloading system [KYK12], and are representative for the power consumption that an actual smart-phone consumes.

\( P_i = 0.05 \) Watts  
\( P_c = 0.4 \) Watts  
\( P_{tr} = 0.7 \) Watts

To complete this scenario, the data \( D \) to be transferred is considered variable.
The Figure 6.1 shows how the execution times vary when the task is offloaded and when the task is executed locally according to the amount of data $D$ that the task needs for being executed and that, as a consequence, needs to be sent to the surrogate. From its part, Figure 6.2 shows the same from an energy consumption approach.

![Figure 6.1: Execution times when increasing the amount of data to send](chart1)

It can be seen, that the two criteria would agree in their decisions to execute a task locally or remotely around the 80% of the times.

Although the considered values are not absolutely precise, interesting conclusions can be taken from both situations:

1. When the data to send is small and the computation is intensive, both criteria will agree
2. When the time criterion concludes a clear benefit due to offloading or, on the contrary, to a clear disadvantage compared to the local execution, the energy criterion concludes this too. In other words, for the cases where it is clear that local execution or offloading are the best option, both criteria are in agreement.
3. There is a threshold where each criteria concludes different results. However, the time or energy losses due to a contrary decision are less significant than the time or energy savings of the cases where both criteria are in agreement.
An important observation can be taken from these conclusions. If the application part to offload has a significant range of executions with high computation costs, these will be often offloaded and the criterion that is used will not be relevant because, in overall, both time and energy will be saved. In other words, the benefit of the cases where offloading will save time or energy will be superior to the time or energy losses that the discordance between both criteria causes.

In fact, this observation is supported by the CloneCloud paper, where it is stated that energy consumption mostly follows execution time and that energy consumption is proportional to the time that the task’s response takes to be resolved [CIM+11]
Chapter 7

Appendix B: How to use the Offloading Engine in an Android application

The steps to follow when willing to use the offloading engine are here described: All the necessary files as well as performed examples can be found at https://www.mi.fu-berlin.de/offload/downloads

1. Copy the Algorithms.java, DataBaseHelper.java and Engine.java files to you application Project.

2. Separate the computation expensive code from the other code of the application. They must be in different java files.

3. For each application part, write the name of the main method that calls the whole task to offload at the AlgName enumeration at the algorithms.java file.

4. Write the cases of each of the enumerations written at 3) at Algorithms.java. For 3) and 4) a finished example of the doSomeLoops method is already done, it can be followed.

5. In case you want to add a function cost for your application parts, write it at the getCost function of the Algorithms.java file. Otherwise, ignore this step.

6. It is now possible to call the engine from the Android application file. The engine creator function must be added (a), and then the engine can be called (b)
   a. offloadingEngine = new Engine(this.getApplicationContext());
   b. String result = offloadingEngine.execute(nameOfTheTask, parameter1, parameter2, parameterN);

   Note that the name of the tasks should be the same as the ones written in 3).

7. The resulting compiled files of the computation expensive task must be uploaded to the server. Use the web interface for this.

8. The application with the offloading engine is now ready to run.
Bibliography


BIBLIOGRAPHY


