A Solution to Profiling for Mobile Computation Cloud Offloading

by

Corey McGrillis

Bachelor of Science, Ryerson University, 2013

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

A Solution to Profiling for Mobile Computation Cloud Offloading

Master of Science 2015

Corey McGrillis

Computer Science

Ryerson University

Abstract

In this thesis, we develop a Generic Profiler which is a solution to profiling for mobile computation cloud offloading. Profiling refers to the process of examining and collecting statistics and information about data. The Generic Profiler uses reflection in many cases to reduce redundant code and allow developers to easily incorporate new profilers to their offloading solutions. Third party developers can include the Generic Profiler into their solutions with ease and without having to worry about the underlying processes of storing data using a Content Provider. The Generic Profiler also contains a set of default profilers, which include a Location Profiler, Software Profiler, Wi-Fi Profiler, Telephony Profiler, and a Battery Profiler. Using the Software Profiler, we have shown that the performance, in terms of inclusive invocation time, of the profilers which are designed by implementing the Generic Profiler adequately store data.

Acknowledgements

I wish to express my sincere thanks to my supervisor, Dr. Jelena Misic, whom graciously accepting me as her Masters student in Computer Science. She has supported me throughout the entire process and has offered much of her experience and resources for the research. I would also like to thank my family and friends who have all been extremely patient and supportive of my research.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Problem Statement 1

1.3 Approach 2

1.4 Thesis Contribution 2

1.5 Thesis Organization 3

2 Related Work **5**

2.1 Service Based Cloud 6

2.1.1 Software as a Service (SaaS) 6

2.1.2 Platform as a Service (PaaS) 6

2.1.3 Infrastructure as a Service (IaaS) 7

2.2 Mobile Cloud Offloading 7

2.2.1 General Architecture 8

2.2.2 Remote Connection 8

2.2.3 Code Offloading 9

2.2.4 Policies 10

2.3 Profiling 10

2.3.1 Hardware Profiling . . . 11
2.3.2 Software Profiling . . . 12
2.3.3 Network Profiling . . . 12
2.3.4 Context Profiling . . . 13
2.3.5 Preference Profiling . . . 13
2.3.6 Environmental Profiling . . . 13
2.3.7 Android Content Provider . . . 14
2.4 Estimations . . . 16
2.4.1 Naive History-Based Approach Using an Exponential Moving Average (EMA) . . . 17
2.5 Energy Optimization . . . 19
2.5.1 Context-Aware Energy Model . . . 21
2.5.2 Hardware Throttling Energy Model . . . 21
2.5.3 Modeling Energy Consumption . . . 22
2.6 Performance Optimization . . . 23
2.7 Tradeoff Decisions . . . 23
2.8 Challenges . . . 24
2.8.1 Offloadable Code Discovery . . . 24
2.8.2 Computation Offloading Decision . . . 26
2.8.3 Cloud Discovery . . . 27
2.8.4 Task Scheduling . . . 28
2.8.5 Data Synchronization . . . 29

3 Methodologies **33**

3.1 Introduction . . . 33

3.2 Generic Profiler 34
3.2.1 Generic Content Provider 34
3.2.2 Generic Contract Class 36
3.2.3 Generic Database Helper 39
3.2.4 Table Builder 39
3.2.5 Content Helper 41
3.3 Profilers 44
3.3.1 Location Profiling 46
3.3.2 Software Profiler 47
3.3.3 Telephony Profiler 48
3.3.4 Wi-Fi Profiler 50
3.3.5 Battery Profiler 52
3.4 Summary 54

4 Evaluation **55**

4.1 Introduction 55
4.2 Development Effort 56
4.2.1 Content Providers 56
4.2.2 Contract Classes 57
4.2.3 Fully Implemented Provider 58
4.2.4 Full Implementation of the Generic Profiler 59
4.3 Performance 60
4.3.1 NQueens Algorithm Evaluation 60
4.3.2 Profiler Evaluation 63
4.4 Summary 67

5 Conclusion and Future Work **69**

5.1 Contributions 69

5.2 Future Work 70

References **71**

List of Figures

2.1 Overview of a Content Provider 15

2.2 General architecture of a cloud-assisted computation offloading system to support mobile services [1] 22

3.1 Example demonstrating how the Table Builder generates both parent and child SQLite tables and applies according foreign keys to the child table. 41

3.2 Overview of the Content Helper. 41

3.3 Inserting, updating, or deleting of a Java Object using the Content Helper. 42

3.4 Query the Content Helper using a Java class type as input. 43

3.5 Overview of the Profiling Content Provider 44

3.6 Overview of the Location Profiler 46

3.7 Overview of the Software Profiler 47

3.8 Overview of the Telephony Profiler 49

3.9 Overview of the Wi-Fi Profiler 51

3.10 Overview of the Battery Profiler 53

4.1 Comparison between lines of code needed to implement the provided Generic Content Provider vs. the Content Provider. 57

4.2 Estimation of lines of code needed to write Contract Classes for each of the provided Profilers. 58
4.3 Comparison between lines of code needed to fully implement a Content Provider solution vs. the provided Generic Profiler solution. 59
4.4 Lines of code written to fully build the Generic Profiler 60
4.5 Average inclusive invocation time of the NQueens algorithm for 0 to 10 queens 61
4.6 Average inclusive invocation time of the NQueens algorithm for 11 to 13 queens 62
4.7 Average inclusive invocation time of the NQueens algorithm for 14 to 15 queens 63
4.8 Average inclusive invocation time of the Software Profiler for the NQueens algorithm(One method at a time) 64
4.9 Average inclusive invocation time of the profilers 64
4.10 Invocation count of the profilers 65
4.11 Average inclusive runtime up to 1,200ns of methods collected during 30-minute profile 66
4.12 Average inclusive runtime up to 12,000ns of methods collected during 30-minute profile 66
4.13 Average inclusive runtime up to 60,000ns of methods collected during 30-minute profile 67
4.14 Average inclusive runtime up to 14,000,000ns of methods collected during 30-minute profile 68

Chapter 1

Introduction

1.1 Background

Cloud offloading refers to the process of sending computation requests to a cloud server and waiting for a response. This process allows the client to use the cloud's vast resources to get results much quicker than normal. For example, if a computation normally takes a few minutes to complete locally on a mobile device, an offloading solution may choose to offload it to a cloud computing center instead for improved performance.

1.2 Problem Statement

In mobile computation cloud offloading profiling data is a necessity. Profiling refers to the process of examining and collecting statistics and information about data. The purpose of profiling is to collect data and then further determine whether or not the data can be used for a particular situation. Profiling in mobile computation cloud offloading includes the process of collecting continuously changing information about the mobile

device and the surroundings of the device. The profiled data can then be further used by optimization problems and algorithms which are designed to make offloading decisions.

Optimization problems and algorithms which make offloading decisions must rely on profiled data to make accurate decisions. Data profiling is a problem in mobile computation cloud offloading because there has yet to be a standardized method for profiling data which allows for simple and modular integration. The aim of this thesis is to provide a tool which solves the data profiling problem related to mobile computation cloud offloading on the Android operating system.

1.3 Approach

An approach to solving the profiling problem in mobile computation cloud offloading includes a significant usage of reflection. Reflection is the ability to modify and examine the structure and behavior of values, meta-data, properties, and functions of a program at runtime. In this thesis reflection is used to provide a generic solution for profiling. The naming convention of many of the classes included in Chapter 3 include the term "Generic". This is because the usage of reflection allows these classes to operate generically over any object or class definition in Java. The purpose of introducing these generic classes is to provide the profiling tool mentioned earlier which will solve the profiling problem for third party developers in the field of mobile computation cloud offloading.

1.4 Thesis Contribution

In this thesis we introduce the Generic Profiler. The Generic Profiler is designed on top of the Android Content Provider and uses reflection where necessary to make data storage

as generic as possible. The Generic Profiler is comprised of the Generic Content Provider, Generic Contract Class, Generic Database Helper, Table Builder, and a Content Helper. These classes are all necessary to fully implement a Content Provider in Android which allows a developer to query, add, update, and delete object entries to a SQLite database.

We also include a set of profilers which fully implement the Generic Profiler. These profilers include a Software, Wi-Fi, Telephony, Location, and Battery profiler. Of these profilers, the Software profiler is the only active profiler and the rest are all passive. That is, the Software Profiler actively fetches all method and thread invocations on a regular basis, where as the others passively listen for changes to occur.

1.5 Thesis Organization

Chapter 2 of this thesis is a Literature Review which begins by explaining in detail what a Service Based Cloud is and the difference between Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Major topics including the General Architecture, Remote Connection, Code Offloading, and Policies of a Mobile Cloud are included to provide a little bit of related work to the topic. Types of profilers are also discussed as well as solutions for Performance and Network Estimation, Energy Optimization, Performance Optimization, Energy vs. Performance Tradeoff, and the main challenges today in regards to mobile cloud offloading.

Chapter 3 discusses the Methodologies of this thesis as well as the Generic Profiler which is designed to make data profiling for third party developers very efficient and a set of profilers which demonstrate the effectiveness of the Generic Profiler. In Chapter 4 we discuss the evaluation of the Generic Profiler. The evaluation considers the development effort contributed towards building the Generic Profiler and the effort involved in building

a set of profilers from scratch as compared to implementing the Generic Profiler. Chapter four also includes a performance evaluation, in terms of inclusive invocation time, of the Software Profiler over an NQueens algorithm. The software profiler records iterations of the NQueens algorithm and the invocation time is evaluated for both batch and individual entry profiling scenarios. The performance, also in terms of inclusive invocation time, of the Location Profiler, Wi-Fi Profiler, Telephony Profiler, and Battery Profiler are also evaluated. The NQueens algorithm is a commonly used algorithm during the evaluation process of topics related to mobile computation cloud offloading. Finally, Chapter 5 discusses the conclusion and possible future work that this thesis may lead to.

Chapter 2

Related Work

Mobile cloud computing refers to the offering of a vast amount of dynamic resources to mobile devices through the use of the Internet. The National Institute of Standards and Technology (NIST) defines mobile cloud computing as "A model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [2]. A cloud usually consists of many data centers and is designed to offer a pool of services in a ubiquitous manner. Key features of a cloud center include agility, location independence, multi-tenancy, reliability, scalability, and maintenance. More specifically, resources should be provisioned relatively rapidly, available from anywhere, shared amongst many users, confidently available at all times, dynamically configurable, and maintenance should be minimal for all users [3]. In a cloud, all services are made available to the user through the Internet.

2.1 Service Based Cloud

Methodologies of cloud computing include Service Oriented Architecture (SOA) and Virtualization. A cloud should be capable of offering anything as a service (XaaS). Three of the most common services offered by cloud centers include Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). However, there are many more granular types of services which may include storage, database, information, process, integration, security, management/governance, and testing [3]. Here, the three main types of services will be visited further.

2.1.1 Software as a Service (SaaS)

SaaS is the most common type of service offered by a cloud center. Some good examples of SaaS include both web and mobile applications where the user interface may only be a thin client which is driven by data stored in the cloud. Cloud applications may be offered to the end user no matter where they are and across multiple platforms, granted the application developers support the platform. Some popular examples of SaaS include email services such as Google's Gmail [4] and Microsoft's Live [5], as well as document suites including Google Docs [6] and Microsoft Office 365 [7]. A single great feature of cloud document suites includes the ability to share and edit documents easily amongst groups of people operating on a variety of machines.

2.1.2 Platform as a Service (PaaS)

Simple hosting solutions are usually offered as a PaaS for developers. Tools available through PaaS may aid application design, development, testing, deployment, hosting, and storage. However, PaaS is not limited to developer tools. It may also offer a com-

bination of other team collaboration or common analysis services packaged into a single integrated solution available through the internet [3]. Popular solutions offering PaaS include Google's App Engine [8], Microsoft Azure [5], and Amazon Web Services (AWS) [9].

2.1.3 Infrastructure as a Service (IaaS)

Opposed to offering software or packaged tools to aid development, IaaS usually offers hardware in the form of Virtual Private Servers (VPS). The flexibility of a VPS allows the users to apply any server configuration they desire. Infrastructure is usually offered to the customer on a pay-as-you-go basis where resources can be dynamically allocated based on server load. Major competitors in IaaS include Google's Compute Engine [10], Microsoft Azure [5], and Amazon EC2 [9].

2.2 Mobile Cloud Offloading

Mobile devices are becoming extremely popular these days and an essential part of everyday life. Unfortunately, energy consumption, performance optimization, and network availability tend to be common problems amongst mobile devices. The study of mobile cloud offloading attempts to solve these problems.

Mobile cloud offloading includes resource monitoring, cost and partition models, and decision making [11]. The resource-monitoring component is usually referred to as a profiler in many solutions and has a main purpose of collecting useful information, in regards to mobile cloud offloading, from the mobile device. The collected data is further used for solving optimization problems or making context-aware decisions. In this section, the general architecture of mobile cloud offloading, common approaches for solving energy

and performance issues of mobile devices, and some policies which must be considered will be visited.

2.2.1 General Architecture

The bare minimum requirements for mobile cloud offloading include a mobile device with limited resources and a cloud server of some sort. The mobile device and the cloud are connected wirelessly via various technologies including Wi-Fi, 3G, LTE, etc. A highly popular mobile operating system used for cloud offloading is Google's Android because it is open source and has a large community and support. In some cases the Android instance running in the cloud is a lightweight version without a user interface [12]. The Android solutions usually include a dedicated version of Android running in the cloud with some sort of synchronization scheme.

2.2.2 Remote Connection

An early attempt at mobile cloud offloading involves running a mobile image (i.e. a copy of the entire state of the mobile system usually stored in a non-volatile form such as a file) on a remote server and connecting to it remotely from a mobile device. Having the image run solely in the cloud relieves the necessity for a data synchronization scheme. Chen et. al. implement this solution and find that the significant resource gain by the cloud allowed the mobile device to run applications normally unavailable to resource poor devices [13]. Unfortunately, the constant streaming of screen images (i.e. screen shot or picture of the screen at time t) in this virtualization technique happens to be a large bottleneck. Although performance had increased, energy consumption and end user experience had suffered because of the constant use of networking interfaces on the

device [13]. This technique cannot be a viable solution for energy optimization because of the constant energy consumed by the networking interfaces and the large network dependency of the remote connection.

2.2.3 Code Offloading

Opposed to remote connection, data synchronization and virtualization methods are employed in order to facilitate computation offloads to the cloud. A proposed virtualization environment for Android applications by Hung et. al. includes a framework that would perform the remote installation of their SDK, an allocation mechanism for choosing a remote cloud, the initialization of a remote environment, and a method for synchronizing state between client and server [14]. ThinkAir [12], CDroid [15], CADA [16], and CloneCloud [17] are all working solutions which take advantage of this methodology.

Pu et. al. propose a virtual cloud solution called SmartVirtCloud which emphasizes cloud discovery [18]. ThinkAir attempts to dynamically adapt to changing network availability, provide simple developer interfaces for computation offloading consideration, significantly improve energy consumption and performance, and dynamically scale remote computational power based on user requirements [12]. Both solutions rely on data annotations which mark specific chunks of code that qualify for offloading. SmartVirtCloud only allows static methods to be marked for offloading, however, ThinkAir allows any method.

Since research in this field is relatively new, most researchers assume that cloud resources are unlimited or free. In reality, cloud offloading would likely be offered as PaaS which an end user would signup and pay for. SmartVirtCloud establishes a multicast system for discovering third party cloud computation centers which opens up the possibility

of monetization schemes.

2.2.4 Policies

Policies in mobile cloud offloading include Performance Optimization, Energy Optimization, the Trade-off between Energy and Performance Optimization, and the Cloud Service Level Trade-off (i.e. Comparing viable cloud options and selecting the cloud which can offer the best service based on the offloading requirements) [12].

2.3 Profiling

Events which occur on a mobile device can be classified as either user events or system events [19]. User events may include tasks such as sending and receiving emails, exchanging files through an application, charging the battery, and installing and uninstalling applications. System events may include device driven tasks such as using network interfaces, GPS radios, applications reading and writing application data, and automatically updating applications. Both user and device driven event data can be collected through the use of profilers.

Components in the Android operating system are often loosely coupled and take advantage of Android's intent broadcasting system for intercommunication between each other. This allows for simple implementation of passive monitoring. Intents are messages that get broadcast throughout the Android environment and any application may register to receive specified intents. Some types of data which can be collected passively using this broadcasting system include network connectivity, battery status, and screen state. However, not all event data can be collected passively. Application data stored as files and databases must be collected actively by scheduling jobs which run periodically on

the device.

Profilers which provide the collected data to the offloading decision-making model [12] capture data both passively and actively . In order to optimize decision-making and improve offloading benefits, the profilers must be both lightweight and accurate because too much overhead or false data could potentially reduce the benefit of offloading. Since profilers could be updated or added in future work, the implementation should be quite modular to allow for simple integration. A few effective profiler types used in cloud offloading include hardware, software, network, context, preference, and environmental profilers which will be discussed further. A method for storing the collected data using a Content Provider in Android will also be discussed. Android has been chosen as the operating system used in this thesis as opposed to other mobile operating systems because of advantages which will be discussed in Chapter 3.

2.3.1 Hardware Profiling

Hardware profilers are responsible for collecting hardware state information. Some examples of hardware state include Central Processing Unit (CPU) clock-frequency, screen brightness level, and network interface state (i.e. on, off, or idle). A good hardware profiler should attempt to make and record performance estimations based off the hardware limitations of any given mobile device.

Performance estimation is mostly affected by factors which include CPU frequency, Last Level Cache (LLC) hit, LLC miss, and when the store buffer is full. There are more factors which effect performance estimation, but these tend to be the most prominent. A high-level performance estimation model is offered by Chae et. al. and would be effective for CPU profiling [20].

Since performance is greatly determined by the hardware of a system, adjusting factors must be considered as well. The goal is to examine the performance of a target machine when compared to a reference machine. In an offloading solution with a single cloud, the reference machine is always the mobile device and the target machine is the cloud server. However, in a system which includes multiple offloading options, it becomes necessary to compare each individual cloud server so the best option for offloading can be selected.

2.3.2 Software Profiling

Software Profilers are responsible for collecting metrics related to computation execution time. The Android Debug API offered by Google includes tools for capturing the overall execution time of a method, thread CPU time of a method, number of executed instructions, number of method calls, thread memory allocation size, and number of garbage collections.

2.3.3 Network Profiling

The Network Profiler is responsible for collecting network related data. Some modern technologies used by mobile devices for connecting to the internet include Wi-Fi, 2G (GPRS), 3G (UMTS, HSDPA, HSDPA+), and 4G (LTE, LTE Advanced). Useful network data includes network latency, congestion [16, 12], and wireless signal strength. Data must be collected from all wireless sources in order to choose the best channel for offloading at any given time.

2.3.4 Context Profiling

Context Profilers collect data relevant to offloading decisions based on time-of-day and location. Instead of collecting data to be used by an optimization problems, the Context Profiler collects data that can be helpful overtime [16]. However, the historical data offered by solving optimization problems from other profilers may help a Context-Aware solution make quicker offloading decisions. This process may reduce the necessity of solving optimization problems in future scenarios. For example, a location which is visited often never has network availability, therefore it is not necessary to solve an optimization problem. Instead, history suggests that the computation be performed locally.

2.3.5 Preference Profiling

User specific preferences can also affect the decisions made by an offloading model [1]. In practice, different users may prefer various types of service and the collection of preference related data can help tailor offloading decisions towards specific individuals. For example, one user may want to conserve as much energy as possible when using a GPS application and is willing to wait longer periods of time before a location change is triggered, whereas another user would prefer better performance which results in less time before triggering a location change, but higher energy consumption. Preferences may be profiled by allowing users to generate or specify their own unique preferences.

2.3.6 Environmental Profiling

Environmental Profilers monitor potential resource providers such as cloud computing centers, cloudlets, and mobile cloud farms [1]. Cloud providers may offer offloading packages in tiered service plans where users could opt-in and pay for better service which

would guarantee a predefined level of performance. Furthermore, any given resource provider could get selected based off the cost of offloading with predefined performance requirements. Free to use services may only offer limited resources, but tiered plans could guarantee significant improvement at different rates. Once again, this option opens up the opportunity for the monetization of mobile cloud offloading.

2.3.7 Android Content Provider

Storing data and making it available to other applications is primarily done through the use of a Content Provider [21] in Android. Content Providers are responsible for implementing the CRUD operations which interact with a SQLite database which include query, insert, update, and delete. They also require the instantiation method onCreate and the getType method for acquiring the mime type of the data entry to be implemented. Content Providers also provide batch access to the SQLite database. That is, one may instantiate a list of ContentProviderOperation objects, each a new operation to perform on the database, and apply them as a batch operation to the Content Provider. The Content provider will be able to perform the operations more efficiently than performing each one at a time. Read and write permissions must also be specified in the application manifest in order to grant a third party application access to the Content Provider.

Interacting with a Content Provider can be done through the use of the Content Resolver interface. The Content Resolver will fetch the desired Content Provider by providing the associated Content Uri which contains the authority (i.e. owner) of the Content Provider and the route to either a single entry or many entries of a specific data type. The Content Uri dictates which table the Content Provider will access. The CRUD data access methods are thread-safe and can be called from any thread because the

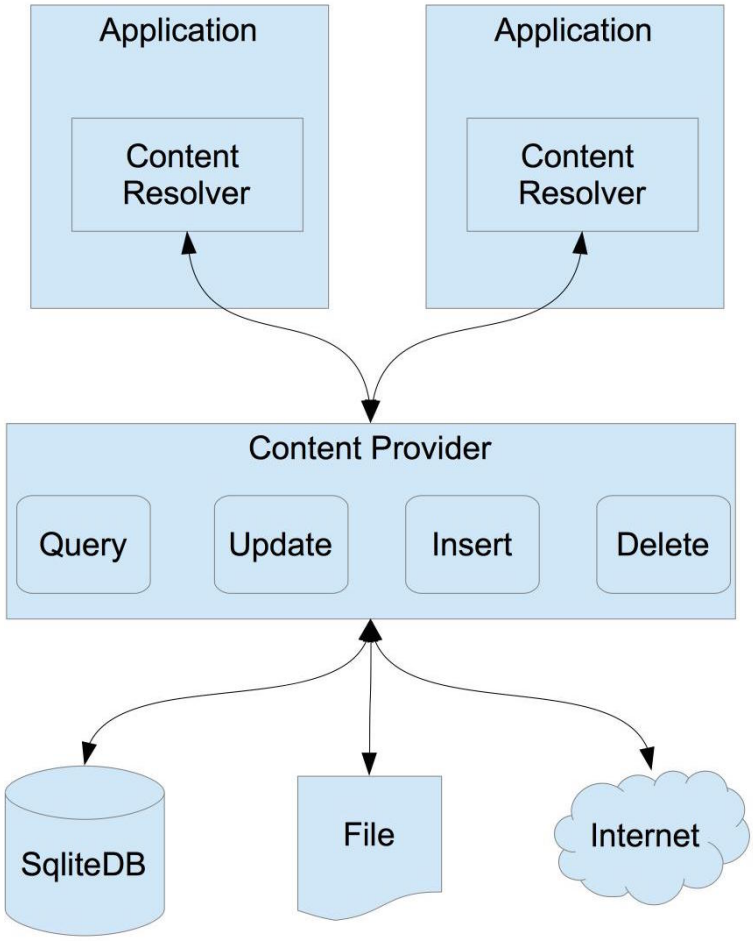

Figure 2.1: Overview of a Content Provider

Content Resolver automatically handles inter-process communication. When querying a Content Provider a projection, selection, selection arguments, and sort order must be specified. The projection is usually an array of column names that should be returned. The selection specifies the criteria for selecting rows. The selection arguments is an array of strings which replace the question mark characters in the selection string. Finally, the sort order string is similar to a SQLite order by command.

It is usually good practice to implement Contract Classes when choosing to implement a Content Provider. A Contract Class defines any given table or model constants that can be used with a Content Provider. The most common properties include the Content Uri, the projection of the table, constants for each column, queries for creating and dropping the table, getting content values given a Cursor, and getting the desired object from the content values.

Implementation of a SqliteOpenHelper is also necessary when implementing a Content Provider. The SqliteOpenHelper deals with database connections including creating and dropping tables. Usually the create and drop table queries presented in the Contract Classes will be used by the SqliteOpenHelper.

2.4 Estimations

The capability of estimating both performance and network states at a given time can help make offloading decisions without the necessity of optimization problems. A context-aware solution could make use of an estimation model because decisions could be made ahead of time. A naive history-based approach will be discussed further.

2.4.1 Naive History-Based Approach Using an Exponential Moving Average (EMA)

A naive approach for estimating both the execution time and energy consumption of a computation can be performed by recording historical observations over time and further computing an Exponential Moving Average [22, 23]. Higher weights can be applied to newer observations and noisy observations (i.e. those which should minimally affect the results) may be easily ignored.

A simple moving average at time t can be calculated using equation 2.1 where N is the observation window size, M_{t-1} is the previous moving average at time $t-1$, x_t is the current observation at time t, and x_{t-N} is the observation at time $t-N$. It is unnecessary to record the moving average at all times, however averages from time t to $t-N$ must still be recorded. This method of averaging allows for significantly less arithmetic than normal averaging.

$$M_t = M_{t-1} + \frac{x_t - x_{t-N}}{N} \qquad [23] \tag{2.1}$$

Furthermore, it will take N new observations before the moving average is significantly changed. In systems with significant noise a large value of N would provide a stable estimate. However, systems which accept large fluctuations should use a small value of N [23].

Equation 2.2 shows the application of the smoothing estimation or EMA and offers the new estimation in a constant model. It becomes unnecessary to record the historical observations since time $t-N$ while calculating the EMA. Application of the EMA is quite flexible, especially for making estimations. When the smoothing constant α is large the estimate $S(x)$ will respond quickly to changes in new observations and when α is small

$S(x)$ will respond slowly.

$$S_t(x) = \alpha x_t + (1 - \alpha) S_{t-1}(x) \qquad [23] \tag{2.2}$$

$$\alpha = \frac{1}{N} \qquad [23] \tag{2.3}$$

Xia et. al. [22] estimate CPU workload and network bandwidth by adopting a modified version of the EMA provided by Burgstahler et. al. [24]. The profiler periodically records the CPU workload c_t and the time it was recorded at time t. Equation 2.5 shows the estimated CPU workload at time t and in equation 2.6 the degree of weighting decrease coefficient α is slightly adjusted.

$$C_1 = c_1 \qquad [22] \tag{2.4}$$

$$C_t = \alpha c_t + (1 - \alpha) C_{t-1} \qquad [22] \tag{2.5}$$

$$\alpha = \frac{2}{N + 1} \qquad [22] \tag{2.6}$$

Network Bandwidth is similarly estimated in equation 2.7.

$$B_t = \alpha b_t + (1 - \alpha) B_{t-1} \qquad [22] \tag{2.7}$$

2.5 Energy Optimization

Energy optimization is one of the most important demands for mobile applications [25]. As the demand for computationally expensive mobile applications continues to grow so does the energy consumed by them. As mobile devices begin to suffer from short battery life this challenge becomes quite evident. Also, in order to benefit from the offload in regards to energy consumption, the cost of turning on wireless interfaces and transmitting data must be lower than the cost of performing the computation locally.

Solving energy optimization problems usually starts by recording useful information using a combination of the profilers mentioned earlier [12, 18, 19, 15, 16, 25]. Criteria such as the screen, CPU, wireless interfaces, and any other energy heavy sensors included in a mobile device must be considered in energy optimization. Kosta et. al. develops an energy estimation model in ThinkAir [12] using recorded energy consumption metrics provided by PowerTutor [26]. These results are further used for making the offloading decisions in ThinkAir. Kosta et. al. found that ThinkAir is only efficient for optimizing energy consumption when offloading computationally expensive methods.

CDroid considers the trade-off between energy consumed when running the computation locally as opposed to accessing wireless routers for performing the offload [15]. Barbera's solution includes many different modules for performing dedicated tasks, but with the final goal of decreasing energy consumption of the mobile device. Some of the CDroid modules dedicated to energy optimization include a mobile advertisement blocker, content compression, and a push notification handler.

Many Android applications are published free to download and free to use. This is made possible for developers because Android includes advertisement options that generate revenue. Unfortunately, the end user may not be interested in the ads and the

ads themselves may cumulatively waste considerable energy. In CDroid all web requests are channeled through a proxy, and if the ad blocker detects ads, it will return a denied message to the client. Furthermore, all HTTP requests and responses between client and proxy server are compressed to gzip [27] streams and images are converted to jpeg. Jpeg images benefit from a smaller file size than PNG, but they do not support transparencies. Since many applications rely on their image resources to have transparencies in them, they may not look as intended. In this particular case, the user should have the option to turn this feature on or off.

The push notification handler attempts to combine all push messaging services running on the client into a single service. Many applications leverage push messaging systems in order to provide data in real time. However, these services start to add up and become quite taxing on the devices energy. CDroid listens for such services and combines them all into a single handler running in the cloud. The push handler aggregates and batches all of the push messages from multiple sources and then delivers them to the client in a close to real time fashion. These modules, although individually insignificant, help optimize energy consumption significantly when combined. Once again, the end user should have the option to disable the push messaging handler or modify the frequency of receiving push messages. This is because many push messaging systems are designed to send messages in real time. Users may mistakenly believe this feature to be an application bug, as opposed to an offloading solution for saving energy.

It is essential to visit strategies for modeling and estimating energy consumption as well as solving energy optimization problems. Methods which are used for modeling energy consumption will be further discussed in this section and include a Context-Aware Energy Model, a Hardware Throttling Energy Model, and a general method for modeling energy consumption.

2.5.1 Context-Aware Energy Model

A context-aware communication energy model is derived by Lin et. al. and considers both network congestion and transmission power. Network congestion is a function of throughput and transmission power is a function of the Received Signal Strength Indication (RSSI) of a wireless network. In this model a profiler periodically samples the throughput and RSSI where a mapping between RSSI and the drawn current of the device is achieved through experiments using an Agilent 66321D power meter [16].

Modeling energy consumption is achieved using PowerTutor [28], an open-source energy monitoring application available for the Android operating system, which records all energy observations necessary for their energy model. The underlying energy model is composed of the component-wise summation of CPU, communication, display, and other energy usage.

2.5.2 Hardware Throttling Energy Model

Wen et. al. [25] have developed an optimal application execution policy which minimizes energy consumed by mobile devices. This energy model is mainly focused on the development of optimization problems [25]. Energy consumption is further modeled as a function of CPU workload, CPU cycles required to complete a task, and the input data required to perform the task. Energy optimization is accomplished by controlling the frequency of the CPU using dynamic voltage scaling (DVS) and the data rate of wireless interfaces [29]. Unfortunately, in this model performance may be sacrificed in favor of energy conservation.

2.5.3 Modeling Energy Consumption

Elgazzar et. al. [1] offer a method for modeling energy consumption and processing time in their architecture for mobile cloud offloading which can be seen in Figure 2.2. This data-modeling scheme is quite effective when paired with an enabling mechanism for both context-aware offloading and optimization problem solvers. In this model the Mobile Service Provider is a mobile application or operating system module which is used for making offloading decisions. Although the Mobile Service Provider still exists on the mobile device, data flow to and from it must be considered. Elgazzar et. al. assume that the data transferred during a computation is only transferred once. In reality, the data may get transmitted multiple times [1].

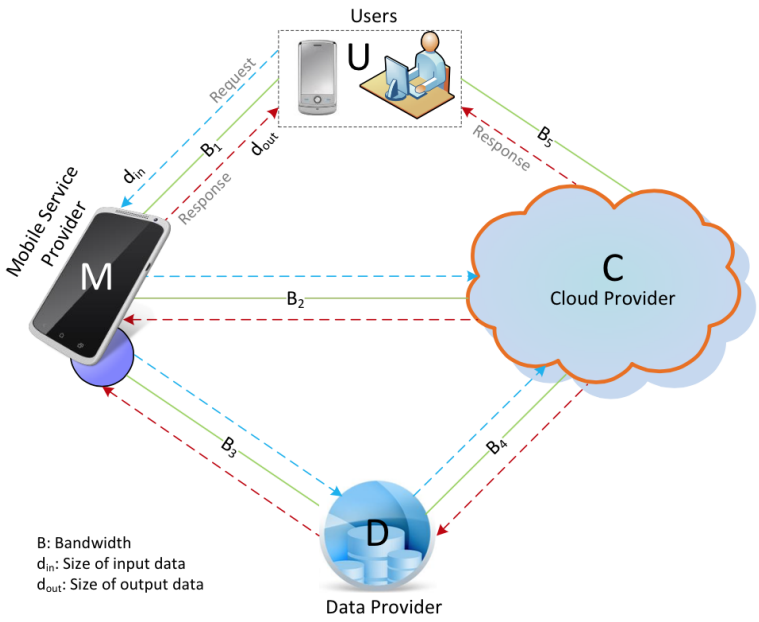

Figure 2.2: General architecture of a cloud-assisted computation offloading system to support mobile services [1]

2.6 Performance Optimization

End users may want specific tasks to perform as optimally as possible, and in many cases, better than their mobile device can provide. In these specific situations performance optimization becomes crucial. ThinkAir makes an attempt to not only consider energy optimization, but also consider performance optimization [12]. The execution controller used in ThinkAir makes the offloading decision based on energy and performance benefits. The user upon configuration provides preference. Once again, an improved performance optimization model coupled with an energy optimization model and a context aware decision algorithm would make a great contribution to the mobile cloud offloading solution. Unfortunately, research in performance optimization in regards to mobile cloud offloading is minimal.

Wu et al. [11] attempt to make trade-off decisions between energy and performance optimizations. However, the overhead of solving performance optimization problems may contribute to further energy consumption. In times of insignificant energy reserves a trade-off decision must be made in order to conserve energy [25]. Rather than solving a performance optimization problem, Wen et al. [25] improve performance by making context aware decisions.

2.7 Tradeoff Decisions

Solving optimization problems may result in benefits for both energy consumption and performance, in these cases offloading will always be performed. If the results show no improvement in either case, then the offloading will not occur. However, in many cases one must consider the tradeoff between energy and performance optimization (i.e. a

decision must be made in order to improve energy consumption or performance, but not both). Using data collected from a profiler, a cost model can be formed and an offloading decision can be made from the cost criteria. Criteria which should be minimized includes energy consumption, offloading cost, and storage space. However, maximization criteria can include performance, robustness, and security [11]. It is important to consider the cost of offloading computations in combination with the remote execution time. That is, the network overhead may cause remote execution time or the energy consumed while the device is waiting for a response to be greater than that of the local execution.

Edge cases include the execution times when offloading fails to provide improvements in either performance or energy consumption [11]. Investigating and modeling these extreme cases helps determine when it is necessary to make a tradeoff decision. Although a tradeoff decision will not often result in significant improvement, it will offer some improvement for either performance or energy. The tradeoff then becomes a user preference that should impact the gravity of the decision.

2.8 Challenges

2.8.1 Offloadable Code Discovery

A preliminary challenge in mobile cloud offloading is determining whether or not a computation qualifies for offloading. Many computations may be regarded as computationally inexpensive and may never qualify for offloading.

Some attempts include the necessity of applying data annotations to methods which qualify for offloading [12, 18]. ThinkAir [12] and SmartVirtCloud [18] are examples which require application developers to perform this manual annotation procedure. In ThinkAir

only methods are required to be annotated, whereas in SmartVirtCloud container classes are required to be annotated as well. This procedure adds extra developer overhead, but may allow offloading decisions to be made quicker. Application developers are also required to download and implement the ThinkAir SDK for compatibility. A code generator is then used during compilation in order to create both a client and cloud version of the application. This generator is necessary to convert ARM code to x86 code which the cloud server can support. The purpose of including a developer SDK is to allow simple implementation for the application developer. SmartVirtCloud does not provide a developer SDK, however an application encapsulation tool was provided to transform the application code into something that can be used with their solution.

CloneCloud makes implementation automatic [17]. Rather than annotating methods with specific tags [12], CloneCloud searches the compiled dalvik bytecode of an application for threads that could be offloaded. Threads which qualify for offloading are partitioned and recorded for use by a dynamic profiler. Performing the tagging and code generation process automatically provides benefits to developers because of the exclusion of manual developer overhead.

The discovery process breaks the code into partitions using a static analyzer and a dynamic profiler which records the partitions. The static analyzer attempts to prevent any methods that rely on the device's internal sensory hardware or system libraries from being considered for offloading [17].

Kovachev et. al. propose a service-based implementation to cloud offloading which also relieves the burden of manually applying data annotations [30]. They suggest a policy where mobile developers must register for services which are offered both locally on the client as well as in computation servers. Decisions for offloading service execution is performed by the service manager built directly into the operating system [30].

Unfortunately, the necessity of writing services twice so they can be run either locally or remotely may not seem appealing to most developers. This process also involves more manual overhead than applying annotations manually because developers are still required to register for services and implement a local implementation as well.

2.8.2 Computation Offloading Decision

After determining which code qualifies for offloading a decision of whether or not to offload must be made. A trivial attempt includes offloading all annotated methods without considering environmental factors at all [18]. However, modern attempts use either context aware solutions which adapt to environmental factors [12, 17] or solve optimization problems which are designed to minimize energy consumption or maximize performance [15, 27, 16, 25].

In ThinkAir and the tactics-based approach offered by Balan et. al. a series of profilers is used to collect software, hardware, and network metrics [31]. The profilers help collect computations which qualify for offloading. Once collected, ThinkAir uses an execution controller which monitors and records environmental and contextual data of the mobile device and uses it to make the offloading decision [12]. The final decision is made based on past invocation results determined by execution time and energy consumption. ThinkAir policies are quite trivial and could be improved in future work.

SmartVirtCloud makes tradeoff decisions between energy consumption and performance. If the device is low on energy it will make the offloading decision based on energy consumption, otherwise it will gravitate towards performance. Decisions are recorded in an XML file after having run the application many times with different energy parameters. During runtime the decision is simply looked up in the XML file [18]. Offloading

decisions continue to be updated in XML during regular usage thus potentially improving decision making over time.

The dynamic profiler in CloneCloud models application partitions into a tree structure and analyzes the cost of performing partitions both locally and on the cloud [17]. In order to consider many operational use cases, the application is run many times with different operation parameters. An optimization solver further processes results and a decision is made. This solution could be further improved by building it directly into the operating system. CloneCloud could be extended to many different cloud providers which wish to provide a computational offloading service to end users opening up opportunity for monetization schemes.

Partitioned code is performed on a per thread basis. When an offloading decision is made the client thread is suspended and all object data is captured by a migration manager [17]. CloneCloud at a byte-code level where all objects are mapped in an object table does this quite elegantly. When the offloaded code has completed, the client thread is resumed at an entrance point where merging can be performed. A successful merge is done by comparing the object tables of both the client and server and making the appropriate changes [17]. Compared to other solutions which tend to offload methods, this approach would ensure less network overhead is performed.

2.8.3 Cloud Discovery

Since research in mobile cloud offloading is still relatively new, many solutions only consider a single cloud server. However, some solutions do include discovery of nearest cloud servers as well as possible handoffs. Rather than performing a web-based implementation for determining closest cloud [32] Pu et. al. use a multicast method in SmartVirtCloud.

SmartVirtCloud was designed to work within a library or school environment where the Wireless Local Area Network (WLAN) would be adjusted to have a cloud or proxy server responding to multicast messages throughout the network [18].

Tasnim et. al. propose a location aware and quality of service solution for determining the most appropriate cloud server to use [33]. Instead of using a multicast system like SmartVirtCloud [18], their solution is composed of a location change decision module, profile tree building module, and a service migration decision module. When the location of a mobile device changes the location change decision module decides whether or not a closer server should be used instead. If the decision to switch servers is made then a profile tree is created using the server capacity vectors of each individual machine. When a server is found to offer higher performance, a switch to the new server will be made immediately. Computations at the old machine will be terminated if they are estimated to be completed sooner at the new machine [33].

2.8.4 Task Scheduling

The incorporation of multiple cloud systems must involve a task scheduling process. In SmartVirtCloud a proxy server or load balancer is used to forward the computation to a server that can handle the request. Multicast messages include the application and method names that need to be performed. Only servers that can fulfill the request will be considered and Java reflection is used to perform the computation remotely [18].

In order to maintain simplicity, ThinkAir does not empower a dynamic cloud architecture. Instead six Android VM images are used, running in a VirtualBox environment [34], and consist of progressively improved resources. Based off historical runtime data, the appropriate VM can be utilized for optimum results. Since the startup of a VM could

potentially take a significant amount of time, ThinkAir uses secondary servers that can be in either a powered-off, paused, or running state. Moving a machine from paused to running is significantly quicker than booting a brand new VM [12]. In a system that includes many end users as well as multiple clouds, being able to manage and predict the type of server resources a client may need on demand is extremely crucial. Another benefit to this solution includes the ability to re-allocate a more powerful VM in case of Android out-of-memory errors. The user can still run an application even though their device does not have the required resources to run it under normal circumstances.

2.8.5 Data Synchronization

It is necessary to provide data synchronization techniques between client and cloud in cases where computations require system data. The Android operating system provides file space for each application and consists of an application package, configuration files, database files, cache buffers, and saved state files [14]. Shared memory also exists for many types of files which may be stored and accessed by any application. In order to successfully run code remotely, all this data must be synchronized. Since our goal is to improve performance it is crucial that data synchronization exists in cloud offloading with as little overhead as possible. A poor data synchronization technique could negatively affect the cloud offloading performance.

Batch Synchronization

CDroid performs batch updates which piggyback on regular traffic [15]. All traffic passes through the cloud which acts as a proxy and is also commonly referred to as proxy-based aggregated synchronization in [35]. Every bit of data is stored remotely, effectively

synchronizing most network traffic. Data which does not normally pass through the network requires a batching process to be performed. Batches are applied periodically as a piggyback to the next available network request and synched when the cloud receives it.

Hung et. al. also propose a batching process for data synchronization [14]. Instead of synchronizing data when it is needed, they opt for a simpler solution which involves utilizing the Android application life cycles. Whenever an application, activity, or service enters an OnPause() function, the system performs an application wide file system synchronization. That is, any configuration, database files, cache buffer, or saved state files relevant to the application are synchronized. Hung et. al. propose this method of synchronization, however, they do not actually provide a solution with any feasible results.

Per-Need Basis Synchronization

Synchronizing data on a per-need basis refers to the act of synchronizing only when it is absolutely needed [12]. During the offloading phase all objects, parameters, and files necessary to perform the computation are gathered and added to the offloaded request. The cloud uses the transmitted data for performing the current computation and stores the data for future requests. Data storing avoids the necessity of synchronizing data on subsequent requests and provides a performance improvement over batch synchronization. This method of synchronization is utilized in CloneCloud on a thread-based level. All data which is necessary for thread completion is transmitted during the offloading phase along with the thread itself.

Delta Updates and Proxy-Based Synchronization

Barbera et. al. achieve data synchronization by using delta updates and proxy-based methods [19]. Full computational support by the cloud can be best accomplished when all mobile data is synched to the cloud. In [19], Barbera suggests a dedicated clone which provides this service through both passive and active data collection. Passive data collection involves the collection of user driven and system events including Android intents related to various environment data like device, network, battery, and screen state. Active data collection refers to a periodic collection of device status, currently running apps, phone calls, emails, text messages, and any other application or system data which can be collected. Barbera et. al. also make an effort to collect and synchronize user, application, and system files on the device [19]. File changes are captured using Android's built in support for the Linux component "inotify" [36], which is designed for monitoring changes to the Linux file system. The monitoring process involves recursively visiting all directories in the file system and adding inotify watches to all directories and subdirectories. By default, many system files are private to the system in Android and file permissions were changed temporarily using a rooted device with sudo access. As soon as a file gets detected as modified, the binary difference of files, using well-known techniques [37], is taken and further used for the actual synchronization. Performing synchronization of files based on binary differences ensures a full data synchronization process with as little overhead as possible.

Lee et. al. [35] suggest that proxy-based synchronization should be performed similarly to the methods in [19]. That is, all proxy-based data transfers must be performed using Update-Triggered Delta Synchronization, where only the deltas or updated information is transferred [35]. The goal is to minimize synchronization and file access as

much as possible. Proxy-Based synchronization can also be scaled to a larger network which includes multiple proxy-servers. An individual proxy-server is required to notify neighbor proxy-servers of all necessary synchronizations. This ensures that an end user in transit can conveniently switch to the best proxy-server with little transfer overhead. It is unlikely however, that many proxy servers would be developed in a fashion similar to mobile networking handoffs because a user would have to travel a significant distance before experiencing a hand-off between proxy servers. An improvement includes a detection system which recognizes when a user is about to switch proxy-servers and in preparation perform a timely synchronization. Otherwise, much bandwidth would be wasted synchronizing data between proxy-servers which a user may never visit.

Chapter 3

Methodologies

3.1 Introduction

In this chapter, we focus on the integral part of data profiling, minimizing the overhead during the collection process, and making the collection process as generic and simple as possible for third party developers to use. We will discuss our solution for profiling data called the Generic Profiler and a combination of profiling services which take advantage of the tool. The Generic Profiler is composed of five independent classes which include a Generic Content Provider, Generic Contract Class, Generic Database Helper, Table Builder, and a Content Helper which will all be discussed in detail. The profilers provided include a Location Profiler, Software Profiler, Wi-Fi Profiler, Telephony Profiler, and Battery Profiler.

3.2 Generic Profiler

3.2.1 Generic Content Provider

The Generic Content Provider extends the Android Content Provider class and implements the CRUD abstract operations which are required for implementation. A convenience function for retrieving Mime Type based on URI is also included. The Mime Type is composed of either the cursor directory base type or cursor item base type and followed by the provided class type offered in the URI. The class type is the last path segment in a URI for a directory and the second last for a single item.

The purpose of the Generic Content Provider is to replace the Android Content Provider which will significantly reduce the amount of development effort required to setup a Content Provider from scratch. Opposed to limiting the power of a Content Provider to a finite amount of provided Content URIs, the Generic Content Provider allows any class type which is registered with it to support the generation and validation of any URI provided to the Generic Content Provider. A developer may extend the Generic Content Provider with the benefit of having the generic CRUD operations already implemented for any registered class type and further implement the abstract methods necessary for the Database Helper to build a SQLite Database. The Generic Content Provider is instantiated at runtime by the Android operating system. During the onCreate() method the abstract class methods getAuthority(), getDatabase(), getDatabaseVersion(), and getObjectModels() are called and further used during instantiation of the Generic Database Helper. The expected implementation of each of the abstract methods is mentioned below:

getAuthority()

The getAuthority() method should return a string object representing the authority name which is used as the owner of the content provider. An owner of a Content Provider may wish to share its data and allow third party developers to specify their authority name. The authority name is usually in the form of "com.name.provider".

getDatabase()

The getDatabase() method should return a string object representing the desired name of the database. The form of the database string is usually "com.name.database".

getDatabaseVersion()

This function exists in order to give the developer the flexibility of upgrading their database schema. If the user wishes to register more class types with the Generic Content Provider then they will increase the version number of their database. The result will let the Generic Database Helper know that the database needs to be updated.

getObjectModels()

The getObjectModels() method should return a list of class types. The class Types are used by the Generic Database Helper for determining relationships of child objects and generating the schema for building and updating the database.

The CRUD operations of the Generic Content Provider accept a Content URI, a projection, a selection string, a list of selection arguments, and a sort order. The URI allows the Content Provider to determine which table the data should be selected from and in conjunction with the remaining parameters is used to generate a SQLite query. A projection is a list of string objects which resemble the column names of a table. Selection strings are similar to a typical SQLite select query, however it is unnecessary to specify

the SQLite 'select' clause or the table name from which to query. Arguments of the selection string can be represented as '?' characters where the actual value is stored in the list of argument strings. The order of arguments is important. Finally the sort order string is equivalent to the SQLite 'order by' clause.

There are four CRUD operations which are implemented by the query, insert, update, and delete methods. The insert method takes as arguments all five parameters. The update method takes as arguments the URI and a content values object. A content values object contains the key-value pairs of each tuple for all entries being inserted. Updating entries includes the same parameters as inserting as well as a selection string and a list of selection arguments. Finally, deleting entries accepts the same arguments as updating except for the content values object.

3.2.2 Generic Contract Class

The most important and powerful component of the Generic Profiler is the Generic Contract Class. Under normal circumstances, an object model would normally include a Contract Class which specifies the Content URI, projection (i.e. column names of a table), create and drop database queries, and operations which convert object model values to a ContentValues object. These properties and methods would have to be implemented manually by a developer for each and every object model. The Generic Contract Class completely abstracts this process consequently saving developers time and effort.

The Generic Contract Class includes the methods which are expected of a normal Contract Class, but does not include any constants which are normally provided for Content URIs, projections, and create and drop table queries. Instead, The Generic Contract Class provides static methods which return the expected results. Generic class

types and base object models are provided as arguments to the methods of the Generic Contract Class and reflection is used to return the desired response. The Generic Contract Class includes static methods for retrieving a Content URI, projection, table name, table object, create table query, drop table query, and get primary key. These methods simply accept a generic class type (i.e. in Java, ObjectName.class) as an argument. In Java the base object, Object class, is the root of all classes (i.e. every class has Object as a superclass). Methods for retrieving a ContentValues object and the primary key value of an object accept a base object as an argument. That is, because Object is the root of all classes, any object in Java can be accepted as an argument to these methods.

Generic Table and Content URI

Objects and object types provided to the Generic Contract Class completely dictate the schema of the database. Every class which is registered with the Generic Content Provider will be used to generate a SQLite table. The table name is represented by the full class name of the registered class. For example, the class TestClass in the Java package "com.coreytm.simplesync.library.models" would be "com.coreytm.simplesync.library.TestClass". Periods are then replaced with underscores because table names in a SQLite database cannot contain periods. The full name of the class is used because a class with the same name can exist in a different package and use of the short name could potentially result in collisions. A Table object is created through reflection of the provided class type and includes the table name and list of columns. A Column object contains the Column name and type which is determined by the corresponding Field type. Generation of the SQLite create and drop table queries is then much more straight forward using the resulting Table object.

The Content URI is composed of the Base Content URI provided by the Generic

Content Provider followed by the table name returned by the Generic Contract Class. The form of a content URI follows the pattern "content://authority/tableName/id" where "/id" is an optional segment which represents the unique id of a single entry.

Generic Projection

The full projection of a class can be represented as an array of String objects which dictate the column names of a SQLite query. Since there is a one-to-one relationship between the models and the SQLite tables, all Strings in the projection can be determined by a class type. The Generic Projection is generated by using reflection to iterate over a list of Field objects defined in the class type. A Field name is then added to the projection if the Field type is either a primitive or an object.

Generic Content Values

Android uses a ContentValues object for inserting and updating data of a SQLite database through the use of a ContentProvider. A ContentValues object is essentially a HashMap which contains the key-value pairs of each tuple in a single entry. Under normal circumstances, inputting the key-value pair for each Field of an object would have to be programmed for every model. However, by accepting an Object type as an argument, any object can be provided and the key-value pairs can be determined using reflection. All primitives and Strings which are not static variables are added to the ContentValues by iterating over every Field in the objects class type and further matching them to the corresponding values.

Primary Keys

Primary keys are essential in a relational database. Tables are automatically given incremental primary keys represented by '_id'. However, in some cases a developer may wish to define their own primary keys for their models. The developer can specify which Field in a model will be the primary key by annotating it with the 'PrimaryKey' data annotation provided in this solution. Providing a class type can retrieve primary keys and their values can be obtained by providing the object itself. Only one Field in an object can be a primary key, so the first Field annotated with 'PrimaryKey' is used. A primary key can be of type Long, Short, Integer, or String and is validated accordingly.

3.2.3 Generic Database Helper

The Generic Database Helper manages all database modifications. As mentioned earlier the Generic Database Helper is utilized by the Generic Content Provider and is provided with a version for the database and a list of class types through the abstract methods implemented by extending the Generic Content Provider. The Generic Database Helper extends SQLiteOpenHelper and implements the abstract methods onCreate() and onUpgrade(). During onCreate() the database is initialized for the first time at the specified version number and the list of class types is used by the Table Builder to build the tables. The onUpgrade() method will first drop all the tables, and then use the Table Builder to create tables based on the new list of class types.

3.2.4 Table Builder

The Table Builder aids in the creation and deletion of SQLite tables based on their class type counterparts. As mentioned earlier, there is a one-to-one relationship between a

SQLite table and a class type. The Generic Database Helper adds class types to the Table Builder and the table builder can be used to either create or drop tables for each of the class types provided. However, if a class contains children other than primitives or Strings (i.e. a single object or array of objects) a new table must be created for the child type and a relationship must be created. A HashMap of Table objects is used to record unique tables so as to avoid collisions in table creation. When a child type is visited, it is looked up using the HashMap and if it already exists relationships will be created, otherwise it will be added first. This method ensures that tables will be created for all children of the initially provided class types to the Table Builder, even if they were not specified.

Foreign keys are introduced during relationship creation and follow the form 'parentTableNameId_childFieldName'. This naming convention ensures that children are associated directly with the parent field in which they are related. When a child of a table is visited it is identified as either an object, an array of objects, an Iterable, a Map, or a ParameterizedType. For fields of type Object, providing the field type to the Generic Contract retrieves the table name of the child. However, for all others it is necessary to use the fields component type instead (i.e. if the field type was ArrayList<Object>, which is of type Iterable, then the component type would be Object). Once all tables and relationships have been identified the Table Builder will use the Generic Contract to create SQLite queries for table creation or deletion and then perform them on the SQLite database.

A simple example demonstrating how SQLite tables are generated using the Table Builder can be seen in Figure 3.1. In this example a class of type Object.class which contains fields for a unique identifier, two string objects, and a child object of type ChildObject.class is input into the Table Builder for table creation. Although the only

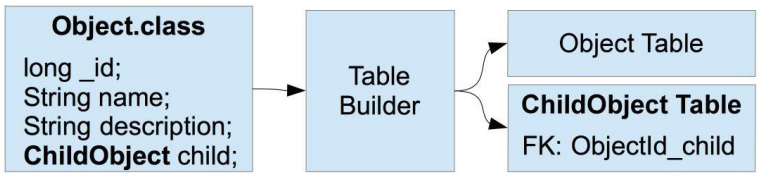

Figure 3.1: Example demonstrating how the Table Builder generates both parent and child SQLite tables and applies according foreign keys to the child table.

class type specified was Object.class, the Table Builder creates a table for ChildObject.class as well. The Table Builder also adds the foreign key column 'ObjetId_child' to the ChildObject table.

3.2.5 Content Helper

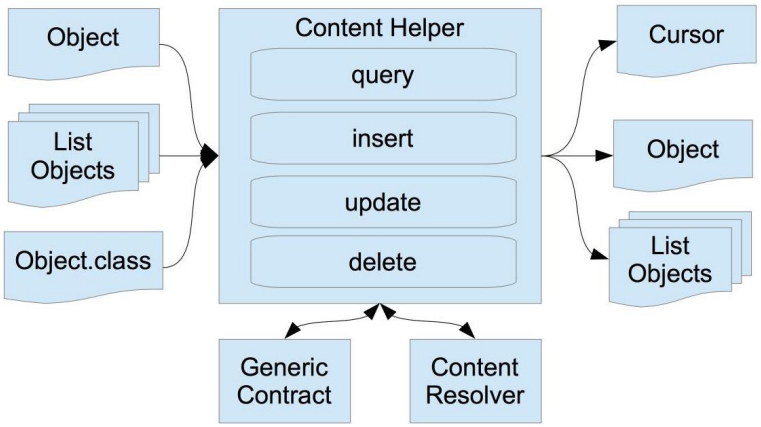

Figure 3.2: Overview of the Content Helper.

The Content Helper is the entity which allows the developer to perform CRUD operations on any object or array of objects. From a developer's perspective, this class abstracts all the features and convenience of a Content Provider and will be the developer's gateway to data storage. Figure 3.2 show an overview of the Content Helper.

The Content Helper accepts as input an object, an array of objects, or a class type. It then utilizes both the Generic Contract mentioned earlier and an instance of the Android Content Resolver to either insert, update, delete, or query data.

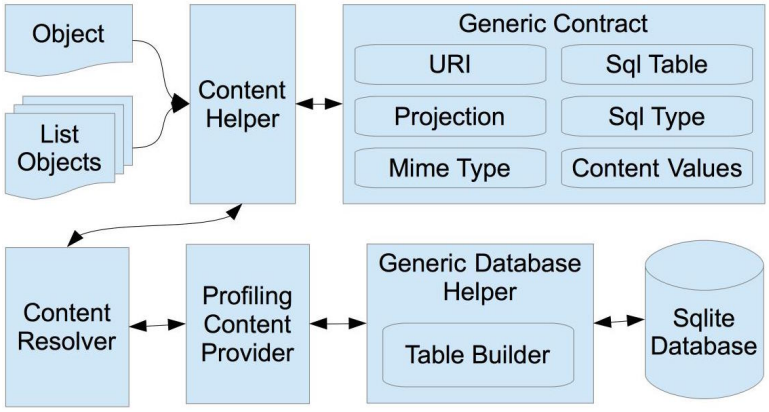

Figure 3.3: Inserting, updating, or deleting of a Java Object using the Content Helper.

Objects being inserted, updated, or deleted using the Content Helper can be seen in Figure 3.3. The process includes sending the objects through the Content Helper, using the Generic Contract to get the necessary information related to the incoming objects, using the Content Resolver to find the implemented Content Provider, and then finally updating the SQLite database with the help of the Generic Database Helper.

The insert(Object object) method accepts as an argument any object. Using reflection, the Content Helper will determine the class type of the object. If the class type is an array or Iterable then the function calls itself recursively over each item. At this point all objects inserted are independent of each other. If the object is of type Object then a private insert method (i.e. insert(Object object, String foreignKey, String primaryKeyValue)) is called which further deals with primary keys, foreign keys, and ContentValues. If the value of the object is null then the method returns, otherwise the Content URI and ContentValues object is retrieved using the Generic Contract. During insertion of a

parent object, the primary key and its value are determined using the Generic Contract. The Content Provider is then queried using the value of the primary key. If an entry already exists then it is updated, otherwise a new entry is inserted. After inserting or updating the object, all children objects are discovered and stored in ChildObject objects with their associated foreign keys. The insert method is then called recursively over each child object with its associated foreign key and the parent's primary key value.

Child objects are collected by iterating over every Field in the class type and determining whether or not they qualify for insertion. As usual, the object type is determined and will qualify for insertion if it is of type Object, array of Objects, or Iterable. In the case of an array of Objects or Iterable each of the objects contained will be added to the array of ChildObjects. As mentioned earlier, the ChildObject contains a reference to the initial object and it's foreign key which is provided by the TableBuilder.

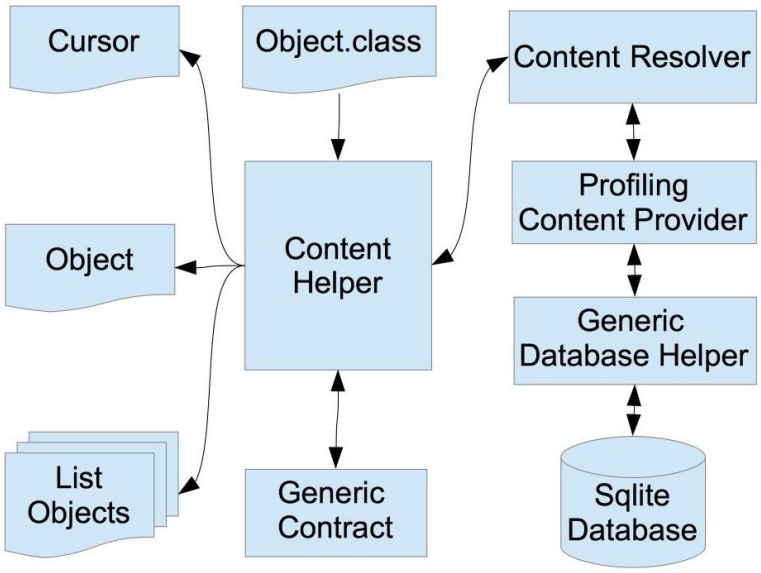

Figure 3.4: Query the Content Helper using a Java class type as input.

Querying data using the Content Helper can be seen in Figure 3.4. Querying is

different from the other CRUD operations because it is the only one which returns data. Instead of providing object references to the Content Helper, the query operation accepts a class type and the same parameters as the Content Provider query operation. The Content Helper then uses the Generic Contract to help return either the default Cursor normally returned by a Content Provider when querying data, a single object, or an array of objects. The Generic Contract can convert Cursor objects into single objects or an array of objects the same way that it converts an object into a ContentValues object using reflection.

Deletion and updating methods will work similarly to that of the insertion method. However, in this thesis the Generic Profiler is only interested in collecting new data and therefore implementation of the deletion and updating methods has been left for future work.

3.3 Profilers

All the profilers in this section utilize the Generic Profiling mechanism discussed in detail in the previous section. The profilers which will be discussed are a Software Profiler, Telephony Profiler, Wi-Fi Profiler, and a Battery Profiler.

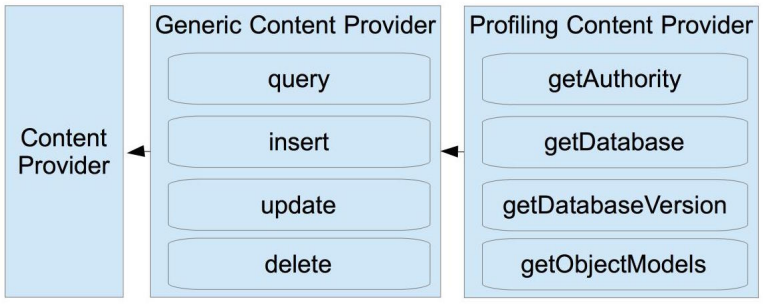

Figure 3.5: Overview of the Profiling Content Provider

Initial setup of the profiling project includes the implementation of the Generic Content Provider. The Profiling Content Provider, which can be seen in Figure 3.5, is provided and extends the Generic Content Provider where the abstract methods are further implemented. For simplicity, the base authority and database Strings are returned by getAuthority() and getDatabase() methods respectively. The class types returned by getObjectModels() include SoftwareProfile, NetworkProfile, WifiProfile, TelephonyProfile, BatteryProfile, and a few sub classes of TelephonyProfile (i.e. CallState, DataActivity, DataConnectionState, ServiceState, and SignalStrength). The value returned by getDatabaseVersion() is insignificant as it is incremented only when the developer wishes to migrate changes to the SQLite database.

Following an M/M/1 queuing system, the amount of time necessary to profile incoming data at time x can be seen in equation 3.1. In an M/M/1 queuing system arrivals follow the Poisson Distribution at a rate of α and the service time of each arrival is distributed using the Exponential Distribution at a rate of μ. The number of users in an M/M/1 queuing system is modeled by $N = \frac{\rho}{1-\rho}$ and the service time is modeled by $T = \frac{1}{\mu-\alpha}$. The service time of each profiler which monitors changes in the system can be modeled using this method. In equation 3.1 $\rho = \frac{\alpha}{\mu}$ and the resulting summation is a product of the number of arrivals and each service time at time x for the location profiler.

$$T_L(x) = \sum_{x=0}^{n} T_x N_x = \sum_{x=0}^{n} \frac{\rho}{(\mu - \alpha)(1 - \rho)} \tag{3.1}$$

The Wi-Fi, Telephony, and Software profilers can all be modeled similarly and result in $T_W(x)$, $T_T(x)$, and $T_S(x)$ respectfully. The total service time necessary to record all

data being profiled can be seen in Equation 3.2;

$$T_{Total}(x) = T_L(x) + T_W(x) + T_T(x) + T_S(x) \tag{3.2}$$

Furthermore, all numerical data collected by the profilers is eventually stored using an Estimated Weighted Moving Average (EWMA). This process is performed in the interest of hard memory on the device and can be used to make quicker decisions when combined with an offloading solution. In order to conserve energy the EWMA for each profiler is calculated during the time of a day when the device is known to be plugged in.

3.3.1 Location Profiling

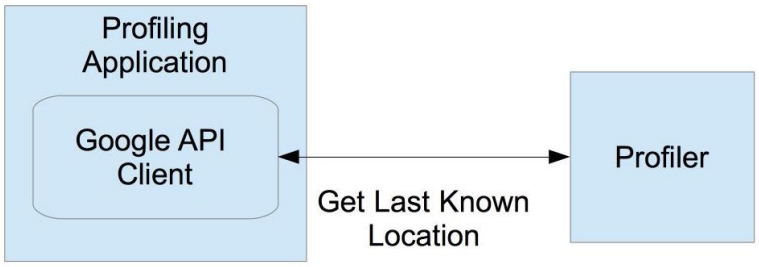

Figure 3.6: Overview of the Location Profiler

In order to maintain as low overhead as possible location profiling is performed only when it is needed. During the Android application initialization phase the ProflingApplication which extends the Android Application initializes an instance of the GoogleApiClient. The GoogleApiClient offers access to the GPS sensors on the mobile device and incorporates some location changing logic in order to maintain accurate locations. Once the ProfilingApplication has successfully connected to the GoogleApiClient any other profilers may request the last known location which can be seen in Figure 3.6. This

method allows the location to be retrieved only when it is needed and maintains only a single connection to the GoogleApiClient. Maintaining multiple connections and polling for location changes on a regular basis would result in a significant amount of energy consumption.

3.3.2 Software Profiler

Unlike many other solutions which require a developer to annotate methods for software profiling, the Software Profiler uses a tool called Traceview [38] which profiles every single method invocation. When the profiler is started Traceview immediately starts collecting data related to method and thread invocations and stores the results in a trace file called 'SoftwareProfile.trace'. Every thirty seconds a SoftwareProfilingService stops the Traceview and records the results of the trace file using the Content Helper. Upon completion of the service the Traceview is resumed. Figure 3.7 shows an overview of the Software Profiler using Traceview to collect useful information from worker threads and then saving the data using the Content Helper.

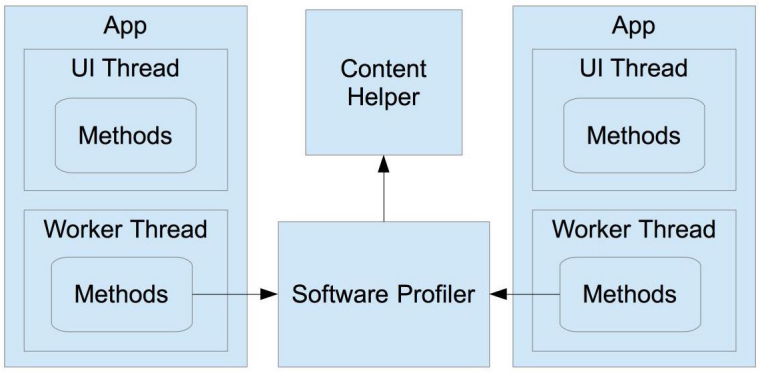

Figure 3.7: Overview of the Software Profiler

During TraceView diagnostics a SoftwareProfile object is created and initialized with

the current time-stamp. The trace file is opened and parsed with VmTraceParser. The parser can read the trace file format and return the trace data in the form of a VmTraceData object. Each method invocation on all threads is then extracted and used to populate the SoftwareProfile object. The properties included in the method information include the class name, full name, id, method name, short name, signature, source line number, source path, invocation count, exclusive time, inclusive time, exclusive percentage, inclusive percentage, and both the inclusive and exclusive battery consumptions. The trace data also includes options for retrieving the method tree if desired.

3.3.3 Telephony Profiler

Instead of running periodically like the Software Profiler, the Telephony Profiler remains persistent and collects data only when it changes. Android broadcasts an Intent containing telephony changes and state whenever a change has occurred. Android also provides a PhoneStateListener which listens for those specific Intents. The Telephony Profiler runs an instance of TelephonyListener which implements PhoneStateListener. The TelephonyListener registers and listens for broadcasts of type LISTEN_CALL_STATE, LISTEN_CELL_INFO, LISTEN_CELL_LOCATION, LISTEN_DATA_ACTIVITY, LISTEN_DATA_CONNECTION_STATE, LISTEN_MESSAGE_WAITING_INDICATOR, LISTEN_SERVICE_STATE, and LISTEN_SIGNAL_STRENGTHS. Furthermore, the TelephonyListener implements the methods onDataConnectionStateChanged(), onCallStateChanged(), onSignalStrengthsChanged(), onDataActivity(), and onServiceStateChanged(). Implementing each method ensures that only specific changes get recorded. The TelephonyListener however, maintains the state of the TelephonyProfile object and updates only the properties that change. When a change has occurred, the TelephonyListener uses the

ContentHelper to record the current state of the TelephonyProfile with the given time-stamp and last known location provided by the ProfilerApplication. Figure 3.8 shows an overview of the Telephony Profiler.

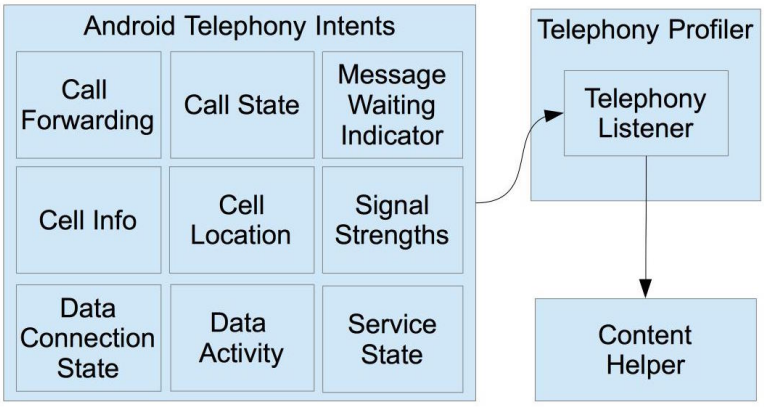

Figure 3.8: Overview of the Telephony Profiler

There are quite a few properties which get recorded by the TelephonyProfiler. They include a time-stamp, call state, data activity, data state, device id, software version, line number, network country ISO, network operator, network type, SIM country ISO, SIM operator, SIM state, subscriber id, voice mail number and alpha tag, an isRoaming boolean, and SMS and MMS information. For the purpose of this thesis only a few properties are considered, but profiling everything leaves opportunity for future work.

When a change to the telephony state has occurred and a callback is invoked a more categorized level of profiling is implemented. Objects which are profiled for each of the implemented PhoneStateListener callbacks include CallState, DataConnectionState, SignalStrength, DataActivity, and ServiceState. These changes in state are particularly helpful because they tend to revolve around network changes. The DataConnectionState object includes properties for changes to the current state of the network (i.e. connected, connecting, disconnected, or suspended) and the network type (i.e. 1xRTT, cdma, edge,

ehrpd, evdo0, evdoA, evdoB, gprs, hsdpa, hspa, hspap, hsup, iden, lte, umts, or unknown). The CallState includes properties for collecting the current state of a call (i.e. idle, offhook, or ringing) and the incoming phone number. SignalStrength collects information related to the type of signal, its GSM strength in dBm, and the GSM bit error rate. It is also concerned in collecting information related to the CDMA and EVDO RSSI values in dBm and Ec/Io values as well as EVDO signal to noise ratio. The DataActivity object includes only the direction in which data is traveling from or to the device (i.e. dormant, in, out, inOut, or none). Finally the ServiceState object includes properties for whether or not the service provider was selected manually or automatically, the current state of service (i.e. emergencyOnly, inService, outOfService, powerOff), and the registered operator name.

3.3.4 Wi-Fi Profiler

The Wi-Fi Profiler works similarly to the Telephony Profiler, however Android does not provide a Listener interface for Wi-Fi state changes. Instead, we've provided a WifiChangeReceiver which extends the Android BroadcastReceiver which is designed to listen for Wi-Fi related Intent broadcasts. BroadcastReceivers are registered with Android and during the registration process an IntentFilter is applied. IntentFilters specify which Intent broadcasts to listen for by action name. The WifiChangeReceiver is registered with an IntentFilter which includes action names RSSI_CHANGED_ACTION, NETWORK_IDS_CHANGED_ACTION, NETWORK_STATE_CHANGED_ACTION, SCAN_RESULTS_AVAILABLE_ACTION, SUPPLICANT_CONNECTION_CHANGE_ACTION, SUPPLICANT_STATE_CHANGED_ACTION, and WIFI_STATE_CHANGED_ACTION. When a broadcast is received it contains Intent extras regarding the BSSID, RSSI,

whether a connection to the supplicant daemon is established, a supplicant error code if it exists, the Wi-Fi state, and a NetworkInfo object. An overview of the Wi-Fi Profiler can be seen in Figure 3.9.

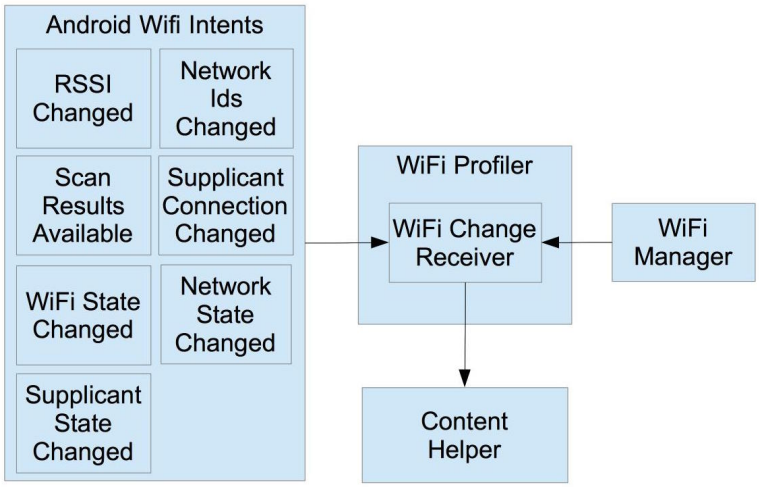

Figure 3.9: Overview of the Wi-Fi Profiler

The NetworkInfo object is used to describe the status of a network interface. Data collected from the NetworkInfo object includes a detailed or fine-grained state of the network (i.e. idle, scanning, connected, authenticating, obtaining IP address, connected, suspended, disconnecting, disconnected, failed, blocked, verifying poor link, or captive portal check), a string containing any extra information about the lower networking layers, a reason why an attempt to establish a connection failed if it failed, a coarse-grained state of the network (i.e. connecting, connected, suspended, disconnecting, disconnected, or unknown), the type and subtype of the network, and some booleans which signify whether the network is available, connected, connected or connecting, if there was a failover, or if the device is currently roaming. All properties provided by the NetworkInfo object and the Intent extras are combined into a single WifiProfile object.

An instance of the WifiManager, which is an Android system service, is also accessed when an Intent broadcast related to a Wi-Fi change has been received. The WifiManager offers a much more detailed view of the network state. Useful information collected from the WifiManager are mainly booleans representing whether Wi-Fi is enabled, if the 5Ghz band is supported, if the device supports device-to-access-point RTT, if the device supports advanced power or performance counters, if P2P is supported, if Wi-Fi scanning is always available, and if the device supports Tunnel Directed Link Setup. The Wi-Fi Manager also includes a WifiInfo object which represents the current state of the Wi-Fi network. The WifiInfo object includes properties for network frequency, IP address, link speed, MAC address, network id, SSID, RSSI, and a boolean representing whether or not the SSID is hidden. Location is once again retrieved from the ProfileApplication and the current time-stamp are set in the WifiProfile object and then recorded using the Content Helper. Once again, for the purpose of this thesis only a few criteria are used, but much more are collected to leave more opportunity for future work.

3.3.5 Battery Profiler

Android also broadcasts any changes in battery state. The Battery Profiler works the exact same way as the Wi-Fi Profiler. Unlike the Wi-Fi Profiler which listens for broadcasts with many action types related to Wi-Fi changes, the Battery Profiler relies only on receiving broadcasts of type ACTION_BATTERY_CHANGED and can be seen in Figure 3.10. Once again, location and time-stamp are obtained and set in the BatteryProfile object which will be stored using the Content Helper. When an Intent is received it includes information regarding battery health (i.e. cold, dead, good, overVoltage, overHeat, unspecifiedFailure, or unknown), the resource id for the small icon used to indicate

current battery state, the current battery level, the plugged in state of the device (i.e. ac, usb, or wireless), a boolean representing whether or not a battery is present, a scale value representing the maximum battery level, the current status of the battery (i.e. charging, discharging, full, notCharging, or unknown), a string describing the technology of the battery, the current temperature of the battery, and the current voltage level.

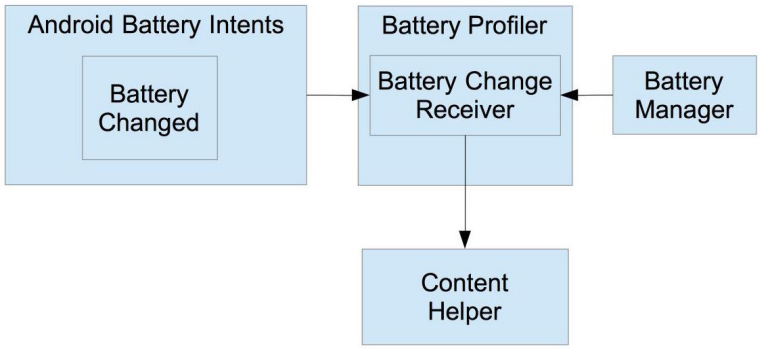

Figure 3.10: Overview of the Battery Profiler

Android also includes a system service for battery information called the BatteryManager. Similar to the WifiProfiler, the BatteryManager is used to collect more detailed battery information when ever a battery change Intent is received. The BatteryManager offers the remaining battery capacity as a percentage, the battery capacity in microampere-hours, the average battery current in microamperes where positive values indicate net current entering the battery and negative values indicate net current discharging from the battery, the instantaneous battery current in microamperes, and the remaining energy in nanowatt-hours. After having collected all the available battery information, the BatteryProfile object is stored using the Content Helper.

3.4 Summary

In this chapter we have implemented a Generic Profiler which is designed to solve the data profiling problem in regards to mobile computation cloud offloading. The Generic Profiler is composed of five individual components including a Generic Content Provider, Generic Contract Class, Generic Database Helper, Table Builder, and a Content Helper. Using reflection, the Generic Profiler can be used to record data related to any object model provided to the solution. This advantage makes it unnecessary for third party developers to implement Contract Classes or specific Content Provider logic based on individual object models. We have also included a series of profilers which include a Location Profiler, Software Profiler, Telephony Profiler, Wi-Fi Profiler, and a Battery Profiler. Each of the profilers take advantage of the Generic Profiler tool for storing data.

Chapter 4

Evaluation

4.1 Introduction

In this chapter, we will discuss in detail the development effort and performance of the Generic Profiler and the profilers included in this thesis. A comparison is made between the amount of development effort in terms of lines of code required to implement a Content Provider and the effort involved when implementing the Generic Content Provider. Another comparison is also made between the development effort, once again in terms of lines of code, involved when creating Contract Classes for custom data models and the Generic Contract Class. We further discuss the development effort involved in fully implementing a profiler from scratch verse one being implemented using a Generic Profiler. Finally, an overview of the effort involved in building the Generic Profiler is provided. We also show performance metrics, in terms of inclusive invocation time, of the Generic Profiler over an NQueens algorithm and evaluate the overall performance of the profilers themselves.

4.2 Development Effort

The largest benefit of the Generic Profiler is that it can be used by third party developers to solve their data profiling needs. As mentioned earlier, developers would normally have to write repetitive code for each Contract Class being included in the Content Provider and SQLite database. This repetitive code can lead to a significant amount of wasted effort. Here we will discuss the improvement of developer effort offered when using the Generic Profiler tool.

4.2.1 Content Providers

It is not uncommon for developers who wish to store data in a SQLite database to implement their own Content Provider. However, implementing a Content Provider from scratch requires research, planning, and time. Figure 4.1 shows a comparison between the lines of code required to implement a Content Provider for typical usage and the Generic Content Provider provided in our solution. Implementing the Content Provider on average uses approximately 225 lines of code, but only 38 lines of code is used to implement the Generic Content Provider. There is clearly a benefit when implementing the Generic Content Provider as it requires 74 percent less code than implementing a Content Provider. One does not have to write much code when implementing the Generic Content Provider because the Generic Content Provider has already used reflection to implement the CRUD operations of a Content Provider. The only methods which need to be implemented are the abstract methods mentioned in the previous chapter.

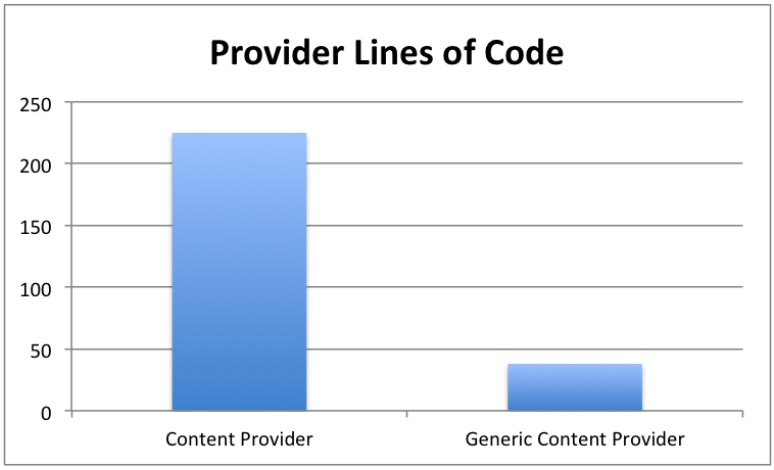

Figure 4.1: Comparison between lines of code needed to implement the provided Generic Content Provider vs. the Content Provider.

4.2.2 Contract Classes

Writing the code necessary for each contract class can also be a time consuming process. When a developer chooses to store more than one data model using a Content Provider the amount of code can increase quite quickly. This can be quite a taxing and repetitive process as most Contract Classes follow a similar design. The design usually includes methods for retrieving a Content URI, projection, authority, column name constants, create table query, drop table query, Content Values from an object, Content Values from an array of objects, an object from a Cursor, and a array of objects from a Cursor. As mentioned in the previous chapter we are interested in profiling changes in location, battery, wifi, telephony, and software. Figure 4.2 shows the estimated lines of code required to build a Contract Class for each of these data models based on the standard Android development guidelines for developing Contract Classes. The function $7n + x$ provides this estimation where n is the number of fields in a class, x is an arbitrary integer to allow for error, and 7 is used because a Contract Class should return approximately

seven methods which are related to the number of fields in a class. Since the Generic Contract Class uses reflection, given the class definition as a parameter, it essentially returns all the same results as any implemented Contract Class. This results in 100 percent less lines of code for a third party developer to implement.

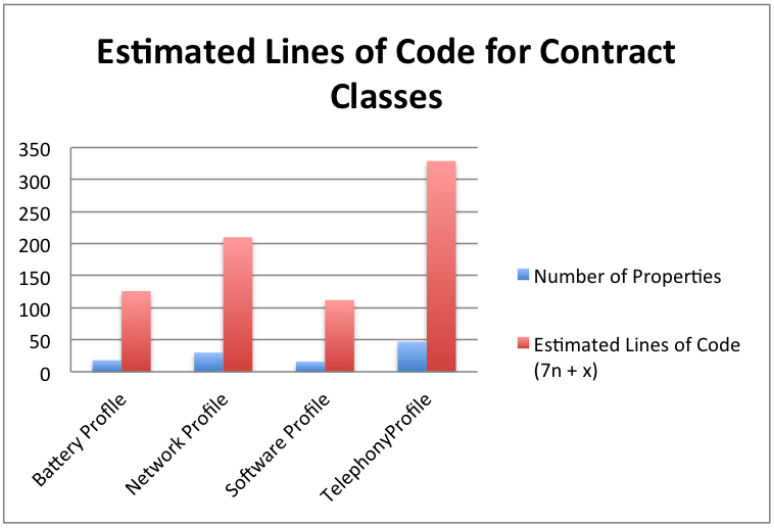

Figure 4.2: Estimation of lines of code needed to write Contract Classes for each of the provided Profilers.

4.2.3 Fully Implemented Provider

Having analyzed the amount of effort necessary to implement a Content Provider, Generic Content Provider, and Contract Classes we can now discuss the full implementation of a Provider. Figure 4.3 shows a comparison similar to 4.1 but includes the total amount of effort necessary for a full implementation. Combining the estimations provided for Contract Classes and the lines of code to implement a Content Provider we arrive at approximately 1002 lines of code to implement a fully functional provider. However, combining the 38 lines of code to implement the Generic Content Provider and the lines

of code required to build each model we wish to store (i.e. approximately the sum of the number of properties of all models) we arrive at merely 149 lines of code. That is approximately 85 percent less code than implementing a Content Provider from scratch.

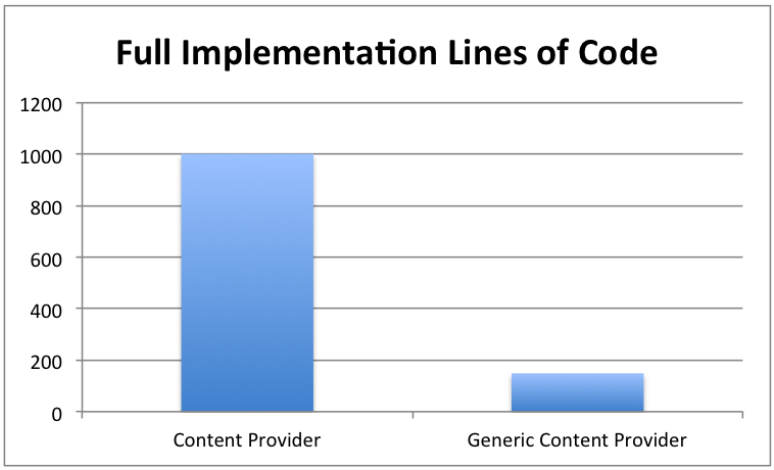

Figure 4.3: Comparison between lines of code needed to fully implement a Content Provider solution vs. the provided Generic Profiler solution.

4.2.4 Full Implementation of the Generic Profiler

As mentioned in the previous section, the Generic Profiler is composed of various classes. These classes are the Generic Content Provider, Generic Database, Generic Contract, Content Helper, Table Builder, a Constants file, a Table object, Column object, and a ChildObject class which is used to represent child properties of a class. Figure 4.4 shows the lines of code necessary to complete each component of the Generic Profiler. The sum of the lines of code for each section results in approximately 1195 lines of code in total. Since the solution relies on reflection in many places there is much less wasted lines of code that would be caused by repetitive procedures. Also, this tool is designed to be used by third party developers and will consequently reduce the amount of code necessary to

achieve the same task by at least 1195 lines of code.

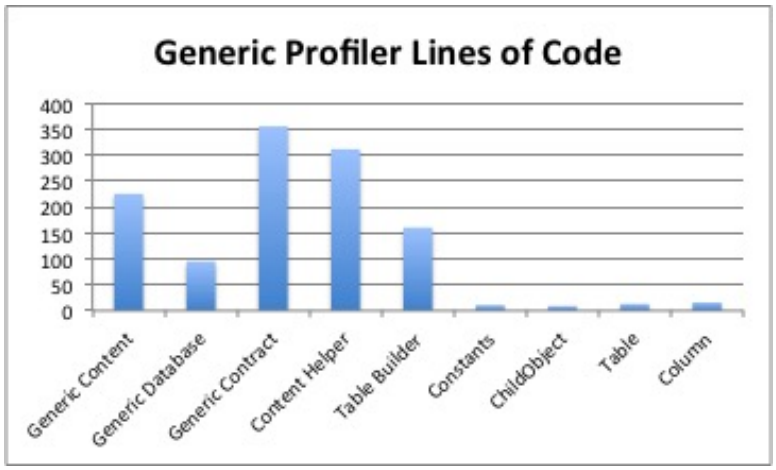

Figure 4.4: Lines of code written to fully build the Generic Profiler

4.3 Performance

4.3.1 NQueens Algorithm Evaluation

A popular algorithm used in the field of mobile cloud offloading for evaluation purposes is the NQueens algorithm. The Algorithm is a backtracking algorithm and will return the number of solutions to the game using an arbitrary integer as input. The input parameter, n, represents the number of queens on an n by n board. The average inclusive invocation times of the NQueens algorithm has been profiled using the Generic Profiler and broken into three tables for conveniently comparing results. Figure 4.5 shows the average inclusive invocation time of the NQueens algorithm where $0 \leq n \leq 10$ queens as input. Each average is composed of 100 independent trials for each input parameter n. It can be seen that the invocation time of the algorithm where $n \leq 10$ is very minimal.

The average invocation times in milliseconds are 0.02, 0.02, 0.1, 0.05, 0.05, 0.05, 0.05, 0.04, 0.2, 0.33, and 2.05 for $0 \leq n \leq 10$ respectfully. At times, the algorithm appears to perform better when there are more queens on the board than others (i.e. iterations 3, 4, 5, 6 vs. iteration 2).

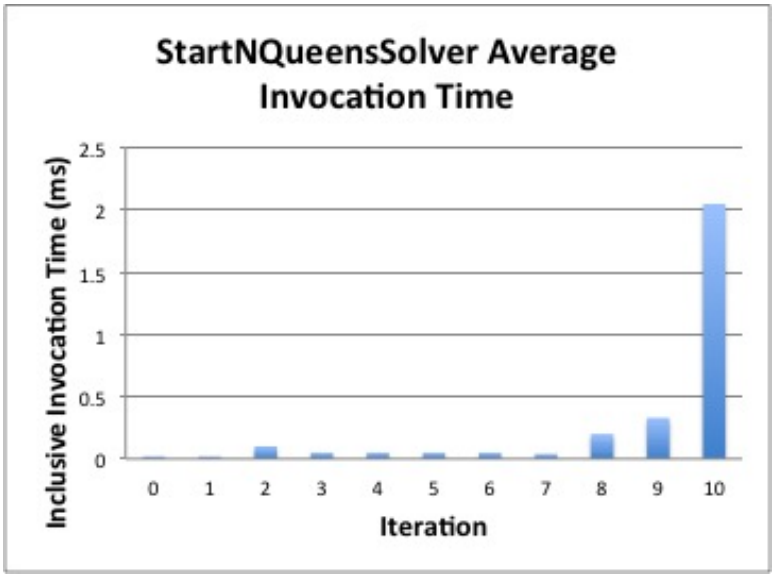

Figure 4.5: Average inclusive invocation time of the NQueens algorithm for 0 to 10 queens

Figure 4.6 shows the results where $11 \leq n \leq 13$ and the average invocation times in milliseconds are 3.87, 17.08, and 91.26 respectfully. Once again, these invocation times are still quite quick.

When trying to solve the NQueens problem where $n \geq 14$ we start to see performance issues. Figure 4.7 shows the average invocation times of the NQueens algorithm where $14 \leq n \leq 15$. The invocation times in milliseconds being 414.84 and 2744.53 respectfully. When $n = 15$ the average invocation time is almost 3 seconds and running the algorithm 100 times results in 300 seconds or 5 minutes. At this point, it becomes quite inefficient to run the algorithm locally on the mobile device and one must consider offloading. The

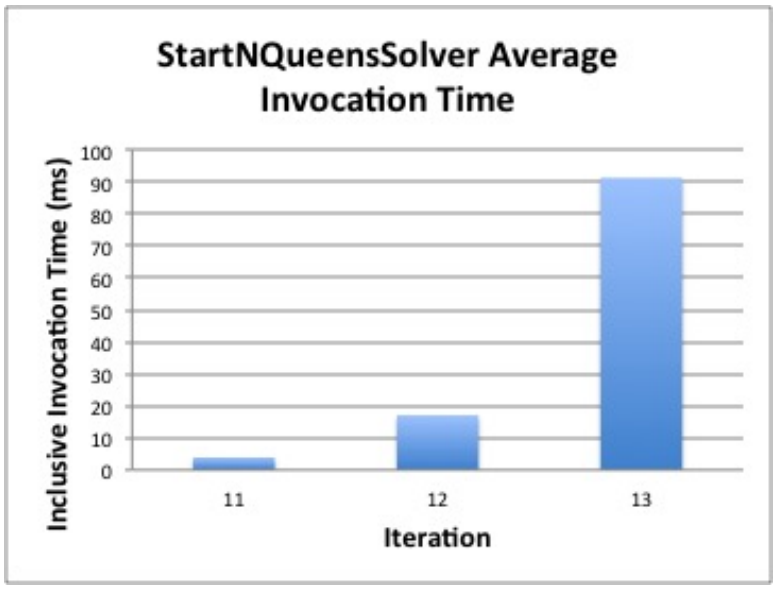

Figure 4.6: Average inclusive invocation time of the NQueens algorithm for 11 to 13 queens

data provided by the Software Profiler can be of vital importance to solving the offloading problem.

The Software Profiler has the added bonus of recording the total invocation time of itself even while it is profiling the NQueens algorithm. Figure 4.8 shows the average inclusive invocation time of the Software Profiler while it is profiling the NQueens algorithm. That is, for each iteration where $0 \leq n \leq 15$, the profiler has monitored the inclusive invocation time of the Content Helper inserting the profiling data for each object individually. The invocations times in seconds are approximately 2.4, 1.7, 2.3, 2.3, 1.7, 1.0, 3.0, 1.6, 2.0, 1.9, 1.8, 2.1, 1.2, 2.4, 1.0, and 0.9 where $0 \leq n \leq 15$ respectfully. Since the profile objects (i.e. 100 per iteration of n) were inserted rapidly and one at a time, the Content Helper takes a performance hit. However, the fairly random results varying from 0.8 seconds to 3.0 seconds indicate that other background processes may

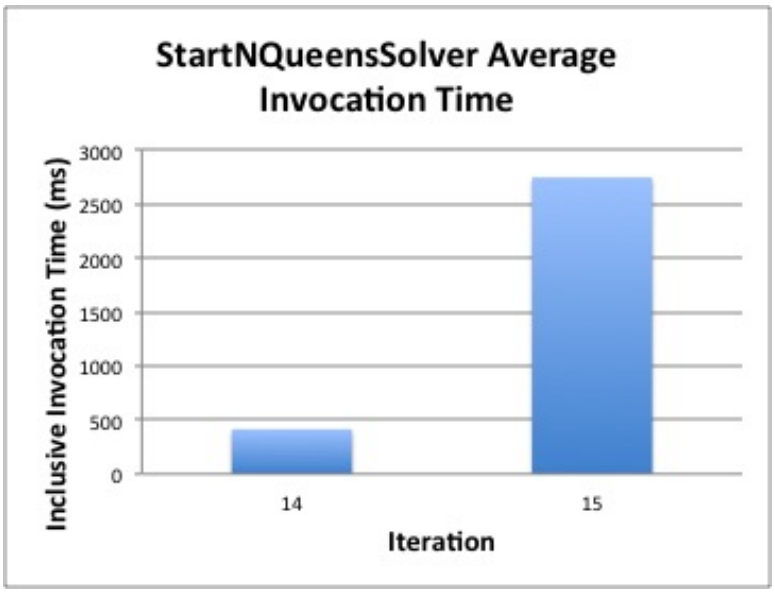

Figure 4.7: Average inclusive invocation time of the NQueens algorithm for 14 to 15 queens

interfere with the invocation time of the profiling process, but the invocation time of the method being profiled does not affect the profiling time. Otherwise, the profiling results would show a fairly linear pattern.

4.3.2 Profiler Evaluation

As mentioned earlier, one of the benefits of the Software Profiler is that it has the capability of profiling the invocation time of other profilers also. Figure 4.9 shows the average inclusive invocation time of the Battery Profiler, Software Profiler, Telephony Profiler, Wi-Fi Profiler, and the Location Profiler. The invocation times in nanoseconds are 1134, 3072, 1239, 1060, and 4085 respectfully. These averages were taken over approximately 30 minutes of runtime. Both the Software Profiler and the Location Profiler take noticeably longer to perform because they deal with software and hardware.

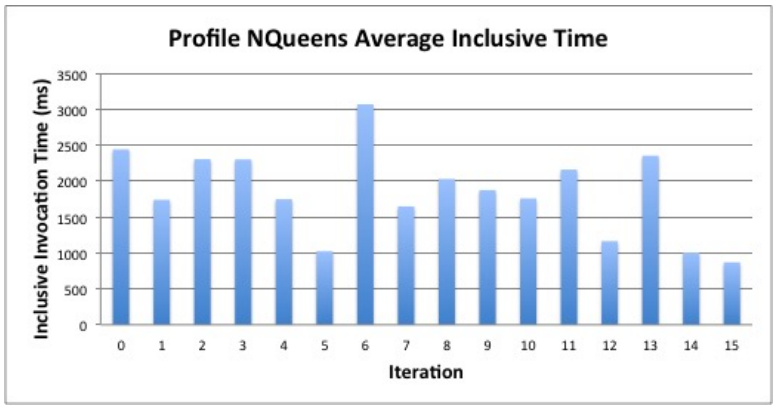

Figure 4.8: Average inclusive invocation time of the Software Profiler for the NQueens algorithm(One method at a time)

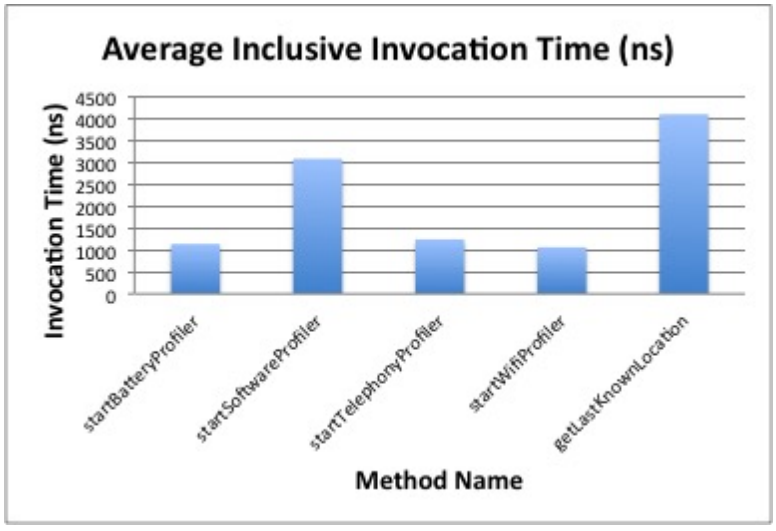

Figure 4.9: Average inclusive invocation time of the profilers

The average invocation times shown in Figure 4.9 are taken over the number of invocations shown in 4.10. It can be seen that the invocations counts of the Wi-Fi Profiler, Battery Profiler, Software Profiler, Telephony Profiler, and Location Profiler are 2, 2, 47, 2, and 22 respectfully.

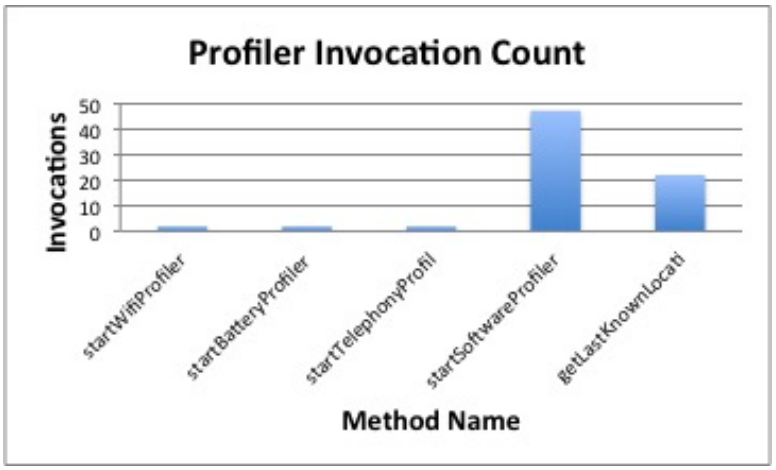

Figure 4.10: Invocation count of the profilers

Although we are mostly interested in the performance of the profilers in regards to the NQueens algorithm, the Software Profiler was designed to record all application methods. Figure 4.11 shows the methods profiled in nanoseconds where invocation time $t \leq 1200$. A total of 23 application methods with invocation time less than 1200 nanoseconds were profiled during the 30-minute execution.

Figure 4.12 shows all application methods which were profiled during the same 30 minutes of execution but under the conditions $1200 < t \leq 12,000$ where t is once again the invocation time in nanoseconds.

Figure 4.13 shows the application methods which were profiled under the conditions $12,000 < t \leq 60,000$. We notice that there are fewer methods because many of the faster methods happen to be children of the longer process methods.

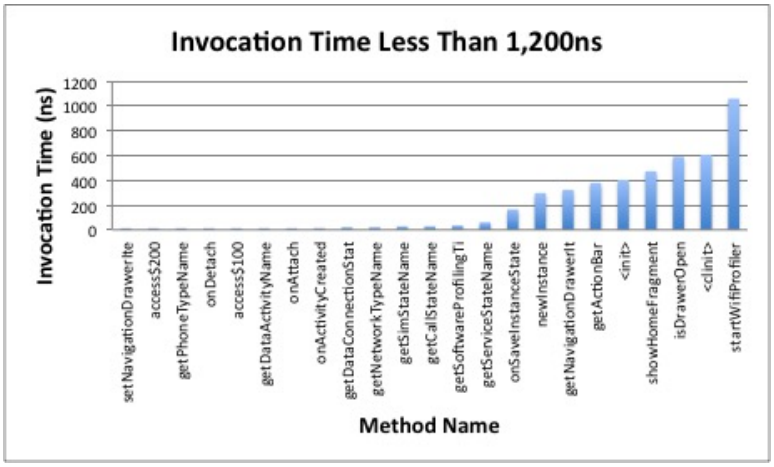

Figure 4.11: Average inclusive runtime up to 1,200ns of methods collected during 30-minute profile

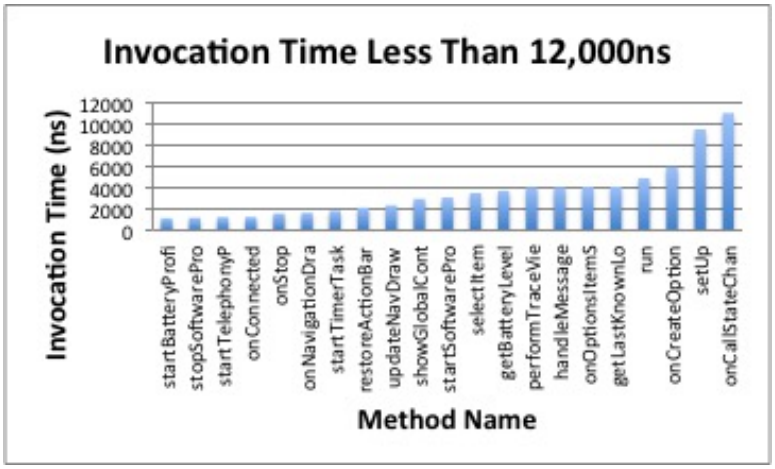

Figure 4.12: Average inclusive runtime up to 12,000ns of methods collected during 30-minute profile

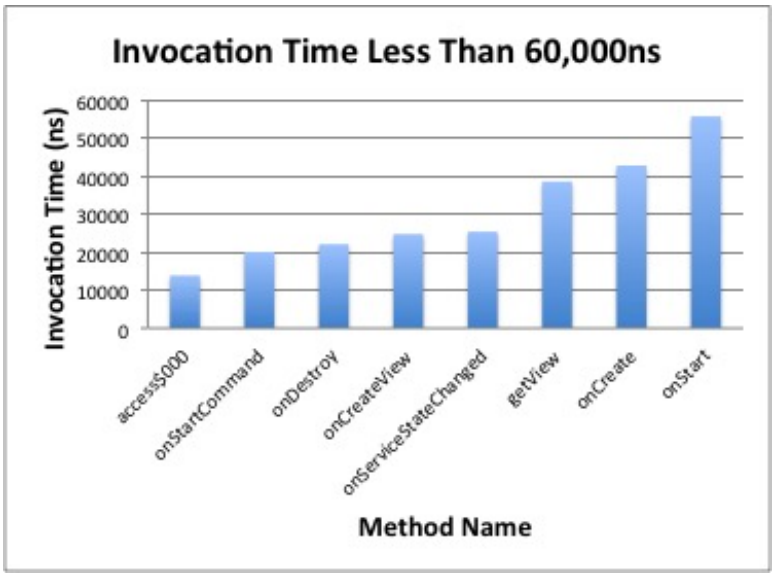

Figure 4.13: Average inclusive runtime up to 60,000ns of methods collected during 30-minute profile

Finally, we can see the methods which follow the conditions $60,000 < t \leq 14,000,000$ in Figure 4.14. Here we can see that most of the computational effort is performed in a background thread. The background thread encompasses the NQueens algorithm which is the primary tool used for evaluation.

4.4 Summary

In this chapter we have shown an evaluation of the development effort involved in both building and implementing the Generic Profiler and compared it to that of the effort involved in building a Content Provider and profilers from scratch. We have seen that by implementing the Generic Content Provider as opposed to the Content Provider a developer can write 74 percent less code. Also, relying on the Generic Contract Class instead of writing contract classes for individual data models saves the developer 100 percent of

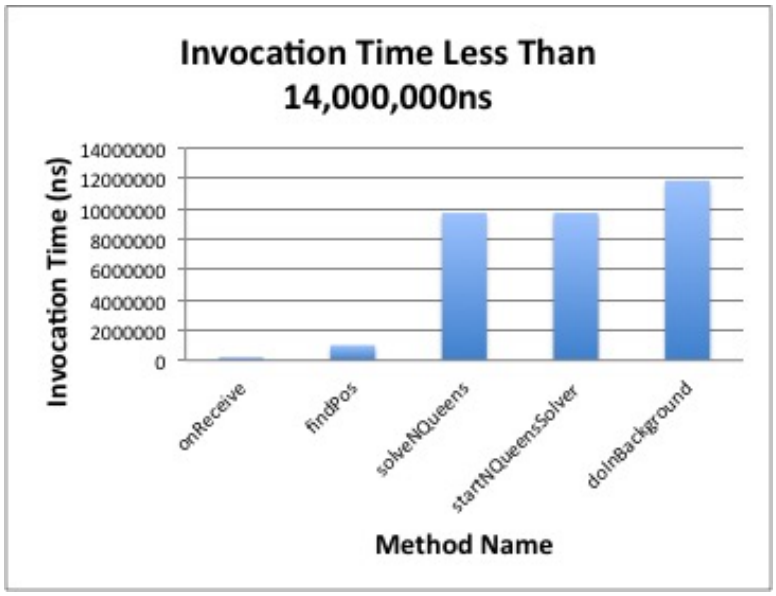

Figure 4.14: Average inclusive runtime up to 14,000,000ns of methods collected during 30-minute profile

development effort. Finally, implementing the Generic Profiler as opposed to writing a custom Content Provider and contract classes can save the developer approximately 85 percent of development effort.

We have also seen the performance of the Generic Profiler over an NQueens algorithm. To get accurate results a software profile was performed over fifteen iterations of the NQueens algorithm. Each iteration was performed 100 times and the Software Profiler recorded each invocation of the NQueens algorithm one at a time. The Profilers were also analyzed during their regular usage where the Software Profiler relies on batch updates instead of single updates. It can be seen that the Generic Profiler works much more effectively during the batch update process. An evaluation of methods profiled during a short period is also provided to show the capability of the Software Profiler.

Chapter 5

Conclusion and Future Work

5.1 Contributions

The primary contribution of this thesis is to provide a tool which can be used to profile hardware, software, and environmental data from a mobile device to aid a mobile computation cloud offloading strategy. The profiling solution takes advantage of well developed concepts from Android development strategies towards data collection and storage using a Content Provider. However, we have shown that by using reflection in many cases that not only can we remove the need to develop Content Provider solutions for specific applications, but also provide a tool to third party developers which can be used out of the box. The Generic Profiler has been proved to simplify data profiling and the incorporation of custom profilers designed by third party developers. Another contribution to this thesis is a series of data profilers which include a Location Profiler, Software Profiler, Wi-Fi Profiler, Telephony Profiler, and a Battery Profiler. The incorporation of these profilers demonstrate the effectiveness of the Generic Profiler when it comes to including newly implemented profilers. Of these Profilers the Software Profiler was ana-

lyzed in depth and shown to accurately record method usage of an NQueens algorithm as well as the other profilers. We have seen that the inclusive invocation time of the software profiler performing over batches of entries, and all other profilers discussed, is in the nanoseconds. However, the inclusive invocation time of profiling the same amount of data one entry at a time is in the milliseconds. This shows that the profilers have significantly little overhead when data is being profiled in batches, but there is room for improvement when inserting data one entry at a time.

5.2 Future Work

The effectiveness of the Generic Profiler leaves a significant amount of opportunity for future work. For the sake of this thesis, we have only implemented the insertion and query portions of the Generic Profiler. This is because currently the only necessary requirements of profiling data are adding new results and collecting the results using a query. However, there may be future cases in which one wishes to update or delete specific data that has been previously profiled. We have also found that the Generic Profiler works effectively for batched operations, unfortunately it still has room for improvement when profiling a significant amount of data one object at a time. Also, the profilers included in this thesis contribute a significant amount of data which can be used to aid an offloading decision strategy. However, there are many more profilers that have been mentioned in Chapter two which can also be included. Future doctoral work could include the contribution of a computation offloading strategy which relies on the Generic Profiler and a data synchronization scheme between the client and cloud.

References

[1] K. Elgazzar, P. Martin, and H. Hassanein, "Cloud-assisted computation offloading to support mobile services," *Cloud Computing, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[2] L. R. Janna Anderson, Elon University, "Technical seminar report on cloud computing," *Intel Executive Summary*, 2005.

[3] L. Wang, R. Ranjan, J. Chen, and B. Benatallah, *Cloud computing: methodology, systems, and applications*. CRC Press, 2011.

[4] "Software as a service: strategic backgrounder," *Software and Information Industry Association*, 2001.

[5] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pp. 44–51, Ieee, 2009.

[6] U. Sait, "The future of cloud computing," *Intel Executive Summary*, 2005.

[7] U. Sait, "Welcome to www.cloudtutorial.com," *Intel Executive Summary*, 2005.

[8] "Google app engine." `https://cloud.google.com/appengine/docs`.

[9] "Amazone web services." `http://aws.amazon.com/`.

[10] "Google compute engine." `https://cloud.google.com/compute/docs/`.

[11] H. Wu, Q. Wang, and K. Wolter, "Tradeoff between performance improvement and energy saving in mobile cloud offloading systems," in *Communications Workshops (ICC), 2013 IEEE International Conference on*, pp. 728–732, June 2013.

[12] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *INFOCOM, 2012 Proceedings IEEE*, pp. 945–953, March 2012.

[13] E. Chen and M. Itoh, "Virtual smartphone over ip," in *World of Wireless Mobile and Multimedia Networks (WoWMoM), 2010 IEEE International Symposium on a*, pp. 1–6, June 2010.

[14] S.-H. Hung, C.-S. Shih, J.-P. Shieh, C.-P. Lee, and Y.-H. Huang, "An online migration environment for executing mobile applications on the cloud," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pp. 20–27, June 2011.

[15] M. Barbera, S. Kosta, A. Mei, V. Perta, and J. Stefa, "Mobile offloading in the wild: Findings and lessons learned through a real-life experiment with a new cloud-aware system," in *INFOCOM, 2014 Proceedings IEEE*, pp. 2355–2363, April 2014.

[16] T.-Y. Lin, T.-A. Lin, C.-H. Hsu, and C.-T. King, "Context-aware decision engine for mobile cloud offloading," in *Wireless Communications and Networking Conference Workshops (WCNCW), 2013 IEEE*, pp. 111–116, April 2013.

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

[18] L. Pu, J. Xu, X. Jin, and J. Zhang, "Smartvirtcloud: Virtual cloud assisted application offloading execution at mobile devices' discretion," in *Wireless Communications and Networking Conference (WCNC), 2013 IEEE*, pp. 4398–4403, April 2013.

[19] M. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *INFOCOM, 2013 Proceedings IEEE*, pp. 1285–1293, April 2013.

[20] D. Chae, J. Kim, J. Kim, J. Kim, S. Yang, Y. Cho, Y. Kwon, and Y. Paek, "Cmcloud: Cloud platform for cost-effective offloading of mobile applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pp. 434–444, May 2014.

[21] "Content Provider managing access to a structured set of data in android." `http://developer.android.com/guide/topics/providers/content-providers.html`.

[22] F. Xia, F. Ding, J. Li, X. Kong, L. T. Yang, and J. Ma, "Phone2cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing," *Information Systems Frontiers*, vol. 16, no. 1, pp. 95–111, 2014.

[23] R. G. Brown, *Smoothing, forecasting and prediction of discrete time series*. Courier Corporation, 2004.

[24] L. Burgstahler and M. Neubauer, "New modifications of the exponential moving average algorithm for bandwidth estimation," in *Proc. of the 15th ITC Specialist Seminar*, 2002.

[25] Y. Wen, W. Zhang, and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones," in *INFOCOM, 2012 Proceedings IEEE*, pp. 2716–2720, March 2012.

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

[27] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh, "Who killed my battery?: Analyzing mobile browser energy consumption," in *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, (New York, NY, USA), pp. 41–50, ACM, 2012.

[28] "PowerTutor a power monitor for android-based mobile platforms." `http://ziyang.eecs.umich.edu/projects/powertutor/`.

[29] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic, *Digital integrated circuits*, vol. 2. Prentice hall Englewood Cliffs, 2002.

[30] D. Kovachev, T. Yu, and R. Klamma, "Adaptive computation offloading from mobile devices into the cloud," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pp. 784–791, July 2012.

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

[32] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. PP, no. 99, pp. 1–1, 2011.

[33] S. Tasnim, M. Chowdhury, K. Ahmed, N. Pissinou, and S. Iyengar, "Location aware code offloading on mobile cloud with qos constraint," in *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*, pp. 74–79, Jan 2014.

[34] "VirtualBox virtualization environment." `http://www.virtualbox.org`.

[35] G. Lee, H. Ko, and S. Pack, "Proxy-based aggregated synchronization scheme in mobile cloud computing," in *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pp. 187–188, April 2014.

[36] "Linux "inotify" tool." http://linux.die.net/man/7/inotify.

[37] P. A. Tridgell, "The rsync algorithm," *The Australian National University, Tech. Rep. TR-CS-96-05*, 1996.

[38] "Traceview profiling with traceview and dmtracedump." `http://developer.android.com/tools/debugging/debugging-tracing.html`.