COMPUTATION OFFLOADING AND STORAGE AUGMENTATION FOR MOBILE DEVICES

by

Mohammed A. Hassan

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
(Computer Science)

Committee:

_____	Dr. Songqing Chen, Dissertation Director
_____	Dr. Fei Li, Committee Member
_____	Dr. Robert Simon, Committee Member
_____	Dr. Chaowei Yang, Committee Member
_____	Dr. Sanjeev Setia, Department Chair
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Fall Semester 2014
George Mason University
Fairfax, VA

Computation Offloading and Storage Augmentation for Mobile Devices

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Mohammed A. Hassan
Bachelor of Science
Bangladesh University of Engineering and Technology, 2006

Director: Dr. Songqing Chen, Associate Professor
Department of Computer Science

Fall Semester 2014
George Mason University
Fairfax, VA

Acknowledgments

I would like to thank the following people who made this possible:

Dr. Songqing Chen, Dr. Fei Li, Dr. Robert Simon, Dr. Chaowei Yang, and Dr. Qi Wei.

Table of Contents

List of Tables

List of Figures

Abstract

COMPUTATION OFFLOADING AND STORAGE AUGMENTATION FOR MOBILE DEVICES

Mohammed A. Hassan, PhD

George Mason University, 2014

Dissertation Director: Dr. Songqing Chen

The increasing popularity of mobile devices calls for effective execution of mobile applications. Due to the relatively slower CPU speed and smaller memory size, plenty of research has been conducted on properly splitting and offloading computing intensive tasks in mobile applications to external resources (e.g., public clouds). Prior research on mobile computation offloading has mainly focused on *how to offload*. Moreover, existing solutions require the application developers to specify the computation intensive segment of the applications beforehand. In addition, these solutions do not work transparently with existing applications. In this dissertation, we aim to design and implement a mobile application offloading framework by addressing these issues. For this purpose, we first design and implement a *transparent* offloading mechanism called `FaST`: a **F***r***a***mework for* **S***mart and* **T***ransparent mobile computation offloading*, to allow mobile applications to automatically offload resource-intensive methods to more powerful computing resources without requiring any special compilation or modification to the applications' source code or binary.

We also find the key features that can *dynamically* impact the response time and the energy consumption of the offloaded methods and thus design `Lamp`, a runtime **L***earning* **m***odel for application* **p***rofiler*, based on which we profile the local on-device and the remote on-server executions of the applications. To address the problem of *what to offload*, we design and implement an application partitioner, called `Elicit`, to **E***fficientl***l***y* **i***dentify* **c***omputation-***i***ntensive* **t***asks in mobile applications for offloading* in a dynamically changing environment by estimating the applications' on-device and on-server performance with `Lamp`.

In addition to the offloading demand for mobile applications, the increasing popularity of mobile devices has also caused the demand surge of pervasive and quick accessing files across different personal devices owned by a user. Most existing solutions, such as DropBox and SkyDrive, rely on centralized infrastructure (e.g., cloud storage) to synchronize files across different devices. Therefore, these solutions come with potential risks of user privacy and data secrecy. In addition, the consistency, synchronization, and data location policies of these solutions are not suitable for resource-constrained mobile devices. Furthermore, these solutions are not transparent to the mobile applications. Therefore, in this dissertation, we design and implement a system **V***irtually* **U***nify* **P***ersonal* **S***torage* (`vUPS`) for fast and pervasive accesses of personal data across different devices.

Chapter 1: Introduction

The year of 2013 had witnessed the smartphone sales exceeding 1 billion for the first time ever, with 1,004.2 million smartphones shipped worldwide [1]. In addition to that, vendors shipped a total of 281.5 million smartphones worldwide during the first quarter of year 2014, which is 28.6% more from the 218.8 million units in the first quarter of 2013 [2].

The pervasive usage of mobile devices has enabled fast growth of mobile applications. Compared to traditional mobile phones that are mainly used for voice communications, today a smartphone is capable of common tasks that were only possible on desktop computers, such as surfing the Internet, taking and editing pictures, gaming, document processing, etc. For example, mobile devices are now widely used for financial tracking [3]. Recently, Amazon released SDK for Android users [4] to develop mobile applications using Amazon cloud such as uploading images and videos to the Amazon cloud storage, and sharing game, movies, and other data among users.

However, mobile devices, albeit their fastly increasing CPU speed and memory size, are not as capable as modern desktop computers when running these applications. In particular, mobile devices are ultimately constrained by the limited battery supply and a prolonged computation process or a computation-intensive application can quickly exhaust the limited battery power. This is worsen by the industrial trend to equip more sensors on mobile devices for a wider scope of non-traditional applications, such as environment monitoring [5], health monitoring [6, 7], social applications [8, 9], which are often more computation-intensive. As mobile devices are constrained by resources, recently a lot of efforts have been made to augment mobile device capability. For example, prior research [10–14] has proposed to partition the application and offload computation-intensive segments to

more powerful counterparts such as servers [12,14] or cloned VMs [10,11], while application-specific offloading is considered in [13]. Experimental applications are emerging as well. For example, Amazon SILK browser [15] splits the browsing task between the server and the user hand-held devices. But the impact of the system environment factors in the total response time to a mobile user, on the offloaded applications has not been well investigated in these studies. Moreover, little attention has been paid on how to efficiently identify the resource intensive segments of an application (for offloading) to maximize the offloading benefits.

In addition to the demand of more computing capability, the increasing popularity of the mobile devices has also caused the demand surge of pervasive data accesses, such as for photos, across a user's different storage space, such as her iPhone and desktop computer. To respond to such a fastly increasing demand, there are many research and industry solutions [16–21] to augment the storage of the resource constrained devices. However, although providing pervasive access, these solutions are prune to the single point of failure and security breaches [22, 23]. Moreover, these solutions are not transparent to the applications with little consideration of the consistency, availability, and location policies of the data. Placing the data to a high latency-bound server may worsen the applications' performance, while maintaining strong consistency between the replicas over a low-bandwidth network may cause non-trivial overhead.

1.1 Computation Offloading

Current offloading schemes often require the applications or the binary executables to be modified [12, 14], or require special compilation [24]. In addition to that, these schemes either require the application developers to provide special notation of the resource intensive methods for offloading, or (implicitly) assume that the resource intensive methods are already known [12, 14]. Clone-based approaches [10] can offload applications without modifications to applications, but they require a full mobile image running on the cloud.

Regardless which approach is taken, a mobile user often desires the fastest response time and the minimum battery consumption. While prior studies show that a better response time and more energy consumption can be achieved by computation offloading, all of them conclude that the resource constrained mobile devices can benefit from offloading only when the offloading overhead is less than the local execution cost. Thus, always executing applications on the server side may increase the response time. Therefore, policies like Aliyun OS [25] where every computation is offloaded regardless of other conditions may actually lead to a prolonged response time.

Young et al. [14] suggested to offload the computation when the data size to transfer is greater than 6 MB. But in practice offloading decision depends on the network bandwidth and other conditions in addition to the data size. MAUI [12] adopts a linear regression model to make the offloading decision, but our experimental results (presented later) show that a linear regression model results in more than 50% wrong decisions, which leads to more energy consumption and a longer response time. Ionan et al. [26] and Odessa [13] proposed history based profiling of the applications running on the server and the smartphone, and leveraged the ratio of them to make the offloading decision. However, it is not practically possible to find the gain ratio in every possible combinations.

Regardless which offloading mechanism and decision maker is adopted, only the computation-intensive segments shall be offloaded when appropriate. However, identifying such segments can be challenging, which received little attention in the current research. Moreover, in practice, we can not offload every possible methods. Some methods may access to camera or other sensors of the mobile device, which cannot be offloaded and have to be executed on the mobile device. Considering these facts and constraints, in this dissertation, we first design and implement a transparent offloading mechanism for existing applications. We then profile the applications to find the resource-intensive unconstrained methods suitable for offloading. Based on the profiling and the system environment, we can offload the most appropriate computation-intensive methods that can achieve most response time reduction and energy savings.

1.2 Storage Augmentation

Besides the increasing demand of computing power from the external resources, the increasing popularity of mobile devices has also caused the demand surge of pervasive data accesses, such as for photos, across a user's different storage space, such as her iPhone and desktop computer. To respond to such a fastly increasing demand, there have been many research and industry solutions to augment the storage of the resource constrained devices. The centralized cloud or server-based approaches [27] [16] [17] typically require a user to store all files on the storage owned by the service providers, which risks data security, privacy, and availability. Some recent examples include Mark Zuckerberg's pictures leak incident in Facebook [28], DropBox account breach with wrong passwords [29], Amazon's data center failure in 2011 [30], etc. Modern file systems [31] [19] [32] have taken into account user's space (home computer, laptop, and smartphone) to avoid third party storage compromise. But these solutions maintain a strong consistency model for different types of files, resulting in unnecessary performance overhead. For example, smartphones are often associated with consuming and producing data which are mostly non-editable (photo, music, and video) and more than 27% pictures are taken by smartphones [33]. For these files, consistency may be relaxed as the contents are not frequently changed. In addition to that, these solutions do not define a policy for the location of the data in the server-side considering latency and usage. Moreover, these solutions are not transparent to the mobile applications and require the source code to be modified to leverage the systems.

1.3 Dissertation Contributions

In this dissertation, we aim to address these issues by designing and implementing effective schemes for computation offloading and storage augmentation for pervasive mobile devices. While details are provided later, our contributions in this dissertation include:

- To transparently offload computation-intensive parts of a mobile application, we first come up with a transparent offloading mechanism called `FaST`, (**F**r**a**mework for **S**mart

and **T**ransparent Mobile Application Offloading), through method interception at the Dalvik virtual machine (VM) level to allow mobile applications to offload their computation-intensive methods without requiring any special compilation or any modifications to the application's source code or binary executable. We further enhance the offloaded methods' performance by parallelism.

- Second, we design and implement `Lamp`, (**L**e**a**rning **m**odel for application **p**rofiling), to profile applications in a dynamically changing environment. For this purpose, we empirically identify pivotal features that impact the application execution time and design a run-time learning based model to profile the local on-device and the remote on-server execution times. It adopts a multilayer perceptron based learning model and dynamically takes into account the key features we have identified that affects the final offloading decision.

- Third, to identify proper method to offload, we further design a framework: `Elicit` (**E**fficient**l**y **i**dentify **c**omputation-**i**ntensive **t**asks in Mobile Applications for Offloading) to determine the set of methods that can be offloaded without any constraints. We propose an algorithm to dynamically choose an optimal partition of the applications for offloading based on the learning model in changing environments.

- Lastly, to augment the storage and support pervasive accesses of the mobile devices, we build `vUPS`, (**V**irtually **U**nify **P**ersonal **S**torage), a file augmentation system for mobile devices with cloud considering the availability and consistency. We implement `vUPS` in Android and evaluate it with Android applications and benchmarks to show that storage augmentation can significantly improve the performance of the mobile devices.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. After presenting some background and motivation in Chapter 2, we describe our offloading mechanism `FaST` in Chapter 3. We

present our learning model `Lamp` for application profiling in Chapter 4 and the optimal application partitioning model `Elicit` in Chapter 5. We discuss our storage augmentation model `vUPS` in Chapter 6 and conclude the dissertation in Chapter 7.

Chapter 2: Background And Motivation

As aforementioned, despite the increasing popularity, mobile devices have limited storage and computation power and there is plenty of research that aims to augment the storage and computation power of weaker devices with external resources, such as servers [34–37]. But existing solutions typically target networked or clustered computers with dedicated networking and guaranteed QoS with high availability. On the contrary, mobile devices are often connected with WiFi or 3G/4G, which do not guarantee the same connectivity as the wired network.

In this Chapter, we discuss the background and the current state-of-the-art of mobile cloud computing and storage. We first discuss in the motivations relevant to computation offloading in section 2.1, then we elaborate the motivation for a new storage augmentation system in section 2.2.

2.1 Computation Offloading

A computation offloading model should answer the following questions: i) how to offload, ii) what to offload, and iii) how to enhance the performance of the offloaded components. To answer the first question, we should first place a mechanism to transparently offload applications' methods to remote servers. Then, we should find the resource intensive segments of an application for offloading. A decision maker should determine the optimal partition of the applications for offloading depending on the dynamics of the environment. Then, the offloading model should further try to enhance the performance by expediting the response time.

2.1.1 Offloading Mechanism

Plenty of research has been conducted on how to offload the computation intensive part of mobile applications to more powerful counter-part. In general, the proposed schemes can be classified in two broad categories: i) offloading by application partitioning and ii) offloading by full VM cloning.

Research efforts [12–14, 38, 39] achieving offloading by application partitioning either switch the execution [13,38] or invoke RPC [12,14] when approaching any resource intensive segment in the code. Balan et. al. [38] show that applications' code can be modified to achieve cyber foraging, while some others [13, 40, 41] only focus on specific types of applications. RPC based schemes are more generic, but they require either annotation [12], binary modification [14], or special compilation [24], which requires modifications to every application so as to fit into the offloading framework.

On the other hand, VM based cloning maintains a full clone of the smartphone image in the cloud [10, 11, 42]. To offload computation intensive segment of the applications, Cloudlets [11] suspend the smartphone execution and transfer the execution in the VM clone in the Cloudlets. While transferring, it requires a lot of data transfer (∼100 MB) for synchronization. CloneCloud [10] and Comet [42] support thread level migration that do not require the smartphone to suspend while offloading. To support thread migration, the application should be partitioned in such a way that communication and synchronization overhead remains acceptable [12]. While offloading, the stack, heap, and instructions are transferred to the cloud for the thread to resume execution there. Any update needs to be synchronized instantly and sometimes such overhead could be high. For example, the offloading synchronization overhead for Linpack [43] in Comet [42] is over 1 MB. If simply the `linpack()` method is offloaded, the data transfer is only a few KB.

Thus, to minimize the offloading overhead and to support the large number of existing mobile applications that do not have offloading logic, we argue that a transparent offloading approach at the method level sometimes are more desirable. As a transparent scheme, it

would not require any modifications to the applications source code or binary executables or special compilation.

In addition to that, mobile applications are expected to have the minimum response time to enhance the user's experience and save the battery power. To minimize the response time to the user, parallel processing the offloaded method is a choice. After investigating some popular mobile applications, we find that some types of applications can be further parallelized after being offloaded [13, 41].

2.1.2 Learning Model for Application Profiling

While prior studies show that better response time and less energy consumption can be achieved with computation offloading, offloading is appropriate only when the offloading overhead and the server side execution cost is less than the local execution cost. Thus, always offloading [25] may lead to a prolonged response time instead of conserving it.

Young et al. [14] offload the computation when the data size is greater than 6 MB. But in practice we find that a notable number of resource intensive popular applications, such as ZXing [44] (Android Barcode reader app) and Mezzofanti [45] (Android optical character recognition app), always use an input of fixed size, typically ranging from 500 KB to 6 MB, where the offloading decision should also depend on the network bandwidth and other conditions. MAUI [12] makes offloading decisions based on a linear regression model among some selected features. In fact, our experimental results presented later show that a linear regression model results in more than 50% wrong decisions, which leads to more energy consumption and longer response time.

So to make a proper offloading decision, we need to know the local on-device and the remote on-server execution times of the method. But it is not possible to profile both the local on-device and the remote on-server executions for every method in every possible configuration. So we need a prediction model. To address this research challenge, we present the design and implementation of `Lamp`, **L**earning **m**odel for application profiling, to learn the on-device and on-server execution times gradually and estimate them to make

the best offloading decision accordingly. For this purpose, we empirically identify pivotal features that affect the applications' execution time. It adopts a multilayer perceptron based learning model and dynamically takes into account the key features that impact the execution time of the application. Thus, it assists the offloading decision maker to take the optimal decision while offloading in a dynamically changing environment.

2.1.3 Application Partitioning

We have mentioned that plenty of research has been conducted on how to efficiently offload computation-intensive tasks in a mobile application to more powerful counterpart [10–15, 42]. However, to implement the offloading, these models either require the application developers to provide special notation of the resource intensive methods for offloading, or (implicitly) assume that the resource intensive methods are already known [12, 14]. Clone-based approaches [10] can offload applications without modifications to applications, but they require a full mobile image running on the cloud, which brings high synchronization overhead [11, 42]. Such requirements prevent these models from being deployed in practice.

Making the offloading decision based on data size [14] may not necessarily offload the most computation-intensive methods. In practice, computation-intensive methods do not necessarily have direct relationship with the size of input data. Moreover, offloading the resource-intensive methods may not save energy and response time in every circumstance. MAUI [12] adopts the 0-1 integer linear programming (ILP) to solve the problem of offloading to maximize performance gain in terms of energy or response time. However, in practice, we can not offload every possible method. Some methods may access camera or other sensors of the mobile device, which cannot be offloaded and have to be executed on the mobile device locally. While choosing the methods to offload, we have to consider these constraints that are neglected in previous research.

Therefore, in this dissertation, we aim to offload such methods so that the offloading overhead is minimized, which comes from the time and the energy consumption for data sending to the server, and high performance gain is achieved by executing it on the more

powerful computer. To achieve this goal, we will formulate our problem, and map our application partitioning problem to the Project Selection Problem [46] in order to choose an optimal partition to offload methods.

2.2 Storage Augmentation

Distributed filesystems [36, 47, 48] have been a research focus in the field of distributed systems for decades. Large scale distributed file systems, such as Goolge File System (GFS) [35] and the Hadoop Distributed File System (HDFS) [49, 50], have also been well studied for distributed computing. Panache [51] also proposes improvement for parallel accesses. These traditional and research-oriented file systems are designed for local area network and desktop terminals, while in modern ages various mobile devices (e.g., smartphones) play a crucial role for users' data generation and access over public network.

More recent solutions, such as ZZFS [19], EYO [32], and Volley [21], proposed systems that can place data in a hybrid cloud to ensure more availability and consistency between the data. ZZFS [19] is a system to make data more available, while EYO [32] makes data available from offline devices as well in limited context. On the other hand, Volley [21] is a system to place data in a geo-distributed cloud system. BlueFS [52] has optimized the data placement policy where data updates were improved by EnsemBlue [53] from a centralized server to a peer-to-peer model. ZZFS [19] has specific data placement and the consistency policies to avoid conflicts, while Eyo [32] does not guarantee any consistency. PersonalRAID [54] provides device transparency for partial replication, which is not suitable for public network as it requires users to move a storage physically between the devices. Cimbiosys [55] provides efficient synchronization with minimized overhead. But it is not device transparent.

Overall, these studies mainly emphasize on how to keep data available and (more importantly) consistent between different replicas. Moreover, these solutions do not distinguish between the users' files and data according to their usage and access frequency. Therefore,

these solutions bring with a lot of network overhead affecting the overall performance.

2.2.1 Consistency and Availability

Mobile devices are exposed to dynamically changing environment. As a result, applying the same data placement, consistency, and replication policy for cloud computing may not be suitable for mobile devices. For example, cloud storage like Google Drive [56] uses a strong consistency policy, while DropBox [16] applies continuous consistency between the replicas. These models may cost 74KB of data transmission for even 1 byte of data transfer in our experiments, which is expensive for limited data-plan users and bandwidth-constrained mobile devices. Observing the fact that strong consistency is not required for every type of files, we propose different replication and consistency policies for different types of files to minimize the overhead and increase the data access performance. We divide files into two broad categories based on their access frequency. Different consistency policies are used for them to minimize the overhead.

Chapter 3: FaST: Offloading Mechanism

3.1 Introduction

Plenty of research [10,12–14,38,39,42] has been conducted on how to offload as well as what to offload. Balan et. al. [38, 39] proposed to modify existing applications for computation offloading. They emphasized that developers can revise and inject the offloading mechanism into existing code with ease. Similarly, MAUI [12] provided an API for the programmers for offloading. By annotating a method, the developer can offload it. But these proposed solutions require the source code to be modified. Young et.al. [14] also suggested RPC based solution for offloading. In fact, they modified the binary executables of existing applications and injected RPC logic for offloading, which also requires specific modifications to each application. Kosta et.al. [24] also proposed a solution with special compilation, which requires applications to be developed and compiled in a unique way. Some other studies [13, 40] are application specific and they do not provide any guideline for generic usage. For example, Odessa [13] showed that parallelization can expedite response time, while they developed their own applications to meet the specific requirement for evaluation. As a result, these proposed solutions can not be generalized for all the existing applications.

On the other hand, VM based cloning maintains a full clone of smartphone images in the cloud [10, 11, 42] and allows transparent offloading. For example, Cloudlets [11] suspends the smartphone execution and transfers it to the VM clone in the Cloudlets. In short, while offloading, it calculates the difference between the server-side and the smartphone images and transfers the difference as an overlay. As a result, it requires a lot of data transfer (∼100 MB) for synchronization.

CloneCloud [10] and Comet [42] support thread level migration with the complete image cloning. It can offload the resource intensive methods to the cloud image. But migrating

a thread requires all the states to be transferred to the server, which incurs too much overhead. This is confirmed by Comet [42] itself. In the experimental results, it showed that the synchronization overhead for offloading simple applications is sometimes over hundreds of megabytes.

Comet [42] also proposes to migrate the thread when its local on-device execution time exceeds the round trip time to the server. But before that, it requires continuous synchronization with the server. In case of not offloading a thread, this whole synchronization overhead is wasted. In addition to that, the thread level migration needs separate cloning for each of the users. If the same method is offloaded by multiple users, multiple separate copies have to be maintained and synchronized in the server. But with the method level offloading, the same method can be executed with different parameters from different users, thus keeping the server overhead minimum. Moreover, not every thread can be offloaded. Threads accessing I/O, camera, and other sensors of the mobile device should not be offloaded to the server. A single access to these restricted resources can deter the whole thread from being migrated, although this access may not be related to the computation-intensive segment (or method) of the application. While in the method level offloading, we can constrain only methods what are accessing the restricted resources, thus allowing more flexibility for offloading. Therefore we argue that instead of the thread level migration, the method level offloading allows us to find the resource-intensive segments and profile them to find the constraints from offloading. In addition, the method level offloading allows us less overhead and it does not cause unnecessary synchronization.

So to support the large number of existing mobile applications that do not have offloading logic, a transparent offloading approach at the method level sometimes may be more desirable. As a transparent scheme, it should not require any modifications to the applications source code or binary executables or special compilation. Therefore, in this Chapter, we propose to design and build `FaST`, (A **Fra**mework for **S**mart and **T**ransparent Mobile Application Offloading Mechanism) to transparently offload methods of mobile applications. While details are presented in later sub-sequences, our contributions in this Chapter are:

- Different from previous studies, FaST allows mobile application methods to be offloaded at method level transparently without requiring any modifications to the applications source code, executables, or demanding special recompilation.
- We implement FaST in the Android Dalvik virtual machine and evaluate it with a few popular mobile applications from Google Play to show that offloading mechanism works seamlessly with existing applications.

The rest of this Chapter is organized as follows. Section 3.2 presents the design, while section 3.3 details our implementation. We evaluate FaST in section 3.4 and present the parallelization model on section 3.5. We then make concluding remark in section 3.6. We also discuss the limitations of the offloading mechanism in section 3.7 which is addressed in Chapter 4.

3.2 Design

In this section we present the design of the offloading mechanism model FaST. Figure 3.1 shows the various components of our computation offloading model FaST. While details are presented in the subsequent sections, at high level we intercept the methods in mInterceptor and offload them to the server by sInterface. The FaST Interface in the server-side executes the offloaded method there and gives the result back from the server to the mobile device. After the mobile device gets the result back, it follow its original way of execution.

As we have discussed, it is desirable that computation intensive methods of any existing application can be offloaded transparently. For this objective, FaST offloading mechanism consists of several components as shown in Figure 3.2. To describe the components' responsibilities and the interaction between them, we use the DroidSlator [57] application (Android dictionary application) as an example. DroidSlator has a member class ThaiDict which has a method translate(String inWord, String toLang). The input parameters of translate(String inWord, String toLang) are two Strings and the return value is of type String. The DroidSlator class is to create an object **dict** of ThaiDict class in

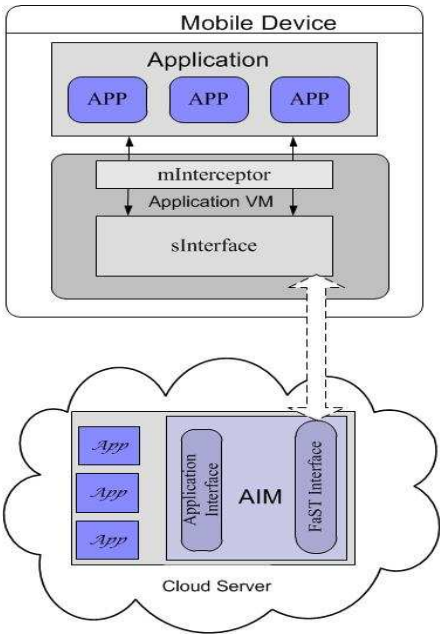


Figure 3.1: Different Components of Computation Offloading Mechanism: `FaST`

the `translateIt` method, and invoke **dict**'s `translate` with two Strings. The return value from `translate` is placed in the object `String`. In the following description, we assume the offloading decision has been made and we are intercepting the `translate` and offloading it to the server.

There are two major components for offloading in `FaST`:

- **mInterceptor:** The `mInterceptor` intercepts the method invocation at the Dalvik VM instruction level and redirects the execution to the server side. After getting the results back from the server, the application proceeds following its original logic. In case of a local execution or a remote server failure, `mInterceptor` lets the application continue its normal execution flow. While intercepting, `mInterceptor` gets the current executing methods' class and parameter list, return type, etc. If a decision is made to offload the method, it serializes the input parameters and the reachable objects from this methods and sends them to the `sInterface` to offload the method along with the parameters. In this example, `mInterceptor` traps the `dict.translate(inWord,`

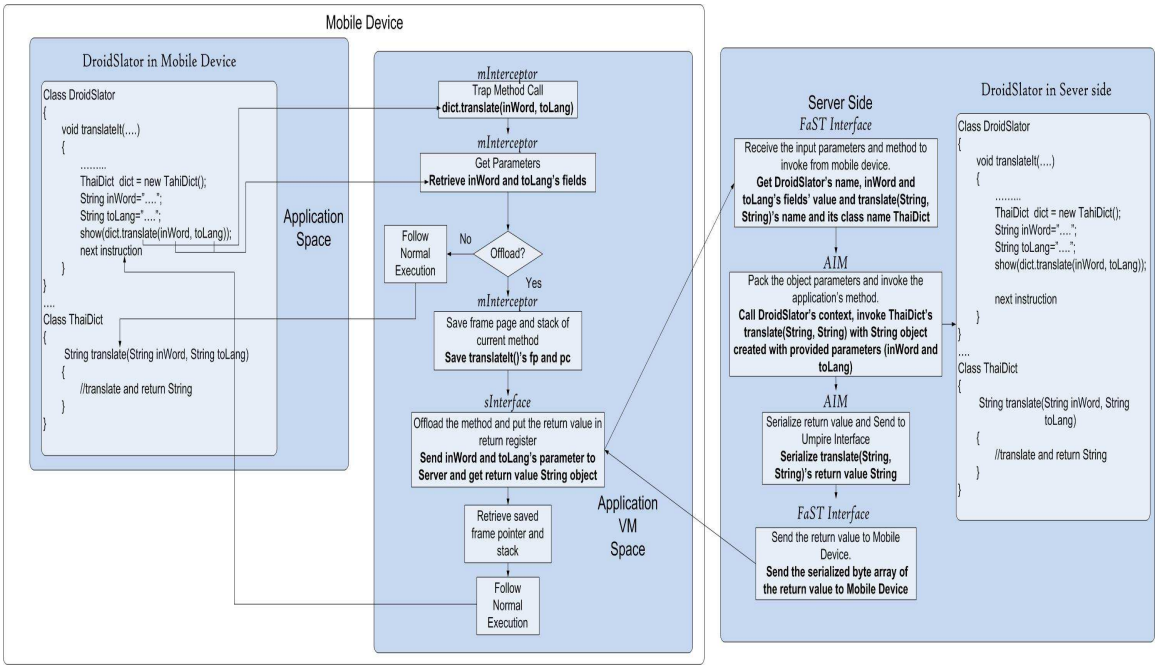


Figure 3.2: Offloading Flow Chart

`toLang`) in `translateIt` method of `DroidSlator` class, and retrieves the input parameters (inWord, toLang) and sends them to the server for execution. After the server side has completed the execution, `mInterceptor` gets the result back from `sInterface`, de-serializes the result to build the appropriate object (String object in this example) and returns to the invoker of the method.

- **sInterface:** Once an offloading decision is made, the intercepted method is offloaded to the server side. For this purpose, a client-side interface `sInterface` is developed on the smartphone so that the entire process can be automated and transparent to `mInterceptor`. The `sInterface` is also responsible to get the result back from the server.

On the server side, `FaST` requires the same application to be installed (not running) so that the server can invoke and execute the same method that is offloaded. The mobile applications can be installed and executed on the application VM or on the desktop machines

if supported. These applications are managed by the `Application Interface Manager (AIM)`. The `AIM` receives the offloading requests along with the parameters through the `FaST Interface`. So in the DroidSlator example, `AIM` instantiates the appropriate method class `ThaiDict` at runtime, creates the input parameters `String inWord` and `String toLang` with the provided information from the mobile device, and executes the method `translate`. Once the execution is completed, the server returns the result (a `String` instance) back to the mobile device through the `FaST Interface`.

3.3 Implementation

We have implemented `FaST` on the Dalvik VM in the CyanogenMod [58] distribution. The major implementation challenge comes from the offloading mechanism we have designed. In this section, we present it first and then describe the server side implementation.

3.3.1 Client-side Implementation

Dalvik VM [59] is a virtual machine that allows applications to run on Android devices. Dalvik supports Java code, but it converts the JVM bytecode to Dalvik bytecode and puts all the classes of an application to a single `.dex` file. While executing, Dalvik loads the definition of the classes from the .dex file and stores relevant information in the classobject structure. The Java functions in each class is mapped to the method structure. The method's definition includes the name and the first instruction address in memory mapped dex file.

In Dalvik, each kernel thread is dispatched to one user level thread. Each thread proceeds instruction by instruction. Similar to any other instruction, a method invocation has its own instruction (dalvik opcode). There are several types of methods (direct, virtual, static, super class, etc.) and each of them has different opcode for invoking. When Dalvik interpretor reaches any method invoking instruction, it jumps to the address of the method's instruction and starts to execute it. Before jumping to the method's instruction, Dalvik saves the frame page and the program counter of the current method to restart from the

same point after returning from the invoked method. Then it makes the invoked method as the current method. The input parameters of the invoked method is saved in the invoked method's frame page. The frame page saves the values for primitive and the address of the object input parameters. This address of an object (string, array, or complex object) points to a structure which holds the memory address and content of the object's member fields. Once the input parameters are set, Dalvik jumps to the invoked method's instruction to start execution. At this point, `mInterceptor` intercepts the method call and offload the method.

While offloading, `FaST` gets all the fields values of the input parameters and packs them in a byte array. For each parameter, in the byte array, `FaST` puts the name, the type, the length of the content, and the actual content of the fields. If any object refers to a superclass or has another class object as a member, `FaST` recursively copies the contents of their fields as well. The name of the application, method, and class; the parameters' type are also encoded in the byte array.

Once packed, `mInterceptor` sends the byte stream to the `sInterface` and suspends for the return value. The `sInterface` communicates with the server side to execute with the provided parameters and gets the result back. It follows the same protocol to construct the return object out of the byte stream sent from the server. After de-serializing the return object, `mInterceptor` returns it to the invoking method. To return from the invoked method, `mInterceptor` puts the value (for primitive data types) or the address (for other objects) in the `return value` field of the Dalvik interpreter. Similar to the `program counter` of `frame page`, the `return value` is a field for the current executing thread where the returned entity from an invoked method is saved. `mInterceptor` then re-starts normal execution of the application. It first retrieves the invoking method's frame page and the program counter (which were saved before), and sets the invoking method as the current method and starts the execution from where it was suspended.

In the entire process, `sInterface` is responsible for communicating with the server to offload computation. `sInterface` uses socket programming to communicate with the server

to send and receive raw data.

3.3.2 Server-side Implementation

To facilitate the offloading, `FaST` also has a simple server side interface. The `FaST Interface` is a component on the sever side, which communicates with the `sInterface` on the mobile device to exchange data for computation offloading. We have implemented this component with socket programming.

`Application Interface Manager (AIM)` gets the raw byte stream from the `FaST Interface` and unpacks according to the protocol discussed before. Based on the received information, `AIM` can find the appropriate application to bring in context and uses Java reflection to invoke the particular class's method dynamically. It also reconstructs the input parameters of the invoked method. After the invoked method returns the result, `AIM` packs the return object's values following the same protocol and sends the resultant byte stream to the mobile devices.

`FaST` server side implementation runs on an Android ASUS R2 virtual machine in VirtualBox [60].

3.4 Evaluation

We evaluate `FaST` with eight popular Android applications. Table 3.1 gives a brief description of these applications as well as the candidate methods for offloading. Among these applications: DroidSlator, LinPack, Picaso, and NQueen are computation-intensive. The data transfer time is negligible compared to the computing time. In contrast, ZXing is data-intensive. Therefore, the offloading overhead mainly comes from the data transfer time. Mezzofanti and MatCal are both data- and computation-intensive. The size of each image in Mezzofanti is around 3 MB. It also takes significant time to filter the text from the image. Matrix operations are time-consuming in MatCal, and the data size could be large for large matrices. Compared to other applications, MathDroid is not computation- or data-intensive.

Table 3.1: Description of Evaluated Applications in FaST

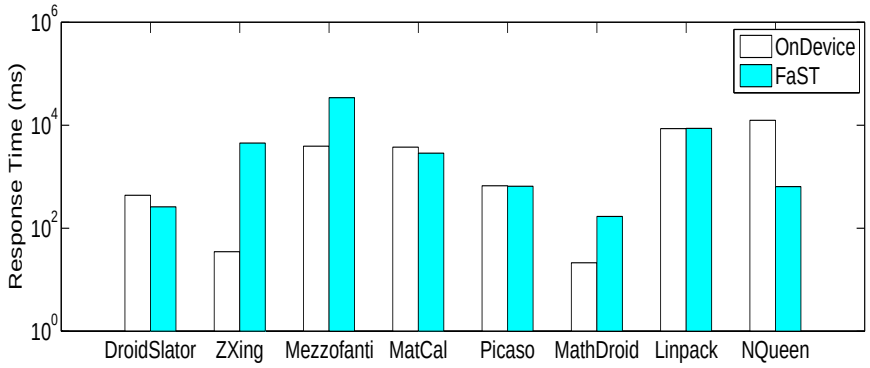
Application	Offloading Candidate	Description
DroidSlator [57]	`translate(String inWord, String toLang)` method of `ThaiDict` class	Android translation application
Zebra Xing(ZXing) [44]	`decodeWithState (BinaryBitmap)` method of `MultiFormatReader` class	Android product bar code reader
Mezzofanti [45]	`ImgOCRAndFilter(int[] bw_img, int width, int height, boolean bHorizontalDisp, boolean bLineMode)` method of `OCR` class	Android optical character recognition application
MatCal [61]	`times(Matrix B)` method of `Matrix` class	Android application for matrix operation
Picaso [62]	`project_and_compare(Bitmap bm)` method of `ReadMatlabData` class	Android face recognition application
MathDroid [63]	`computeAnswer(String query)` method of `MathDroid` class	Android Calculator application
LinPack [43]	→ `linpack()` method of `LinpackJavaActivity` class	Android CPU and Memory Benchmark
NQueen	→ `findSolution(int board[],int n, int pos)` method of `NQueen` class	Android application for NQueen problem

We evaluate FaST in a dynamically changing environment where the network bandwidth, latency, input data size, server CPU and memory change. On the server side, we run Android ASUS R2 virtual machine in Virtual Box [60]. We change the server's CPU availability and memory by utilizing CPU and memory cap of the Virtual Box. We have five different configurations to emulate five different environments in practice, where the network bandwidth, CPU availability, memory size, and network latency are set as the following Table 3.2. For experiments, we collected one-month traces of three users from these applications using a Google Nexus One phone with 1 GHz CPU and 512 MB RAM and we use PowerTutor [64] to measure the power consumption of the applications.

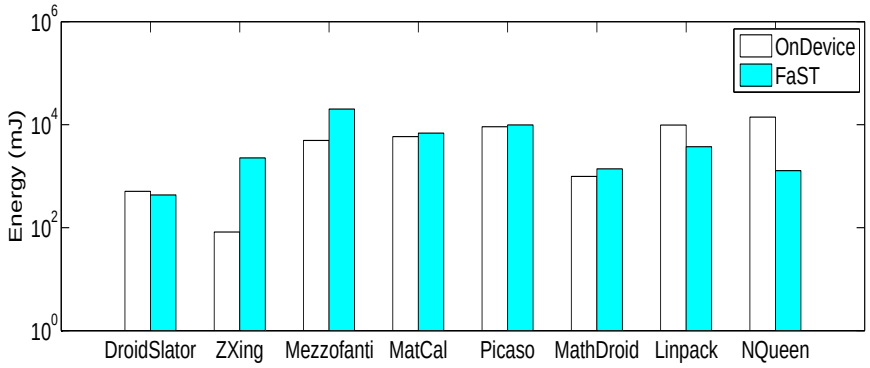
Figure 3.3 shows that compared to always local executions (OnDevice), FaST provides the user faster response time for DroidSlator, MatCal, Picaso, and NQueen (59.62%, 76.26%, and 97.91%, and 5.16% of local execution time, respectively). The energy consumption is about 84.98% and 9.14% of the energy consumption when they are executed locally for DroidSlator and NQueen. However, by offloading, FaST is consuming significantly higher response time and energy consumption for ZXing, Mezzofanti, and MathDroid compared

Table 3.2: Experimental Environment Setup

Simulated Network	Network Bandwidth (Kbps)	CPU	Memory (MB)	Network Latency (ms)
LAN	100000	2 GHz	1024	20
WLAN 20 ms	30000	1 GHz	2048	20
LAN 802.11g	25000	2 GHz	2048	50
4G	5000	2 GHz	2048	75
3G	500	2 GHz	2048	200



(a) Response Time



(b) Energy Consumption

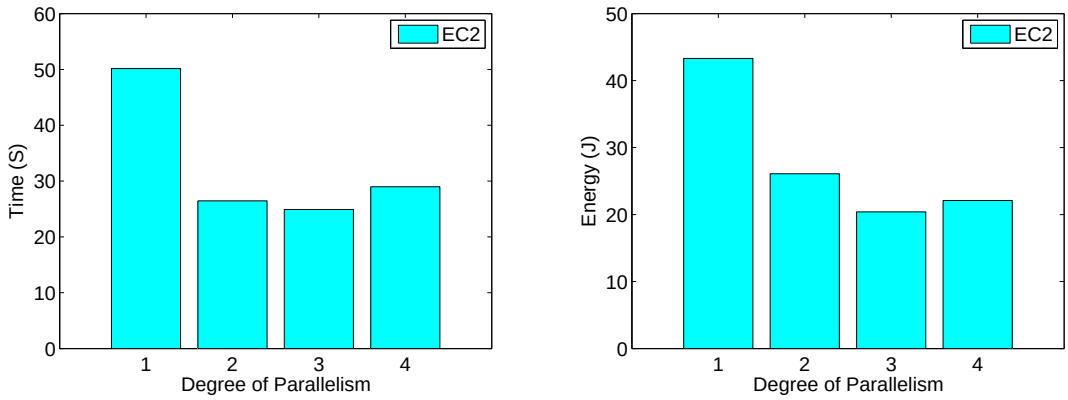
Figure 3.3: Comparison of Response time and Energy Consumption

to OnDevice execution. So, always offloading the resource intensive method may not favor the response time and energy consumption of mobile applications. Nevertheless, from our

experiments, we can see that FaST can work seamlessly with real world applications.

3.5 Parallelization Model

Parallel processing for computation has been well studied for big data computation and large clusters [65, 66]. But these solutions mainly focused on high-end scientific and big data computation for wired computers with stable network. Both the nature of the computation and the network connection are different for mobile devices. The network does not provide any guarantee and the performance of the computation processed in parallel is significantly dominated by the bandwidth and latency of the network. Therefore, we design and implement a model to farther increase the performance by automatic parallelism for offloaded methods if possible.



(a) Response Time (b) Energy Consumption

Figure 3.4: Response Time and Energy consumption for Parallel Processing

To study the impact of parallelism on offloading in the server-side, we have conducted some preliminary experiments for text search application with Amazon EC2. We have offloaded the method from a Google Nexus One with Android 2.2 OS, 1 GHz CPU, and 512 MB RAM to Amazon EC2 instance with 5 GHz CPU, 1.7 GB RAM, and 300Kbps of network

speed. We have further parallelized the offloading into 1, 2, 3, and 4 instances. Figure 3.4(a) shows the response time, while Figure 3.4(b) shows the energy consumption. Motivated by our findings, we have designed farther parallelism whenever possible to enhance the performance of the offloaded method of mobile devices.

To expedite the performance of the offloaded method, `FaST` farther parallelizes the offloaded method. Figure 3.5 shows the high level design of the model. While parallelizing, the mobile device is responsible to divide the offloaded method among multiple servers and merge the results after getting the results back. In addition to that, the mobile device also finds the appropriate server-nodes and handles the node failures as well.

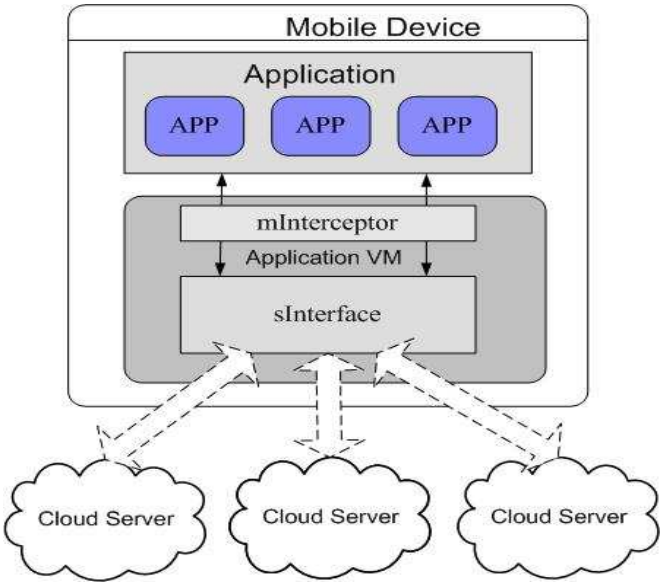


Figure 3.5: Components of Parallel Computation Offloading model

3.5.1 Design

Figure 3.6 illustrates the work flow for the parallel processing. After exploring servers where offloaded methods can be executed, the `FaST` Scheduler establishes connections between the

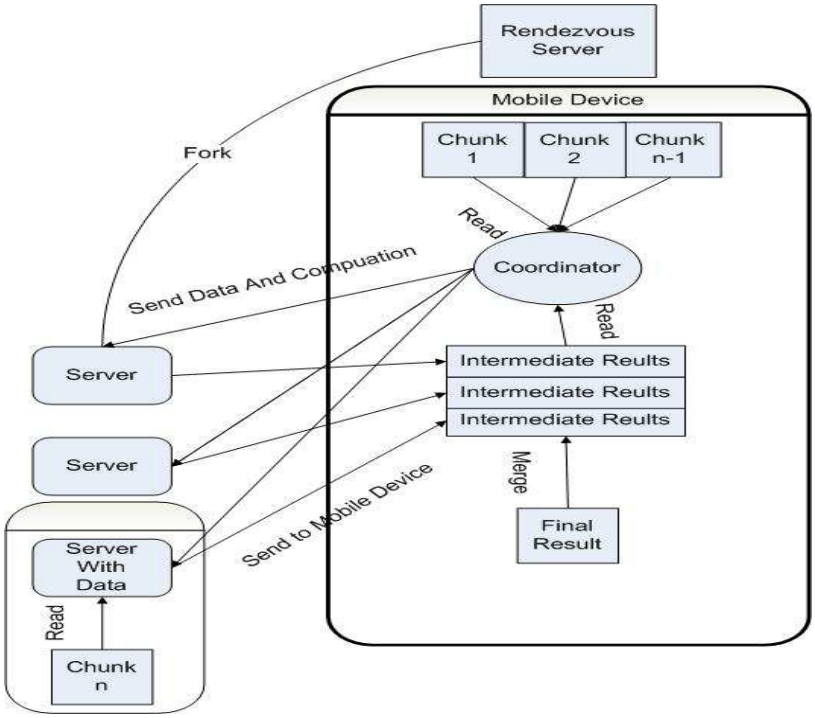


Figure 3.6: Parallelization Model

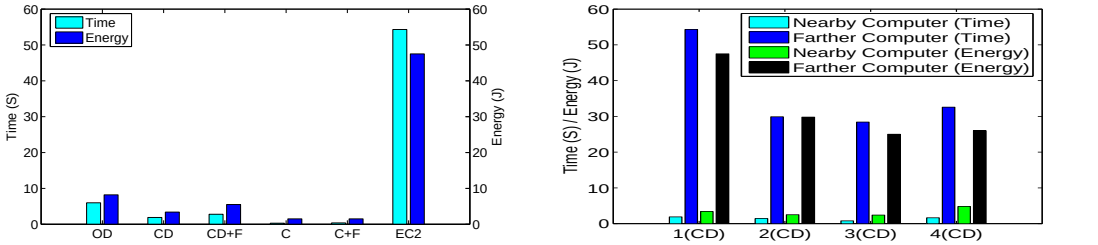
mobile device and the servers. The mobile device divides the whole input data into `n` small chunks and read input chunks. It then sends the data and the computation to the appropriate servers. Note here that some data servers may have some data in prior. These servers are referred by the `FaST` scheduler so that only the computing program needs to be sent there, thus reducing the data transferring overhead. Some data chunks may be executed in the mobile device itself due to security reason. For all the other servers, both computing program and data need to be sent over. The mobile device also keeps track of the the processing progress of the job. Currently we send the computation in serializble byte code from the master node to the data servers. It also defines the return value type as well. The servers receive the data portion in desired input data format and start the computation and sends back the output to the mobile device. The mobile device saves the intermediate results and marks the entry for that particular portion of the input data chunk as finished. The mobile device sends the next portion of the job to the servers and the server start

computation for the next portion of the job. This continues until the job is finished or any server fails (e.g., a server is shut down or loses the connection from the mobile device). If any server fails, mobile device tries to get the result for that data portion from the resource overlay or re-execute that portion of the job by exploring another worker server. After finishing the remote execution, the mobile device begin merging the intermediate results.

3.5.2 Evaluation

In this section, we describe the performance of the parallelized offloaded method with three applications. We repeat each experiment five times and present the average of the results.

- **Text Search**



(a) Response Time and Energy Consumption

(b) Response Time and Energy Consumption for Parallel Processing

Figure 3.7: Text Search

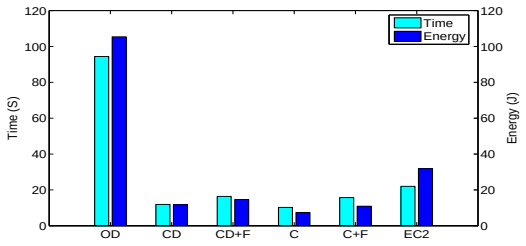
Figure 3.7 depicts the response time and the energy consumption of the text search application when it is executed on the Android and the computation is offloaded with and without parallel processing. In this experiment, the amount of data transferred is 2.6 MB when both computation and data are transferred, otherwise it is 1 KB if only the computing program is needed to be offloaded. In offloading the data file, it is divided into 512 KB chunks. So in the cases with node failure (`CD+F` and `C+F`), only one chunk is lost, which minimize the re-execution overhead.

In Figure 3.7, the left-y axis represents the response time while the right-y axis represents the corresponding energy consumption. Figure 3.7(a) clearly shows that offloading to EC2 results in the worst performance in terms of both the response time to the user and the amount of energy consumed, even worse than the on-device execution. Compared to the case when it is executed locally on the mobile device, offloading to the nearby residential computers results in 69% and 59% less response time and energy consumption, respectively, although offloading to nearby residential computers demand some bandwidth for file transferring. In the node failure cases where residential computers may not be reliable or the home user can depart a residential computer from `FaST` at any time, Figure 3.7(a) shows that the performance of offloading still outperforms the on-device computing in terms of both the response time and the total energy consumed on the mobile device, although there is a 47% and 61% increase compared to if there is no node failure.

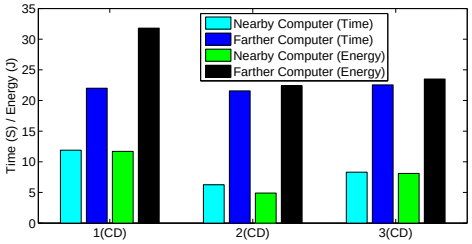
When the computation is parallelized among multiple machines, Figure 3.7(b) shows the result. Again, the left-y axis represents the response time while the right one represents the energy consumption. The residential computers are identical with a 2 GHz CPU and 2 GB RAM. The rented EC2 instances have 5 GHz CPU and 1.7 GB RAM each. Without parallel processing, the response time may be well over the average connection time of a mobile user with a roadside WiFi ranges between 6-12 seconds [67]. This makes it impossible for a mobile user to get the result in time in the same communication session although EC2 has a faster CPU speed. This would be a critical problem for delay sensitive mobile applications when a user waits to get the result back. As shown in the figure, parallelization can clearly improve the performance when the number of computers is increased from 1 to 2 and 3. However, Figure 3.7(b) also shows that the response time and energy consumption first decrease with the increase of parallelization level, then it increases when the parallelization level increases (from 3 to 4 nodes). So here it is also important to calculate the the appropriate degree of parallelism to optimize the performance.

Again, in Figure 3.7(b), we also observe that the residential computers perform significantly better than EC2 when both the data and computation (`CD`) are offloaded. But when only the computation (`C`) is offloaded, they have similar performance.

- **Face Detection**



(a) Response Time and Energy Consumption



(b) Response Time and Energy Consumption for Parallel Processing

Figure 3.8: Face Detection

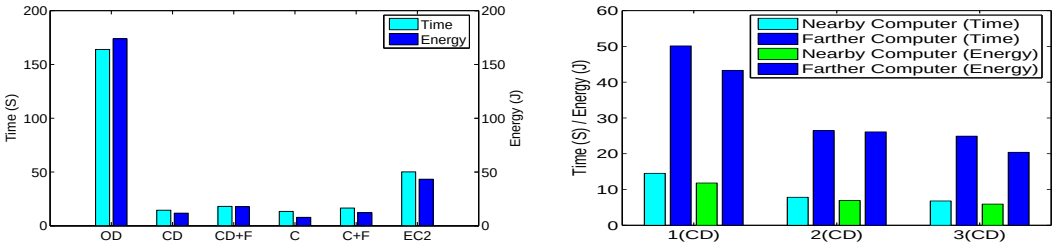
Figure 3.8 shows the performance results when the face detection program is executed in different environments. In particular, figure 3.8(a) shows that executing on the Android takes the longest time of about 94.5 seconds. Not surprisingly, the corresponding energy consumption is the largest for the on-device execution.

Figure 3.8(a) also shows that both the response time and the energy consumption are reduced when the computation is offloaded. When the program is offloaded to the nearby residential computers, the performance improvement is more pronounced than when the program is offloaded to EC2: on the residential computer, the response time is about 10.25 seconds and 11.90 seconds without or with the data transferred. Correspondingly, when the computation is offloaded to the nearby residential computer, the energy consumed is only about 23% and 36%, respectively, of the total energy consumption when we have on device execution without or with data transfer.

With the help of parallelization, the performance is better. Figure 3.8(b) shows the

effect of parallelism on the response time and the energy consumption. However, as shown in the figure, although using 2 nodes to parallelize the computation does improve the user response time and the total energy consumption on the mobile device, the response time and energy consumption of the computation actually increase when the parallelization is further increased (from 2 nodes to 3 nodes). When the computing nodes have the data in prior, the performance is better than when the data need to be actually transferred before the computation. This indicates offloading computation to the nodes where data resides may be more beneficial than to the nodes with higher computation power without any data in prior. But again, an appropriate parallelization level is always desired as more resources may not improve the performance.

- **Image Sub Pattern Search**



(a) Response Time and Energy Consumption

(b) Response Time and Energy Consumption for Parallel Processing

Figure 3.9: Image Sub Pattern Search

For *Image Sub Pattern Search*, Figure 3.9(a) shows that executing on the Android takes the longest time of about 163.9 seconds. In all the offloading scenarios, the response time is significantly reduced. Correspondingly, the energy consumption is the largest for the on-device execution. The reduction when the program is offloaded to the nearby residential computers is more pronounced than when the program is offloaded to EC2: on the residential computer, the response time is about 13.37

seconds and 14.52 seconds without or with the data transferred. Correspondingly, the energy consumed is only about 10% and 11% of the On-device (OD) computation, respectively, when the computation is offloaded to nearby residential computers. The node failure only causes 54% and 51% increase of the response time and the energy consumption compared to their counterpart without any node failure. These results are still much better than those when offloading to EC2.

Figure 3.9(b) shows the response time and the energy consumption when the computation is parallelized. When two and three nodes are used for computing, both the response time and the energy consumption on the mobile device decrease. However, when more nodes are used in the computing, performance degrades due to parallelization overhead. On the other hand without parallelization, the response time is more than 10 seconds, but with parallelization, the user gets the result back in 5 seconds, which is more feasible.

3.6 Summary

The pervasive mobile devices are driving the fast development of mobile applications. For resource-intensive mobile applications, there is an increasing demand to offload to more powerful counterpart, such as clouds and servers. Existing offloading mechanisms often require changes to the application binary or source code or recompilation. To address this problem, we have designed and implemented FaST, a framework for smart and transparent mobile application offloading. In FaST, we have implemented a transparent offloading mechanism at the method level. Compared to existing schemes, FaST does not require any modifications to the source code or binary, or special compilation. Thus, the large number of existing applications can be directly offloaded without further efforts. Extensive experiments have been conducted to evaluate FaST. The results show that FaST can work seamlessly with existing mobile applications.

3.7 Discussion

Our experimental results in section 3.4 shows that `FaST` can not improve the offloaded methods performance in every case. Sometimes the performance of the offloaded method is even deteriorated. The offloaded method's performance depends on many factors including application type, network condition, server's availability, etc. Having built a transparent offloading mechanism, in the next Chapter, we design and implement a profiler to estimate methods' OnDevice and OnServer execution time for proper offloading decision.

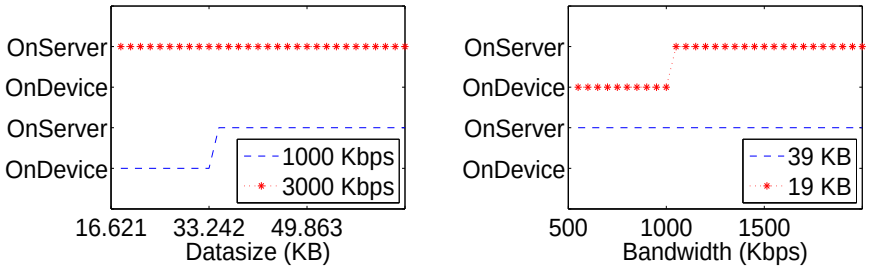
Chapter 4: Lamp: Learning Model for Application Profiling

4.1 Introduction

In the previous Chapter, we discussed the transparent offloading mechanism for mobile applications. Regardless which approach is taken, a mobile user often desires the fastest response time and the minimum battery consumption. While prior studies show that better response time and less energy consumption can be achieved with computation offloading, offloading is appropriate only when the offloading overhead and the remote execution cost is less than the local execution cost. Thus, always executing applications on the server side, which greatly simplifies the system design and implementation, may sacrifice a user's best interests in terms of the response time and the energy consumption.

At a high level, making the appropriate decision for a method offloading depends on the performance difference between the local on-device and remote on-server executions as well as offloading overhead. In practice, many factors can impact the above aspects [11, 12, 68], and such impact may be dynamic. In making the offloading decision, MAUI [12] adopted a linear model to make the offloading decision, while Odessa [13] leveraged previous execution statistics and used the index of the ratio-of-benefit to make the offloading decision. Ionan et al. [26] proposed history based profiling of the applications running on the server and the smartphone, and leveraged the ratio of them to make the offloading decision. Comet [42] also proposes a threshold based policy for offloading computation: when the thread execution time exceeds twice of the RTT to the server, then it migrates the thread.

Some prior studies (e.g., [14]) suggested a threshold based policy for offloading decision-making. A method is offloaded only when its parameter data size is greater than a pre-defined threshold value. Apparently, it is difficult to set a generic threshold as it is often application-dependent. More importantly, this might work in a static environment where the



(a) Data size (KB) (b) Bandwidth (Kbps)

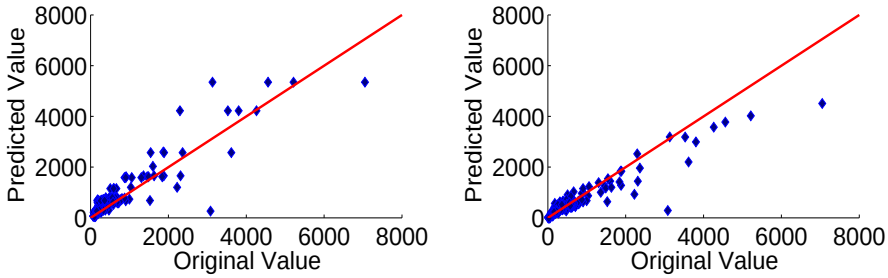
Figure 4.1: A Threshold Policy Fails

resource information is stable, such as stable bandwidth between the sever and the mobile device. For a mobile connection where conditions change continuously and dynamically, it is hard to make the offloading decision solely based on a static threshold. Moreover, many resource-intensive applications, such as Mezzofanti [45] and ZXing [44], always have relatively stable input sizes. For these applications, a threshold based policy always makes the same offloading decision without considering network bandwidth or latency, let alone other factors such as memory or CPU availability.

To demonstrate the limitation of a threshold based policy, we evaluate an Android face recognition application Picaso [62]. We use an Android Google Nexus one (with 1 GHz CPU and 512 M memory)and a virtual machine (with 2 GHz CPU and 2 GB memory) on the server to offload computation. Figure 4.1 shows the results. We conduct experiments for 30 runs. In Figure 4.1, the y-axis represents the optimal decision (whether offloading or local execution is the better). In Figure 4.1(a), the x-axis represents the data size in KB, while in Figure 4.1(b), it represents the bandwidth in Kbps. As shown in Figure 4.1(a), when the bandwidth is 3000 Kbps, it is optimal to offload the computation for every data size (so the threshold value should be zero KB here). But when the bandwidth drops to 1000 Kbps, it is better to offload only when data size is greater than 39 KB. Thus the threshold value should be changed to 39 KB in this case.

Figure 4.1(b) shows the optimal decision for the data size of 19.51 KB and 39.03 KB

on average, where 19.51 KB is the average size of one picture and 39.03 KB is the average size of total two pictures. When the data size is 39.03 KB, it is optimal to offload the computation if the bandwidth is greater than 500 Kbps. On the other hand, if the data size is 19.51 KB, then the threshold value changes to 1000 Kbps. These experiments only consider the interplay of the data size to transfer and the network bandwidth. Apparently, a threshold based policy could not handle such complexity.



(a) Linear Regression (b) Support Vector Machine

Figure 4.2: Linear Regression has Low Accuracy

MAUI [12] suggests linear regression for predicting the on-device and on-server execution time with the given system features and makes the offloading decision accordingly. A linear regression model works well when only a limited number of features are changing, such as the bandwidth or latency as considered in MAUI [12]. But the complexities among the interplay of more features cannot be captured by a linear regression model. Furthermore, these key features do not necessarily have linear relationships. To verify our intuition, we conduct the translation of random-sized paragraphs using DroidSlator [57]. Figure 4.2(a) details the result of linear regression and Figure 4.2(b) illustrates the result of SVM. In these two figures, the x-axis represents the real response time and the y-axis represents the predicted response time in ms with a same setup as in the evaluation (subsection 4.5.2). Ideally the points here should follow a straight line with a slope of 45^o. From these two figures, we can see that SVM is comparatively closer to the desired straight line while the

linear regression model produces many more erroneous predictions: linear regression has a root relative absolute error rate of 58.01%, while it is 17.66% for SVM.

In practice, a number of factors other than the data size (to transfer) and the network bandwidth could change dynamically, and their values and interplay could lead to different on-device and on-server execution times. We thus aim to identify such key features and look for the most appropriate classification model for such purpose. Ideally, we expect a model that is i) lightweight to run on mobile devices, ii) of higher dimensional, iii) able to capture the interplay between key features, iv) tolerant of noise, v) highly variant with low bias, and most importantly vi) highly accurate and vii) self-learning gradually over time.

While details are presented in the subsequent sections, our contributions in this Chapter include:

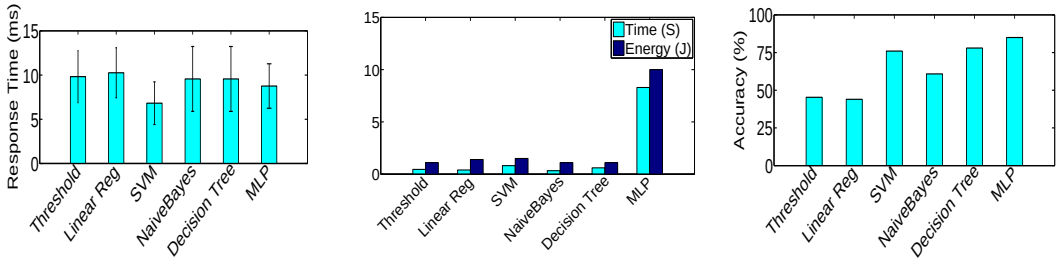
- `Lamp` adopts a dynamic learning based approach that can adaptively and intelligently estimate the on-device and on-server execution time for making more accurate offloading decision in real-time with high accuracy, while consuming acceptable energy overhead when running on mobile devices.

- We have implemented `Lamp` in the Android Dalvik virtual machine and evaluated it with a few popular mobile application from Google Play to demonstrate its effectiveness.

The rest of this Chapter is organized as follows. Section 4.2 presents the classifier comparison, while section 4.3 details the significance of different features on the offloaded method's performance. We present design and implementation in section 4.4 and evaluate in section 4.5. We then make concluding remark in section 4.6. We also discuss the limitations of `Lamp` in section 4.7 which is addressed in Chapter 5.

4.2 Classifier Comparison

We have empirically shown that the threshold and linear regression models are oversimplified and cannot capture the relationship between the features in section 4.1 and we also found



(a) Classification Time (b) Time and Energy Consumption (c) Accuracy

Figure 4.3: Comparison between Different Classifiers

SVM performing better. Such results hint us to look for a better classifier to capture the relationship between various features impacting the offloading decision. For this purpose, we have considered several commonly used classifiers.

Figure 4.3 shows the result of different classifiers we have examined with our data set of Picaso [62] collected from three different users. We implement the classifiers using open source library `Weka` [69] on the Android phone Google Nexus one with 512 MB memory and 1 GHz CPU. We have trained the classifiers with 50% of the data (for Picaso) because more training could help some classifiers. Figure 4.3(a) shows that the time to classify one instance remains almost the same for all the classifiers, but multi-layer perceptron (MLP) takes more time and energy for training (Figure 4.3(b)). Obviously higher dimensional classifiers like MLP, SVM, and Decision Tree (DT) achieve better accuracy (around 80%) than the other lower dimensional classifiers (Figure 4.3(c)).

Among these models, probabilistic classifiers such as `Naive Bayes` is lightweight and easy to train. But such classifiers are biased towards previous observations. Moreover, Naive Bayes assumes the features to be independent of each other which is not true in practice. We have also considered other probabilistic classifier like `Expectation Maximization` and `Markov Chain`. `Expectation Maximization` is not suitable as it does not support on-line training and assumes the feature values to follow a particular distribution, while `Markov Chain` has too many states to handle.

Statistical classifiers such as `Decision Tree` are also lightweight with moderate accuracy. But it cannot extrapolate its decision, suffers from over-fitting, and lacks on-line learning property. MLP and SVM perform well in terms of accuracy and classification time. In fact, MLP takes more time for training. SVM has several advantages over MLP such as SVM is easy to train and free from over-fitting. Both of them have expressive power and can capture the relationship between multiple features. But the drawback of SVM is that it does not support on-line learning. All the instances have to be trained at once to draw the boundaries between the higher dimension features. On the other hand, though MLP takes a little bit more time for training, it supports on-line learning and works well in noisy environments. It also supports higher dimension space and can capture the relationship between them. The classification and training time for one instance is really low. We argue that the first time training is one-time cost for each device so the amortized cost will be reduced over time. Thus we decide to use MLP for `Lamp` as the learner for our prediction model.

4.3 Input Features and Impact

The input features are critical for classifier training and prediction. To figure out key features, we have collected three users' trace of different Android applications for one month. During the data collection, we have changed the memory and CPU availability on the server randomly and used different network configurations. In this section, we present our empirical study to show the impact of different features using the face detection application Picaso [62] as an illustrative example. We have used an Android Google nexus one smartphone with 1 GHz CPU and 512 MB RAM, while on the server side, we have an Android ASUS R2 running on the virtual machine in the Virtual Box [60] with varying CPU and memory availability.

- **Network Bandwidth**

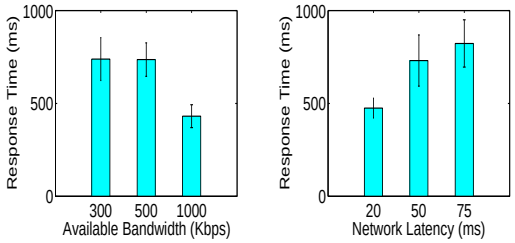
  To observe the impact of network bandwidth between the device and the server on the

response time, we plot the response time of Picaso by offloading the face recognition task to the server side from the smartphone. We take the average response time of 10 experiments (a different image for each experiment) by only changing the network bandwidth while keeping all other parameters unchanged. We used `tc` [70] to shape the network bandwidth and latency. In our setup, the latency between the mobile device and the server is set to 20 ms. We keep the CPU availability in server to be 74% (representing 2 GHz of a 2.7 GHz CPU). The memory in the server is set to 2 GB.

Figure 4.4(a) presents the average of 10 experiments with 95% confidence interval for bandwidth of 300 Kbps, 500 Kbps, and 1000 Kbps. We can observe similar impact of network bandwidth as in [12]. Figure 4.4(a) shows that the response time changes (getting better) with the increase of network bandwidth, especially when the network bandwidth is changed from 3G (500 Kbps) to 4G (around 1 Mbps), indicating the non-linear relationship between the network bandwidth increase and the response time.

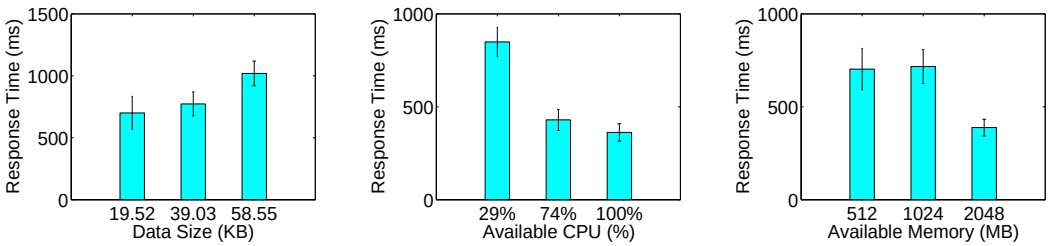
- **Network Latency**

  Previous studies [12, 13] have presented empirical data to show that higher network latency worsens the response time of the offloaded computation. To verify this finding empirically, we plot the average response time for Picaso for 10 offloaded experiments (a different image for each experiment) by only changing the network latency while keeping the other features constant. In this setup, the bandwidth is set to 1000 Kbps, with 37% CPU availability (represents 1 GHz of a 2.7 GHz CPU). The memory of the server is set to 2 GB. The network latency is changed between 20 ms to 75 ms. Figure 4.4(b) shows the average response time with 95% confidence interval when the network latency changes. The result does show that a low latency can improve the response time. It also shows the latency reduction is not proportional to the reduction of the response time.



(a) Impact of Bandwidth (Kbps) (b) Impact of Latency (ms)

Figure 4.4: Bandwidth and Latency



(a) Impact of Data Size (KB) (b) Impact of CPU availability (%) (c) Impact of Memory Size (MB)

Figure 4.5: DataSize, CPU, and Memory

- **Data Size**

  The offloaded computation often requires some data as the input parameter. The transfer time of such data is thus proportional to the size of the data if the network is stable. The bigger the data size, the longer the transfer and the response time.

  Figure 4.5(a) presents the response time of the offloaded method for Picaso with different data sizes. In the figure, the average response time is shown for 10 runs of data size of 19.51 KB, 39.03 KB, and 58.54 KB (which is the total. size of 1, 2, and 3 images) with 95% confidence interval. The other features are kept constant, the bandwidth is set to 1000 Kbps with 50 ms latency. The server has 74% CPU availability (representing 2 GHz of a 2.7 GHz CPU) with 2 GB memory. Again, the results indicate that the increase in the response time is not proportional to the

increase of the data size.

- **CPU Availability**

  For computing-intensive applications, the available CPU cycles play an important role in the response time. If the server-side VM has limited CPU cycles, then it might be worth executing the application locally.

  To observe the impact of CPU availability, we plot the average response time of 30 Picaso experiments while offloading. Each group consisting of 10 experiments (same image for each experiment) have the same available CPU cycles in the server while the other features remain constant for all 30 runs. In our setup, the bandwidth is set to 1000 Kbps with a latency of 20 ms. The server has 1 GB memory size.

  Figure 4.5(b) shows that the average response time with 95% confidence interval. In this figure, the response time gets halved when the CPU availability is increased from 19% to 74% (representing 512 MHz to 2 GHz of a 2.7 GHz CPU). More available CPU cycles (from 74% to 100%, representing 2.0 GHz to 2.7 GHZ) further improve the response time, but the improvement is not as significant.

- **Memory Size**

  Similar to computing-intensive applications, data-intensive applications highly depend on the available memory on the device. In our experiment with 1000 Kbps bandwidth, 20 ms latency, and 37% CPU availability (representing 1 GHz of a 2.7 GHz CPU) in the server-side VM, we test with 30 runs (same image for each group of 10 experiments) of Picaso for the memory size of 512, 1024, and 2048 MB in the server-side VM.

  Figure 4.5(c) shows that the average response time with 95% confidence interval. Clearly the response time gets better with more memory, which re-confirms that the available memory is an important factor to consider when making the offloading decision. However, we also observe that the memory increase does not proportionally reduce the response time.

So, we consider bandwidth, latency, data size, server's available CPU and memory as the multilayer perceptron model's input while predicting the OnDevice and OnServer execution time.

4.4 Design and Implementation

So far, we have discussed which parameters impact the execution time and their significance. Now we discuss how we measure the features that are demanded by the `Lamp`.

- **Network bandwidth and latency**:

  Due to the portable nature of mobile devices, the network bandwidth and latency between a mobile device and a remote server change dynamically. Thus, we need to frequently measure the network bandwidth and latency.

  Network performance plays a vital role only when the amount of data to be transferred is relatively large. Therefore, in our current implementation we take a trade-off. When the amount of data to be transferred is higher than a threshold (500 KB), we measure the network bandwidth and latency on the fly. Otherwise, we first examine how the network interface is connected, whether it is wifi, 3G, or 4G by `ConnectivityManager` interface in Android. Then we take the previous weighted bandwidth and latency for that configuration as the current network bandwidth and latency. Other location based bandwidth estimator like [71] can improve accuracy based on user behavior and location, but we have not implemented it in our current prototype.

  For measuring network bandwidth and latency on the fly, `Lamp` pings the server with two small sized packets P_1 and P_2 and waits for the response. Then it calculates the bandwidth BW and latency L. These values are weighted with the previously observed values to estimate the current value. `Lamp` keeps track of previous observations for different network configurations (currently we are bookkeeping for wifi (home/office), wifi (public), 3G, and 4G). For each network configuration, `Lamp` maintains a window W_{bw} for bandwidth tracking. Each window has n number of bins in the histograms.

Each bin i spans within the interval R_{i-1} to R_i. We set R_0 as the lowest available bandwidth (0 Kbps) and R_{n-1} as the typical highest speed for WLAN (30 Mbps) in the bandwidth window. The end interval is infinite for the window.

We estimate the current value with the following equation:

$$V = \sum_{j=1}^{n} \frac{C_j}{W} \times R_{j-1},$$

where C_j is the counter of the jth bin in the histogram and V is the estimated value. Whenever it measures a real value on the fly, it also updates the counters in the bins of the histograms. The new value replaces the oldest one in the window. To estimate latency, a similar window W_{lat} is used with 20 as the lowest latency (for WLAN) and 200 ms as the highest value (for 3G). Based on experimental results, we use a window size of 100 for both bandwidth and latency measurement, while the interval for bandwidth and latency is 1 Kbps and 10 ms, respectively, because our experimental results (omitted due to page limit) show the least error in prediction under these settings.

- **Data Size**

  We get the information about the input parameters' data size from the `mInterceptor`. When a method is intercepted, `mInterceptor` inspects its parameters' content from the method's frame page. If a parameter is of primitive data type, its size can be deduced directly. Similarly, for string and array object, the size can be obtained from their length and data type. For complex objects, we calculate the size by summing up the size for each of its member fields. If a member of an object is another object or has any super class, we calculate its size recursively.

- **CPU and Memory Availability**

  In our implementation, to measure the available CPU cycles, we read the idle cycles

that are left for the system to utilize. The underlying operating system provides us with the CPU statistics in /proc/stat file for Linux along with Android.

This statistics provides the total and idle CPU cycles. We use a similar technique as in [72] to predict the CPU availability just before making the offloading decision. We initially consider a moving average strategy for CPU estimation. But most of the Unix kernels have an `OnDemand` strategy as CPU governor, which changes the CPU frequency (and thus its availability) frequently. Because we obtained a high error rate with moving average, we predict the CPU availability right before making the offloading decision on the fly.

The available memory can change over time due to contemporary processes running in the system which can be found in */proc/meminfo* file in Android. We maintain another window W_{mem} to estimate the available memory with a window size of 500 and 1 MB interval based on our experimental results.

Note that in our implementation, to reduce the overhead, the CPU and memory availability information of the server is piggybacked when the server replies the ping requests or returns the results of the offloaded method.

Figure 4.6 shows the various components of our computation offloading model `Lamp`. In high level whenever an application starts to execute, we monitor the system environment to estimate the performance of the methods to be offloaded. If our learner model decides to offload the method, we make the offloading decision. By offloading these methods, we achieve optimum gain in response time savings. As we have discussed, it is desirable that computation intensive methods of any existing application can be offloaded transparently. For this purpose, we offload these methods to the server side and returns the result back to the applications running in the mobile device transparently by adopting `FaST` (Chapter 3). In case of not offloading, we execute the method locally. We monitor the offloaded (or the locally executed) methods response time for gradual learning of MLP.

There are four major components for the learning model in `Lamp`:

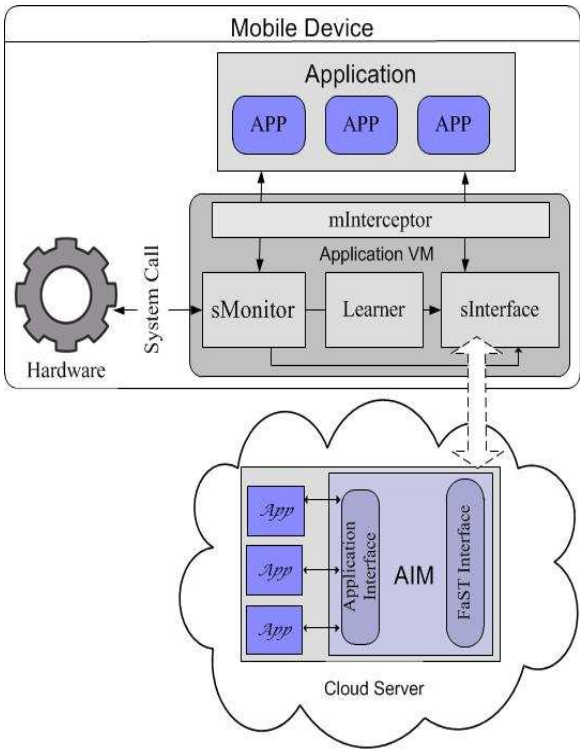


Figure 4.6: Components of Computation Learning and Offloading Model: `Lamp`

- **sMonitor:**

  The `sMonitor` is responsible for dynamically collecting the system environment and resource information, such as network bandwidth, latency, and CPU availability at real-time following a specified pattern. Such information is the input to the learner for making the offloading decision dynamically. In addition, `sMonitor` also keeps track of the execution time of the methods in the applications. This is to assist the identification of the resource-intensive methods as potential candidates for offloading.

- **Learner**

  As we discussed, `Lamp` uses MLP to characterize the relationships among different factors impacting the offloading performance. Whenever a potential candidate method is intercepted, the `learner` takes the method parameters, calculates the size of the arguments, and considers the feature values collected by `sMonitor`. Based on collected

information, it predicts the on-device and on-server execution time of the methods and make the offloading decision accordingly. The learner has a feedback channel as well so that it can learn by itself through the entire decision process. We have implemented the `learner`'s MLP in Weka [69].

- **mInterceptor** and **sInterface**

  We implement `mInterceptor` and `sInterface` according to `Lamp` as mentioned in section 3.3. We have also implemented the `server-side` accordingly.

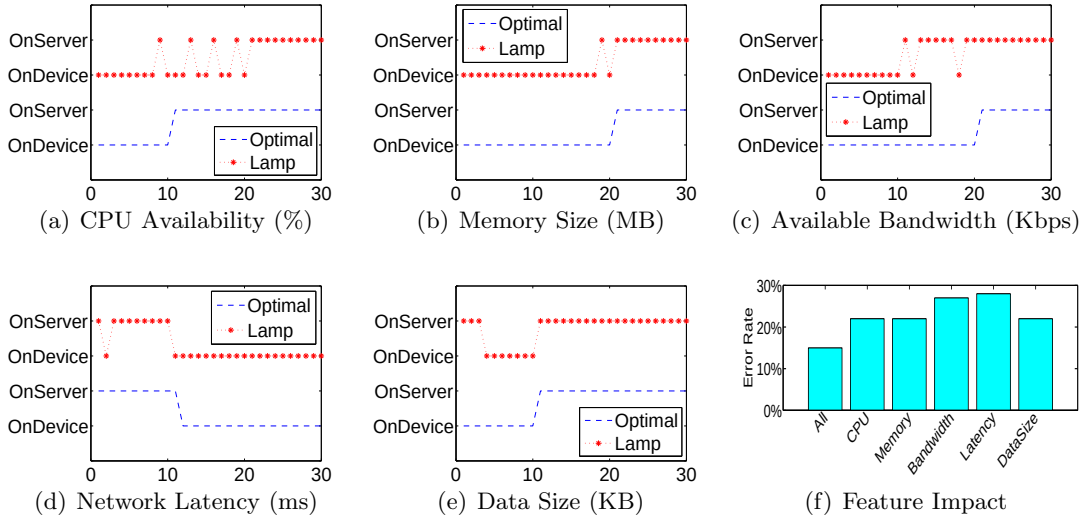
4.5 Evaluation

In this section, we study the performance of `Lamp` in a dynamically changing environment where the network (bandwidth and latency) and server capacity (available CPU and memory) changes.

4.5.1 Adaptability Evaluation

In order to study the adaptability of `Lamp`, we first run experiments when the network bandwidth, latency, and input data size change. On the server side, we run Android ASUS R2 virtual machine in Virtual Box [60]. We change the server's CPU availability and memory by utilizing CPU and memory cap of the Virtual Box. The virtual machine runs on a 2.7 GHz CPU on the server and the CPU availability is changed among 19%, 37%, 74% to 100% (e.g., 512 MHz, 1 GHz, 2 GHz to 2.7 GHz). The available memory is also changed from 512 MB to 2048 MB. The network bandwidth varies widely from 100 Kbps to 5 Mbps, while the network latency is changed from 20 ms to 75 ms by `tc` [70]. For experiments, we collected one-month traces of three users from these applications using a Google Nexus One phone with 1 GHz CPU and 512 MB RAM.

Among all applications, we use Picaso as a case study for adaptability evaluation. Other applications show similar results. We have 150 different Picaso images from the user trace for face recognition. That is, each experiment runs with a different image. We randomly



(a) CPU Availability (%) (b) Memory Size (MB) (c) Available Bandwidth (Kbps)

(d) Network Latency (ms) (e) Data Size (KB) (f) Feature Impact

Figure 4.7: `Lamp` Adaptability Evaluation

select 20% from the 150 optimal classifier output data set to train the MLP.

Figure 4.7(a) depicts the results when the CPU availability is changed in 30 experiments. In this set of experiments, the network bandwidth and latency are fixed at 1000 Kbps and 20 ms, respectively. The memory size of the server remains 1 GB. The CPU availability of the server is increased from 19% to 74%, and then to 100% (representing 512 MHz, 2 GHz, 2.7 GHz of a 2.7 GHz CPU). The first 10 experiments are run when the CPU availability is 19% (e.g., 512 MHz) while the method is executed on the device. For the second 10 experiments, the CPU availability is 74% and the CPU availability is 100% for the last 10 experiments. In these two sets of experiments, the computation is offloaded to the server according to the Optimal policy (i.e., the minimum of the on-device and on-server execution). As shown in the figure, `Lamp` makes about 12% wrong decisions at the beginning of the experiments.

Figure 4.7(b) depicts the details of experiments from 31 to 60 when only the available memory changes. In this group of experiments, the CPU availability maintains at 37% (i.e., 1 GHz) while the network bandwidth and latency are set to 1000 Kbps and 20 ms, respectively. In this set of experiments, the available memory changes from 512 MB (the first

10 experiments) to 1 GB (the second 10 experiments), to 2 GB (the last 10 experiments). For the first 20 experiments, the optimal decision is to execute the method locally on the device. In the last 10 instances when the memory is increased to 2 GB, the computation is offloaded to the server. Figure 4.7(b) shows great improvement on the decision-making with more training instances.

Figure 4.7(c) details the results when the network bandwidth changes while the network latency is set to 20 ms. In this figure, the server CPU availability is set at 74%, (i.e. 2 GHz) and the memory size is 2 GB. In this set of experiments, the network bandwidth changes from 300 Kbps (for the first 10 experiments), to 500 Kbps (for the next 10 experiments), to 1000 Kbps (for the last 10 experiments). For the first 20 experiments, when the bandwidth is less than or equal to 500 Kbps, the method is executed locally on the device. In the last 10 experiments, the Optimal policy offloads computation to the server where the bandwidth is increased to 1000 Kbps. Figure 4.7(c) shows a higher error rate of `Lamp` compared to Figure 4.7(b). This is likely due to the heavier impact of network bandwidth, which we will evaluate later.

We further provide detailed results for the experiments from 91 to 120 in Figure 4.7(d), when only the network latency changes. In this group of experiments, we keep the network bandwidth fixed to 1000 Kbps while the available CPU and memory are 37% (representing 1 GHz of a 2.7 GHz) and 2 GB. Here the computation is offloaded to the server when the latency is 20 ms for the first 10 experiments. In the second 10 experiments, the latency is increased to 50 ms and the Optimal policy executes the computation on the device. The optimal decision remains unchanged when the latency is further increased to 75 ms for the last 10 experiments.

In the previous 120 experiments, we set the data size to 19.5 KB (1 image) on average. To evaluate `Lamp` with respect to data size, we provide more details for the experiments from 121 to 150 in Figure 4.7(e). We keep the network bandwidth to 1000 Kbps and the latency to 50 ms while the available CPU and memory are 74% (representing 2 GHz of a 2.7 GHz) and 2 GB. In these experiments, the data size is 19.5 KB (1 image) on average

for the first 10 experiments, which is later increased to 39.03 KB (2 images) for the next 10 experiments, and to 58.5 KB (3 images) for the last 10 experiments. The Optimal policy executes the first 10 experiments on the device, while for the last 20 experiments, the computation is offloaded to the server according to the Optimal policy. Compared to the previous experimental results, Figure 4.7(e) indicates less error decisions of `Lamp` with respect to the data size, potentially due to its relative weight in all features as we confirm in Figure 4.7(f).

Overall, the above experimental results under various conditions show that `Lamp` has an error decision rate around 14.66% when compared to the Optimal policy. The error rates for always-OnDevice and always-OnServer executions are 48.67% and 52.33%, respectively. As we can see, most error predictions come from the dynamics of CPU availability and network conditions, both of which could change quickly. To study the relative importance of different features, Figure 4.7(f) shows that after removing the CPU availability or memory feature, the error rate increases to 22% and 21.9%, respectively, while the removal of bandwidth and latency leads to a 27% and 28% error rate, respectively. The removal of the data size feature leads to an error rate of 22%. Note that Figure 4.7(f) reflects the overall results based on 150 instances while previous results are based on 30 instances. Figure 4.7(f) shows that our identified features are all important, albeit that network conditions play a slightly more important role.

4.5.2 Performance Results

Having studied the adaptability of `Lamp Learning Model` , we now run experiments with more real-world applications to examine its performance under various environments. Table 4.1 summarizes the applications, the name of the method we are offloading, and their functionality. We have five different configurations in the experiments. In each configuration, the CPU, memory, data size along with the network parameters are dynamically changing as mentioned in Table 3.2.

Table 4.1: Description of Evaluated Applications in `Lamp`

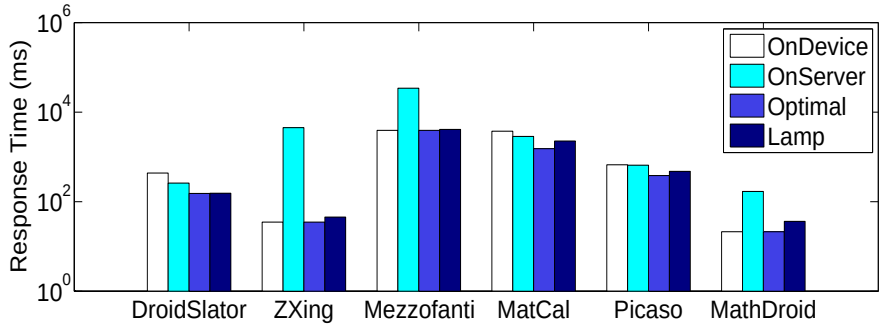
Application	Offloading Candidate	Description
DroidSlator [57]	`translate(String inWord, String toLang)` method of `ThaiDict` class	Android translation application
Zebra Xing(ZXing) [44]	`decodeWithState (BinaryBitmap)` method of `MultiFormatReader` class	Android product bar code reader
Mezzofanti [45]	`ImgOCRAndFilter(int[] bw_img, int width, int height, boolean bHorizontalDisp, boolean bLineMode)` method of `OCR` class	Android optical character recognition application
MatCal [61]	`times(Matrix B)` method of `Matrix` class	Android application for matrix operation
Picaso [62]	`project_and_compare(Bitmap bm)` method of `ReadMatlabData` class	Android face recognition application
MathDroid [63]	`computeAnswer(String query)` method of `MathDroid` class	Android Calculator application

Figure 4.8 shows the average response time and energy consumption of the the applications when different decision-making approaches are adapted for 50 experiments of each application. 10 experiments are conducted in each setting. We use `PowerTutor` [64] to measure the power consumption of the applications.

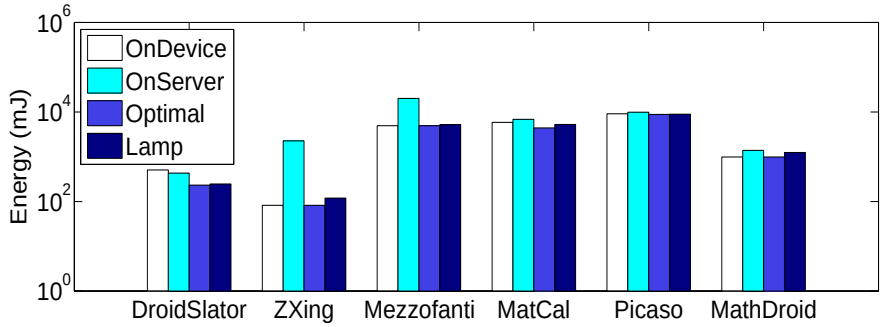
Figure 4.8 shows that compared to always-offloading (OnServer), with `Lamp`, the user's response time is reduced to 59.39%, 1.00%, 11.99%, 78.98%, 72.28%, and 21.44% of the response time if DroidSlator, ZXing, Mezzofanti, MatCal, Picaso, and MathDroid applications are always offloaded to the server. Correspondingly, with `Lamp`, the energy consumed on the smartphone is only 57.21%, 9.49%, 25.80%, 77.03%, 90.01%, and 77.97% of the energy consumed when these applications are offloaded.

On the other hand, compared to always local executions (OnDevice), `Lamp` also provides the user faster response time for DroidSlator, MatCal, and Picaso (35.40%, 60.23%, and 71.25% of local execution time, respectively). The corresponding energy consumption is about 48.61%, 90.14%, and 97.48% respectively, of the energy consumption when they are executed locally. For Mezzofanti, `Lamp's` performance is similar to those of local execution.

Compared to the Optimal policy, `Lamp` is expected to perform worse because of training and classification overhead along with some misclassification. However, for most of the time,



(a) Response Time



(b) Energy Consumption

Figure 4.8: Comparison of Response time and Energy Consumption

`Lamp` achieves the closest results to these of the Optimal policy. As shown in the figure, `Lamp` takes 1.46%, 29.54%, 4.82%, 48.29%, 23.47%, and 69.56% more time than the Optimal policy for DroidSlator, ZXing, Mezzofanti, MatCal, Picaso, MathDroid; and it consumes 6.03%, 45.12%, 5.23%, 19.30%, 1.02%, and 9.67% more energy of that consumed by the Optimal policy.

Among the evaluated applications, `Lamp` does not perform well for ZXing and MathDroid. This is because ZXing is purely data intensive and always has the best performance while executing on device [14]. MathDroid is neither data- nor computation-intensive. Looking into the applications, we find execution time on the device is very small. Thus training and classification time in `Lamp` becomes significant overhead compared to its actual execution time (in terms of percentage). If we look at the absolute values, they are very small (e.g., 119mJ for ZXing under `Lamp`).

Figure 4.8 also re-confirms that a static policy of always offloading or the default on-device executions may work for some applications, but not for other applications. In contrast, the offline Optimal policy can always lead to the best performance, while `Lamp` offers near-optimal performance.

Table 4.2: Accuracy (%) of Different Classifiers

	DroidSlator	ZXing	Mezzofanti	MatCal	Picaso	MathDroid
Threshold	74	0	0	38	60	40
Linear	82	10	10	72	40	100
SVM	80	92	100	62	40	80
Naive	74	92	88	44	80	20
Decision Tree	94	94	98	68	40	100
MLP	94	100	100	64	100	80

Table 4.3: Average Response Time (ms) and Energy Consumption (mJ) by Different Classifiers

	DroidSlator		ZXing		Mezzofanti		MatCal		Picaso		MathDroid	
	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy
Threshold	270.28	430	4511.3	2254	34200.4	20124	2857.02	6836	661.68	9120	64.8	1220.92
Linear	276.86	451.28	4459.14	2036.8	31858.6	18605	3462.02	5492	757.5	9209	24.3	988.38
SVM	241.98	455.84	118.12	255.76	3922.48	4934	3746.16	5842	762.5	9214	37.18	1053.08
Naive	279.82	437	230.62	256.95	7402.28	6756.8	3225.08	6856	521.46	9246	94.10	1287.42
Decision Tree	166.28	451.28	256.96	234.32	4618.24	5237.8	3142.76	5354	765.5	9247	29.3	993.38
MLP	154.42	246	45.16	119	4100.88	5192	2256.56	5266	473.44	8887	36.18	1080.65

4.5.3 Comparing to Other Offloading Approaches and Classifiers

Several classifiers have been used in previous studies, such as Threshold [14] and Linear-Regression [12]. In the evaluation, we also compare MLP's performance against those, including Threshold, Linear Classifier, Support Vector Machine (SVM), Naive Bayes, Decision Tree. Similar to MLP, we train each of the classifiers with 20% of all the instances

of these applications (Picaso, DroidSlator, ZXing, Mezzofanti, and MatCal) and test with the rest of the dataset and take the decision accordingly in the mobile device.

Table 4.2 shows the accuracy of different classifiers for these applications. According to the classifier's output (decision), we either offload or locally execute the methods for different applications and measure the response time and energy consumption. Table 4.3 shows the average response time and the average energy consumption for the test set (which is the rest of the dataset excluding the training set). Further, we have designed and implemented a learning based decision maker for offloading decision. Compared to the most existing decision-makers utilizing a static policy, we have shown that a static policy may worsen the applications performance. We have also designed and implemented a profiler of mobile applications to efficiently partition the applications in an optimal way to find the appropriate methods for offloading.

Table 4.2 shows that most of the time MLP has better accuracy than the other classifiers. SVM and Decision Tree also have better accuracy, response time and energy consumption. But as mentioned in section 4.2, these two classifiers do not support training on stream data.

Moreover, in Table 4.2, we can see that for MatCal, Linear has a better accuracy rate than MLP but MLP has faster response time and lower energy consumption (Table 4.3) than Linear. This is because different instances have different response time and energy consumption for OnDevice and OnServer executions for MatCal. For a particular set of instances, it is possible that Linear may have a better accuracy rate than MLP, but at the same time Linear may mis-classify some instances with high penalty (which may be correctly classified by MLP), resulting worse performance although its accuracy rate is better. In fact, we find that there are some instances with small matrix sizes that have better OnDevice performance. These instances are correctly classified by Linear, but Linear fails to correctly classify some matrices with high dimensions and makes the decision of local execution. On the other hand, MLP can capture the wide range of various feature values and perform more consistently.

4.6 Summary

Most of the existing studies have focused on how to make offloading, whether or not an offloading should be conducted received much less attention. In this Chapter, we have designed and implemented a learning based model for executing execution time to assist offloading decision. Compared to the most existing decision-makers utilizing a static policy, we have shown that a static policy may worsen the applications performance. Thus, in `Lamp`, we have designed and implemented a learning model based on MLP that can capture the dynamic nature of the relevant key factors and their relationship. Our experimental results show that `Lamp` can work near-optimally with existing mobile applications while making offloading decision.

4.7 Discussion

In our experiment, we have hand-picked the methods for offloading. A comprehensive offloading model should find these candidate methods for offloading automatically. In addition, not every method of an application can be offloaded. For example, some methods may access mobile device camera or other sensors. They have to be executed on the mobile device itself. In the next Chapter, we come up with a framework to find such a list of candidate methods and make an optimal partition of the applications for offloading.

Chapter 5: `Elicit`: Application Partitioning

5.1 Introduction

Some prior research [10–13,38,42] has investigated how to partition mobile applications and offload computation-intensive tasks to the more powerful counterparts such as clouds and servers. These studies either manually identify the methods for offloading [12,14,73] or use more fine grained offloading mechanism to offload threads [42]. It is possible to find the resource intensive method of an application manually to achieve the optimal performance gain. But the main drawbacks are (1) it requires programmer's knowledge to find the appropriate method to offload; (2) the cost is prohibitively high to offload an existing application in this way, given that there are hundreds of methods in an even very simple mobile application.

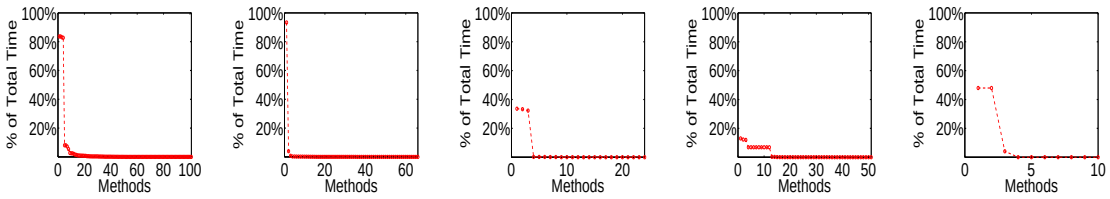
On the other hand, fine grain offloading can provide seamless thread migration without any modification of the source code and it also does not require any special user notation for offloading. But the thread migration approach has two major limitations:

- When migrating a thread, it requires a clone of the mobile OS image to be saved on the server side for executing the offloaded thread there. While offloading the thread, these two images must be synchronized continuously, which brings a lot of network overhead. For example, the overhead to offload a single method in Cloudlet [11] is around 100MB.

- VM based cloning is flexible for the application to be offloaded at any point during the execution, but it does not find the appropriate resource intensive computation to offload. For example, COMET [42] migrates a thread only when its OnDevice execution time exceeds twice of the round trip time from the mobile device to the server.

In this way, the offloading performance gain is not guaranteed: it is possible that the thread migration overhead may be greater than the performance gain obtained by offloading the rest of the execution of the thread.

Zhang et. al. [74] showed that the performance gain can be achieved by distributing an application execution between the mobile device and the server by adopting a shortest path algorithm to find an optimal cut to minimize the offloading overhead. Although this solution provides a generic approach for application partitioning, it does not consider many practical constraints for partitioning real-world applications. In practice, partitioning an application has to consider other factors. For example, it is possible to find a partition that gives the best performance gain, but at the same time the methods (in the optimal partition) can not be offloaded due to some practical constraints, such as the candidate methods may need to access mobile device's camera or other I/O devices. In addition, this work does not provide any guideline on how to profile the applications to find the bottleneck to offload.



(a) DroidSlator (b) LinPack (c) MatCal (d) MathDroid (e) NQueen

Figure 5.1: Methods Consuming the Most of the Execution Time

In practice, mobile applications' performance bottleneck can usually be narrowed down to a few methods. For example, Figure 5.1 shows the percentage of time (of the total execution time) that different methods are consuming in different applications. Here we have considered four different applications downloaded from Google Play (we are not part of the development of any of those applications) and one hand-coded application (NQueen).

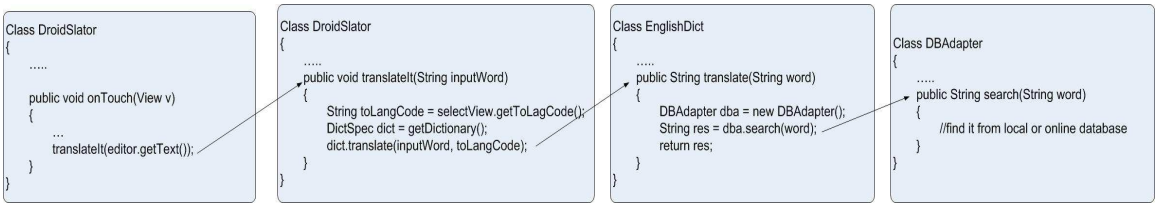


Figure 5.2: Call Flow of DroidSlator Application

While details can be fond in Table 5.1, these applications are: DroidSlator [57], LinPack [43], MatCal [61], MathDroid [63], and NQueen (hand coded). In the experiments, we have considered the execution time of the methods that can be offloaded (for example, the display rendering method for various graphic intensive applications consumes the most of the execution time of the application, but as those methods can not be offloaded, we have omitted them here. Section 5.6 describes how we find this set of methods in practice).

Figure 5.1 shows that only a few (five or six out of hundreds of methods in an application) are responsible for more than 80% of the total execution time. In addition to that, these resource intensive methods are actually invoked in a chain fashion.

Figure 5.2 shows an example of how the most time-consuming methods of DroidSlator application [57] (Android dictionary app) are invoked. Here the `onTouch()` method consumes most of the execution time of the application. It calls the `translatrIt()` method. Thus, the `onTouch()` method's execution time includes the execution time of `translateIt()` as well. In this way, `translateIt()` also includes the execution time of `translate()` and subsequently that of the `search()` method.

This example also shows that it is not possible to offload all the methods. For example, the `onTouch()` method interacts with the `View Display`, so it is not feasible to offload it. The `translateIt()` method interacts with the user input so it is not possible to offload it either. Two other remaining candidate methods are `transpalate()` and `search()` and we want to offload the one with less overhead compared to the offloading gain.

Therefore, in this Chapter, we propose to build `Elicit` (**E**fficient**l**y **i**dentify **c**omputation-**i**ntensive **t**asks in Mobile Applications for Offloading) to find the appropriate methods in applications for offloading. Considering the fact that even a very simple mobile application may consist of hundreds of different method calls, `Elicit` works by first profiling mobile applications to find the methods resource consumption, their dependency among each-other, and constraints which may hinder them from being offloaded. Then, considering the environmental setup (e.g., network latency and bandwidth between the mobile device and the server, size of the overhead data, etc.), `Elicit` dynamically chooses an optimal partition of the application for offloading based on the projected gains in terms of both the response time and the energy consumption. An `Elicit` prototype is implemented on Android Dalvik virtual machine. We evaluate it with a few popular Android application. The results show that `Elicit` can find the most resource consuming methods of the applications for offloading, which are consistent with previous studies [14, 73]. Our experimental results also confirm that `Elicit` can dynamically choose the optimal partition of applications for offloading the method and can improve the response time.

While details are provided in the later context, the main contributions of this Chapter include:

- `Elicit` can efficiently find the candidate methods that are not constrained and appropriate for offloading.
- `Elicit` dynamically chooses the optimal partition of mobile applications to offload to achieve the trade-off between the response time and the energy saving gains.
- A prototype of `Elicit` is implemented in the Android Dalvik virtual machine and evaluated with a few popular mobile applications from Google Play to demonstrate its effectiveness.

The rest of this Chapter is organized as follows. Section 5.2 presents our objective and constraints, and we propose our solution in section 5.3. We give an illustrative example

in section 5.4 and propose our optimal partitioning algorithm in section 5.5. We further present our implementation in section 5.7 and conclude our chapter in section 5.8.

5.2 Objective and Constraints

We can formulate the candidate methods finding as a constrained graph partitioning problem. Here we have a list of methods M, and each method $m \in M$ has the following members: i) execution cost on the mobile device, denoted as m_m, ii) offloading (the parameters) and execution cost on server, denoted as m_s, iii) class of this method: MC (Method Class), iv) list of global or class variables accessed by the method, denoted as V, v) list of the classes of V of this method, denoted as VC (Variable Class), vi) list of all the variables reachable from this method, denoted as RV (Reachable Variables), vii) list of all the methods reachable from this method, denoted as RM (Reachable Methods), viii) list of the classes of the methods in RM, denoted as MCC (Method Closure Class), and ix) list of the classes of all the variables of the methods in RM, denoted as VCC (Variable Closure Class).

Let us also denote the list of classes that can not be offloaded as CC (Constrained Class), and the method call graph as MCG. Each node of MCG represents a method and an edge e_{xy} represents whether a method x is invoking another method y. Assume that the execution cost of a method m in the mobile device is m_m and on the server (without considering the parameters' sending cost) is m_{swp}. In addition, the cost of invoking a method m from method x is denoted as e_{xm}, and the cost for accessing a variable (global or class) v is e_{mv}.

So the cost of sending and executing a method (including parameter access overhead) on the server is:

$$m_s = m_{swp} + \sum_{\forall x \in \{M-m\}} e_{xm} + \sum_{\forall x \in RV} e_{mx}. \tag{5.1}$$

And the cost saved by offloading a method is

$$b_m = m_m - m_s. \tag{5.2}$$

Thus, our goal is to partition the application's MCG into two partitions S and T and offload all the methods $m \in S$ to the server side such that:

$$Execution\ Time = \sum_{\forall m \in T} m_m + \sum_{\forall m \in S} m_s \tag{5.3}$$

is minimized, subject to the following constraints:

$$\forall m \in S\ \forall c \in m \cdot MCC \rightarrow c \notin CC \tag{5.4}$$

$$\forall m \in S\ \forall c \in m \cdot VCC \rightarrow c \notin CC \tag{5.5}$$

$$\forall m \in S\ \forall y \in m \cdot RM \textbf{ and } \forall x \in (M - m \cdot RM) \rightarrow e_{xy} \notin MCG \tag{5.6}$$

Algorithm 1 Find Reachable Methods()

```
1: for ∀m ∈ M do
2:    m · RM ← DFS(m)
3: end for
4: Begin Function{DFS}{$m$}: list
5:    list · add ← m · MC
6:    if m is visited then
7:       do nothing
8:    else
9:       for ∀x ∈ neighbours of m do
10:         list · add ← DFS(x)
11:      end for
12:   end if
13:   m is visited
14:   return list
15: End Function
```

Algorithm 2 Find Reachable Variables(m)

1: $list \cdot add \leftarrow empty$
2: **for** $\forall x \in m \cdot RM$ **do**
3: **if** $list$ doesn't contain $x \cdot V$ **then**
4: $list \cdot add \leftarrow x \cdot V$
5: **end if**
6: **end for**
7: $m \cdot RV \leftarrow list$

Algorithm 3 Find Method Closure Class(m)

1: $list \cdot add \leftarrow empty$
2: **for** $\forall x \in m \cdot RM$ **do**
3: **if** $list$ doesn't contain $x \cdot VC$ **then**
4: $list \cdot add \leftarrow x \cdot VC$
5: **end if**
6: **end for**
7: $m \cdot MCC \leftarrow list$

Algorithm 4 Find Variable Closure Class(m)

1: $list \cdot add \leftarrow empty$
2: **for** $\forall x \in m \cdot RM$ **do**
3: **if** $list$ doesn't contain $x \cdot MC$ **then**
4: $list \cdot add \leftarrow x \cdot MC$
5: **end if**
6: **end for**
7: $m \cdot VCC \leftarrow list$

5.3 Solution

With the problem formulated and metrics defined, we map our problem to the `Project Selection Problem` [46]. Kleinberg et. al. [46] propose this `Project Selection Algorithm` for profit maximization. Suppose there is a set of projects which provide profit, and a set of equipments which require some cost to purchase. Each project is dependent on one or more of the equipments. Kleinberg et. al. [46] have shown that the max flow min-cut theorem can give an optimal partition to find the list of projects and equipments to be chosen to maximize the profit. Kleinberg et. al. proposed their method for a dependency graph as

well, where some project execution depends on some of the other projects. Our goal is to find a similar set of methods for offloading to minimize cost by optimal partitioning. But in our set of methods, we may have many methods that can not be offloaded due to the previously mentioned constraints. So, we have to filter out the constrained methods first, only after that we may map the problem to Kleinberg method.

Algorithm 5 Candidate Method Selection Algorithm()

```
1: for ∀m ∈ M do
2:    find the members of m from Algorithm 1, 2, 3, and 4
3: end for
4: for ∀m ∈ M do
5:    if m · MC ∈ CC then
6:       discard m
7:    else if ∃x ∈ m · MCC and x ∈ CC then
8:       discard m
9:    else if ∃x ∈ m · VCC and x ∈ CC then
10:      discard m
11:   else if MCG has one or more edge from {M − m · RM} to {m · RM} then
12:      discard m
13:   else
14:      CAN · add m
15:   end if
16: end for
17: return CAN
```

Algorithm 6 Partition Algorithm()

```
1: CAN ← List of methods can be offloaded obtained from Algorithm 5 along with their
   estimated OnDevice and OnServer execution time
2: O ← list of methods to be offloaded according to Project Selection Algorithm [46]
3: return O
```

To filter out the constrained methods, at first we find the methods' member parameters mentioned in the previous subsection 5.2. Given the execution time m_m and m_s, class MC, the list of global or class variables V of the method (and their class VC), and the list of caller callee methods (details of how these information can be found is stated in implementation section 5.6), we build up the method call graph MCG from the list of the caller and callee

methods. To find the list of the reachable methods RM, we come up with Algorithm 1 which is a variation of Depth First Search Algorithm [75]. Once we find RM from Algorithm 1, we find the RV, MCC, and VCC according to Algorithm 2, 3, and 4, respectively. From these parameters, we discard the methods that violate the constraints 5.4, 5.5, and 5.6. Thus, Algorithm 5 finds CAN the list of methods eligible for offloading.

Once we get the list of the unconstrained methods CAN and their `Method Call Graph` MCG, we map our as follows. We convert this unconstrained MCG to `Method Selection Graph`. We introduce two nodes m and m' for each method $m \in CAN$ to the new graph. We also introduce two dummy nodes λ and μ. Each node m has an edge from λ and the edge has capacity $rank_m$ of that method, which represents the gain of executing the method on the server side. Each node m' (corresponds to the additional node introduced for each method) has an edge from itself to μ with capacity 1. We choose this edge capacity to be 1 to normalize it with the rank of the methods. Then we set the edges between m and m' with capacity $C = \sum_{m \in M} rank_m$. We also set the internal edge's capacity to be C as well. Thus our problem is mapped to `Project Selection Problem` and we can find the optimal set of methods to be chosen for offloading according to [46] with $O(VE^2)$ complexity, where V and E are the number of vertices and edges of the `Method Selection Graph`. We omit the details of the proof or correctness due to space limitation.

After we deduce the set of unconstrained methods from Algorithm 5, we find the optimal partition of the application for offloading by Algorithm 6. To make an optimal partition, we have to estimate the OnDevice and OnServer execution time, which we discuss in the previous Chapter 4.

5.4 An Illustrative Example

In this subsection, we illustrate our algorithm with an example. Figure 5.3 shows a method invocation diagram of an application. Here each white node represents a method and the solid arrow represents a method invocation incident from another method. The dark node

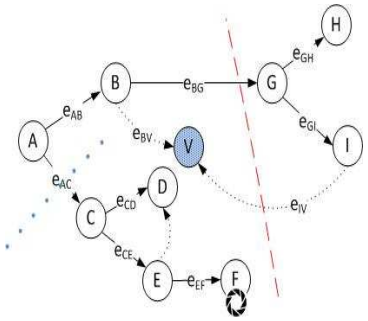


Figure 5.3: Method Call Graph

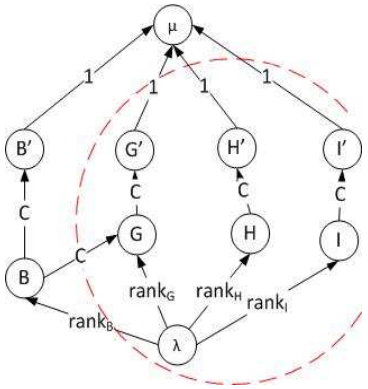


Figure 5.4: Method Selection Graph

represents a variable. The dotted arrow represents an access to a variable (global or class) modified by other methods. Each arrow has some associated cost with it. For example, when method A invokes another method B, A sends the parameters to that callee method B. When we offload the callee method, these parameters have to be sent to the server as well, which demands time and energy consumption to send it over the networks, which is denoted by e_{AB}. This cost can be found by monitoring the time and the energy consumption when these parameters are sent to the server side from the mobile device. Similarly, if an offloaded method I accesses a global or class variable V modified by other method B, this variable has to be sent over the network so that the clone method on the server side can

access these variables and execute flawlessly. We denote this cost as e_{VI}.

Suppose in this graph we find that method C and G consume most of the resources. If we offload C, methods D, E, and F are executing on the server side as well. Method F is accessing a camera, as a result it is not possible to execute method F on the server side. If we still decide to offload method C, the execution has to be transferred to the mobile device again while F is accessing the camera. To limit this back and forth executions, we should not offload C. To identify such constrained methods, our Algorithm 3 finds the list of classes, MCC of the reachable methods in RM of method C, which is $\{D, E, F\}$. As we have found that MCC of C includes a class accessing camera (from method F), we should not offload C.

Similarly, by offloading G, we are also executing H and I on the server side. So when method I is executed on the server side, it has to access variable V. As a result, while offloading method G, we have to send variable V to the server, which will cost us e_{IV}. In order to find such a set of variables, we deduce RV of method G. This set RV of method G has to be offloaded to the server side in order to offload the method G to the server. Note that here if G's VCC (the classes of RV) includes any of the constrained classes C, we do not offload G as well.

In addition, if we offload a method whose RV or RM is updated in parallel from other methods executing on the mobile device, we have to communicate again from the mobile device to the server. To minimize such overhead and inconsistency, we do not allow these methods to be offloaded. As illustrated in Figure 5.3, if method $A, C, D, E, , F$ or B is accessing the RM or RV of G, we do not offload G.

Thus, we find the `Method Call Graph` of B, G, H, and I and convert it to `Method Selection Graph` shown in Figure 5.4. In this graph 5.4, we introduce one additional node m' for each of the methods m. So for each of B, G, H, and I; we add B', G', H', and I' in Figure 5.4. We introduce two additional nodes λ and μ. The capacity of the edges between λ and B, G, H, I is set to their corresponding rank. The capacity of the edges between B', G', H', I' and μ is set to 1. The capacity C between any other two nodes is set to the

summation of the ranks of the methods in this graph ($C = rank_B + rank_G + rank_H + rank_I$). Figure 5.4 shows the corresponding `Method Selection Graph`. The dash curve line shows a min-cut partition where we are offloading the methods G, H, and I.

Algorithm 7 Partitioing Algorithm()

```
1: Train the MLP learners with labelled data
2: CAN ← List of methods can be offloaded obtained from Algorithm 5
3: for each m ∈ CAN do
4:    predict the OnDevice and OnServer execution time by MLP considering the bandwidth, latency, data size, server-side CPU and available memory
5: end for
6: O ← list of methods to be offloaded according to Project Selection Algorithm 6
7: if O is empty then
8:    execute the app locally
9:    monitor the OnDevice execution Time and
10:   train the OnDevice Time learner with the new data
11: else
12:   offload the methods m ∈ O
13:   monitor the OnServer Time and
14:   train the OnServer Time learner with the new data
15: end if
```

5.5 Finding Optimal Partition

To find an optimal partitioning according to Algorithm 6, we adopt MultiLayer Perceptron model to estimate the response time for both local OnDevice and remote OnServer executions in a dynamically changing environment where bandwidth, latency, data size, server's available CPU and memory change in Chapter 4. Algorithm 7 presents the pseudo code for this dynamic application partitioning and training. With Algorithm 7, we execute the application locally if there is no method to offload. If any method is offloaded, we monitor the response time and the energy consumption while it is executed on the server side. These values are used for training the learning model for the OnServer response time. We also update the OnDevice learning models in a similar fashion when the methods (and application) are executed locally.

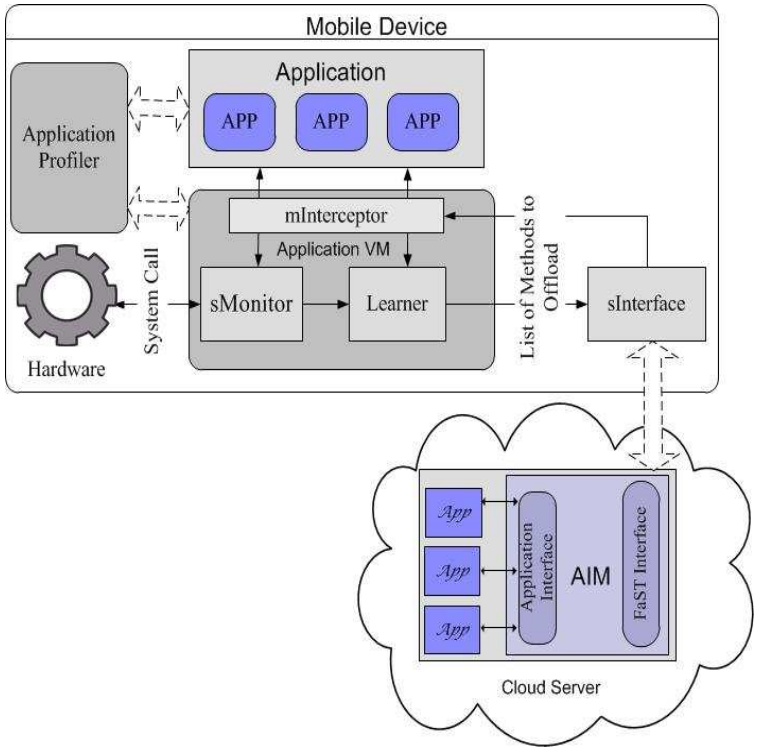


Figure 5.5: Components of Computation Offloading Model

As we have discussed, it is desirable that computation intensive methods of any existing application can be offloaded transparently. For this purpose, we adopt the offloading mechanism `FaST` mentioned in Chapter 3.

Figure 5.5 shows the various components of our computation offloading model `Elicit`. In high level we profile the applications off-line to find the set of feasible candidate methods for offloading. Whenever an application starts to execute, we monitor the system environment to estimate the performance of the methods if they are offloaded and find a list of methods for offloading. By offloading these methods, we achieve optimum gain in response time savings. We offload these methods to the server side and returns the result back to the applications running in the mobile device transparently by adopting `FaST` (Chapter 3).

There are six major components for the learning model in `Elicit`:

- **Application Profiler**

`Application Profiler` gets the predicted OnDevice and OnServer execution time from the `learner` and finds the list of methods to be offloaded according to optimal partitioning Algorithm 7).

- **sMonitor**, **Learner**, **mInterceptor**, and **sInterface**

  We implement **sMonitor** and **Learner** according to `Lamp` in section 4.4. Similarly, we implement `mInterceptor` and `sInterface`according to `FaST` as mentioned in section 3.3. We have also implemented the `server side` accordingly.

- **Server-side** We design `Server-side` as mentioned in section 3.3.2.

5.6 Implementation

We implement a prototype of `Elicit` in Dalvik VM for Android applications. We have modified the Cyanogemond [58] open source distribution to profile the applications without modifying the application to find the methods' members and the method call graph. After profiling, we partition the application optimally to achieve the highest gain regarding the system environment.

To profile applications, we have modified the instructions for a method invocation and method return. In Dalvik, whenever a method is invoked, it is translated to a method invocation instruction. The caller method's program counter and frame page are saved and then the callee method starts its execution. When the callee method finishes its execution, it returns to the caller method. The return instruction saves the return value to the caller method's return address and starts the caller method by retrieving the program counter and the frame pointer of the caller method. We have modified the method structure of the Dalvik VM so that it keeps records when a method is invoked and when it returns to the invoking method. From these two timestamps, we keep track of the execution time of the methods. We use PowerTutor [64] to measure the methods' energy consumption. We also keep track of the parent class of the methods and the caller of each method. Such information is available from the class information of each method structure.

Furthermore, whenever a method accesses a variable, it leverages two separate instructions (IGET and IPUT namely) to retrieve and save values. We also trap the IGET and IPUT instructions to keep track of the variables (and their parent classes) that are accessed by the methods. To offload these methods, we have to send these parameters and variables (with GET tag) to the server side. Once the method has successfully finished its execution and returns back to the mobile device, we save the return value and synchronize the modified variables (the variable having PUT tag). Here if a method is accessing a variable related to mobile device's I/O, camera, or similar sensors, we do not offload it. We do this by examining the variables' parent class which is fetched from the variable's class structure in Dalvik VM.

In this way, we obtain the parameters for Algorithm 5. These parameters include list of variables (and their classes) accessed by methods, method call graph, methods' execution time, methods' parent classes, etc.

Then we conduct analysis to find the optimal partition of an application according to Algorithm 6. As discussed in section 5.2, we have to discard the methods that access mobile device equipments (not exclusively, camera, sensor, view, etc.). To find this list, we populate a list from Android Java definitions. Based on the list of methods, their parent class, and the global and class variables (and parent class of these variables) we discard these methods (and their parents) to be considered list for offloading.

We find the list of methods to offload along with the variables' states that must be synchronized before and after offloading. We intercept those methods' invocations and offload them according to `FaST` (Chapter 3).

5.7 Evaluation

We evaluate our `Elicit` prototype with five different Android applications. Table 5.1 describes the applications and their functionality. We choose these applications considering

their different characteristics: DroidSLator and MatCal are both computation- and data-intensive. LinPack and NQueen are purely computation-intensive while MathDroid is data-intensive. Moreover, most of them are the applications being evaluated in the previous studies, where the offloaded methods were hand-picked. By evaluating them in `Elicit`, we are able to tell whether `Elicit` can efficiently identify the same method for offloading.

Table 5.1: Description of Evaluated Applications in `Elicit`

Application	Offloading candidate	Total number of methods	Description
DroidSlator [57]	→ `translate(String inWord, String toLang)` method of `ThaiDict` class : no global variable	100	Android translation application
LipnPack [43]	→ `linpack()` method of `LinpackJavaActivity` class : x0 (PUT) of `LinpackJavaActivity` class	66	Android CPU and Memory Benchmark
MatCal [61]	→ `times(Matrix B)` method of `Matrix` class : m (GET), n (GET), and A (GET) of `Matrix` class	24	Android application for matrix operation
MathDroid [63]	→ `computeAnswer(String query)` method of `Mathdroid` class : calculator (GET) of `Mathdroid` class	51	Android Calculator application
NQueen	→ `findSolution(int board[],int n, int pos)` method of `NQueen` class : no global variable	10	Android application for NQueen problem

For each of these applications, we first find the optimal partition of the applications and find the method to be offloaded. Table 5.1 shows the method which is found to be optimal to be offloaded based on Algorithm 7. In the table, each partition method shows the method name, parameter list, and global variables that are accessed by this method (and its subsequent methods). Each global variable has a tag GET or PUT, which indicates whether a variable is accessed by the method (GET) or modified by the method (PUT).

In our experiments, `Elicit` indeed finds similar methods as the candidate to be offloaded as in previous studies. For example, in Young et. al. [14], DrdoidSlator application [57] is evaluated. Our candidate is the `translate` method, which was found manually to be the most resource intensive method in [14]. Similarly, for MatCal [61] and MathDroid [63], we find that the applications are partitioned in the `times` and `computeAnswer` methods which were also found to be the most resource intensive methods in POMAC [73]. Here we

have added two new applications, LinPack and NQueen, for which we have found `linpack` and `findSolution` methods to be offloaded according to the optimal cut. Note that here each application has many methods (even a hundred excluding the basic Java methods like `String Compare` or `println` as shown in Table 5.1), and we are offloading the most resource consuming method(s) of the application for optimal performance.

Table 5.2: Gain Ratios of the Applications in Different Environments

	DroidSlator		LinPack		MatCal		MathDroid		NQueen	
	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy
LAN	5.30	4.3	1.01	2.56	4.05	1.06	1.05	0.97	19.62	11.10
WLAN	3.95	2.8	0.98	2.75	3.73	1.68	0.99	0.96	19.61	11.01
802.11	1.21	1.1	0.97	2.60	4.34	1.15	1.09	1.07	19.44	10.99
4G	3.72	2.7	0.98	2.62	1.31	1.01	0.99	0.97	19.26	10.89
3G	1.01	1.2	0.97	2.56	1.90	1.37	1.05	1.03	18.93	10.71
Average	3.03	2.3	0.98	2.62	3.07	1.25	1.02	1.01	19.37	10.94

Next, we experiment `Elicit` in different environments. We want to (1) empirically evaluate the offloading performance and (2) evaluate that the methods suggested by `Elicit` for offloading can achieve the trade-off between the energy consumption and the response time savings. We evaluate our prototype in a Google Nexus one with 1 GHz CPU and 512 MB of memory. We have five different configurations in the experiments as mentioned in Table 3.2.

Table 5.2 shows the gain ratios of the applications in different environments. Note that the training and classification overhead of the MLP has been accounted. The final average of the gain ratios are shown in the last row. We find that for DroidSlator, MatCal, MathDroid, and NQueen; the gain ratios are greater than 1. MathDroid has both ratios very close to one. For MathDroid, it does not save that much time and energy by offloading. For all applications, mostly the gain ratios are higher in LAN and WLAN settings compared to the 3G or 4G, which is expected, as the bandwidth gets higher and the latency gets lower, our

Algorithm 7 can find a better partition to save energy and response time and offload them accordingly. The better network condition also enhances the performance of the offloaded methods. Here we have found that network condition plays a very important role on the offloaded methods' performance, which is consistent with previous studies [12] and [73].

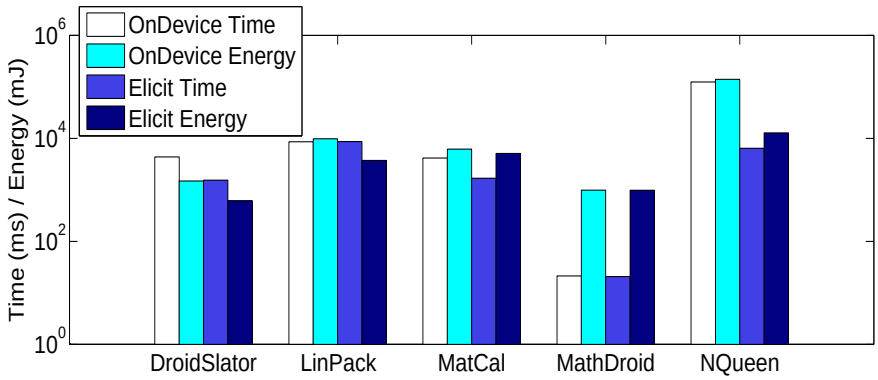


Figure 5.6: Response Time and Energy Consumption of Different Applications

Figure 5.6 shows the average response time and energy consumption of the 50 experiments in the 5 different environments. In this figure, the y-axis shows the response time in millisecond and energy consumption in millijoule. Note that y-axis is in log scale. Figure 5.6 shows that by offloading: DroidSlator, LinPack, MatCal, and NQueen can save significant time and energy. `Elicit` reduces 3.03x, 3.07x, and 19.37x response time of DroidSlator, MatCal, and NQueen compared to the on-device execution. `Elicit` can also save 2.3x, 1.25x, and 10.94x energy consumption for these applications, respectively. We also found that `Elicit` can not save significant time or energy for the MathDroid application. In fact we have found that most of the time it is optimal to execute MathDroid on the device itself. For Linpack, `Elicit` saves a lot of energy (2.62x) but slightly increases the response time (0.98x). In separate experiments, we have found that executing LinPack on the device does not deteriorate the response time but consumes a lot of energy (2.62 times more

than offloading). By offloading, although the response time increases a little bit (2.04%), eventually we may save a lot of energy (262.37%).

5.8 Summary

The increasing popularity of smartphones drives the fast development of mobile applications. For resource-intensive mobile applications, there is an increasing demand to offload to more powerful counterpart, such as clouds and servers. While most of the existing studies have focused on how to offload, little research has been conducted on how to automatically find the most appropriate methods to offload. In this study, we have designed and implemented `Elicit`, a profiler of mobile applications to efficiently partition the applications in an optimal way to find the appropriate methods for offloading. Extensive experiments have been conducted to evaluate `Elicit` and the results show that `Elicit` can work with existing real-world mobile applications and efficiently find the best offloading methods to reduce the response time.

Chapter 6: Storage Augmentation

6.1 Introduction

The increasing popularity of mobile devices has also caused the demand surge of pervasive data accesses, such as for photos, across a user's different storage space, such as her iPhone and desktop computer. To respond to such a fastly increasing demand, the current research and practice have provided many solutions, such as `Dropbox` [16], `Google Docs` [18], `Amazon s3` [27], `Windows SkyDrive` [17] and `SME Storage` [31]. They mainly rely on cloud-based services or a server-based approach.

These centralized cloud or server-based approaches [27] [16] [17] typically require a user to store all files on the storage owned by the service providers, which risks data security, privacy, and availability. For example, it has been reported on the news about Mark Zuckerberg's pictures leak incident in Facebook [28], DropBox account breach with wrong passwords [29], Amazon's data center failure in 2011 [30], etc. Some modern file systems [31] [19] [32] have taken into account user's storage to avoid third party storage compromise. But they maintain a strong consistency model for different types of files, resulting in unnecessary and heavy performance overhead. For example, smartphones are often associated with consuming and producing data which are mostly non-editable (photo, music, and video) and more than 27% pictures are taken by smartphones [33]. For these files, a strong consistency model is an overkill.

On the other hand, today an average user typically possesses multiple computers, such as personal home computers, office computers, and mobile devices. While the storage of one device at a time may not be sufficient for storing all data files of the user, the total storage space is often large enough to store all files owned by the user. Even when the total

storage space of a user is not large enough, it is relatively cheap to buy additional storage nowadays as one-time cost.

Therefore, in this Chapter, we aim to design and implement a system to **v**irtually **U**nify **P**ersonal **S**torage (vUPS) for fast and pervasive file accesses. With vUPS, all participating devices of the user are transparently and seamlessly unified. A file can be accessed through a web browser, considering that today the web browser is the vehicle for end-user's accesses. In vUPS, there is no central node that maintains the file system states. Instead, any device can serve as the server when it is been actively used. To minimize data transferring, only meta-data is pro-actively accessed while data files are accessed in an on-demand fashion. In vUPS, different types of files are treated with different consistency policies in order to balance the consistency and the maintenance overhead.

To evaluate vUPS, we have implemented a prototype based on HTML5 and Javascript. The prototype provides a standard API for other web and desktop applications to access and manipulate vUPS data. We have conducted experiments with micro-benchmarks and also compared to DropBox. The results show that vUPS can offer a similar user experience to DropBox. Our contributions in this Chapter include:

- vUPS provides an alternative solution to existing cloud-based or other centralized approaches for responding to the demand surge of quick and pervasive file accesses across multiple devices owned by a user.
- By differentiating and treating different types of files, vUPS strives to achieve a balance between the file consistency and the maintenance overhead.
- With a web browser interface and a standard file access interface, vUPS can be adopted by other applications to transparently and seamlessly access personal files.

The rest of this Chapter is organized as follows. Section 6.2 provides the design of vUPS. We present the trade-off between consistency and availability in section 6.3. Our implementation and evaluation details can be found in section 6.4 and 6.5. We further summarize our work in section 6.6.

6.2 vUPS Design

Instead of a centralized architecture, vUPS adapts a flexible P2P architecture. Figure 6.1 illustrates the architecture of vUPS that runs across multiple devices, such as a user's home computer, an office computer, a laptop, and a smartphone.

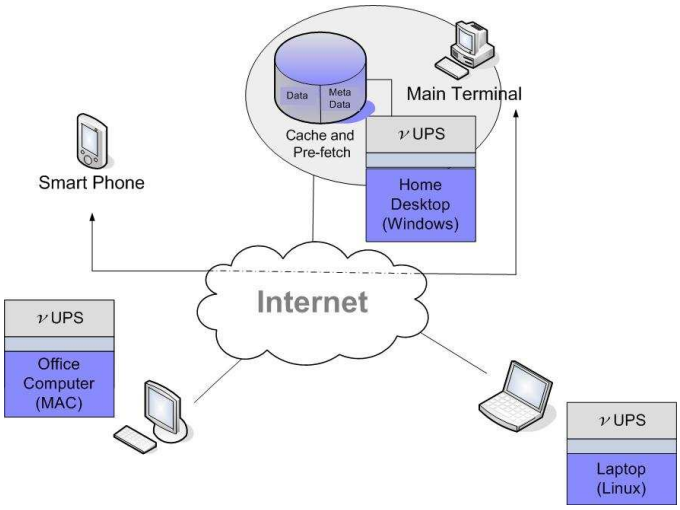


Figure 6.1: The Architecture of vUPS

In vUPS, the participating devices peer with each other to virtually form **a** single and global storage space. Communications (e.g., to fetch data or to execute a user's commands) between devices are based on RESTful web services [76]. With such a P2P architecture, the device that the user is actively using as the main device (i.e., on which the user initializes her request) is responsible for supporting the main functions of vUPS. For this reason, we refer to this device as the `main terminal` of vUPS. Currently, at each time, there is only one main terminal. As the principal component of the vUPS, the main terminal is responsible for maintaining the vUPS namespace. In our design, the vUPS namespace is stored in a logically separate site. Once the main terminal is activated upon user accesses, the namespace will be loaded. In practice, this namespace can always be stored on the user's

home desktop computer. All other participating devices are referred to as passive terminals, as they are mainly responsible for responding to the requests from the main terminal. In addition, users and other applications can interact with vUPS via vUPS APIs. Currently, we have designed a web-based user interface based on vUPS APIs.

Note that when a user actively uses her mobile device as the main terminal, it can deplete the limited battery power soon, because we expect that the main terminal could be relatively stable and stay on-line for a long time. Thus, in vUPS, the main terminal functions can be delegated by the mobile device to a more powerful computer, such as the home desktop (as shown in Figure 6.1) or the office computer.

When an application or a user needs to access a file, vUPS first finds the real device hosting the file based on the proper mapping in the vUPS namespace. vUPS resolves this mapping via the user name, device ID, resource path and operation type as: http://<usr>.vUPS.com/DeviceID/Path&Operation. Note that in the case of a delegated main terminal, the actual file transferring happens directly between the two involved devices without involving the main terminal.

To support the desired functionalities, we design vUPS with the vUPS API, the vUPS Controller, the vUPS Synchronizer, the vUPS Data Manager. Figure 6.2 depicts the design with these major components. Next we discuss each of these components.

- *vUPS API:*

  To provide an interface of vUPS for users and applications, we develop vUPS API. These vUPS APIs support typical file operations, such as create, delete, read, write, as well as typical directory operations. To provide an easy access to users, we further develop a Web browser based user interface based on vUPS APIs. It uses HTML5 and Javascript to make it accessible from heterogeneous platforms such as smartphones, desktop and office computers that may run different operating systems.

- *vUPS Controller:*

  The vUPS controller consists of two modules. The first one is the bootstrapping module or bootstrapper. Basically, when the main terminal is accessed by the user,

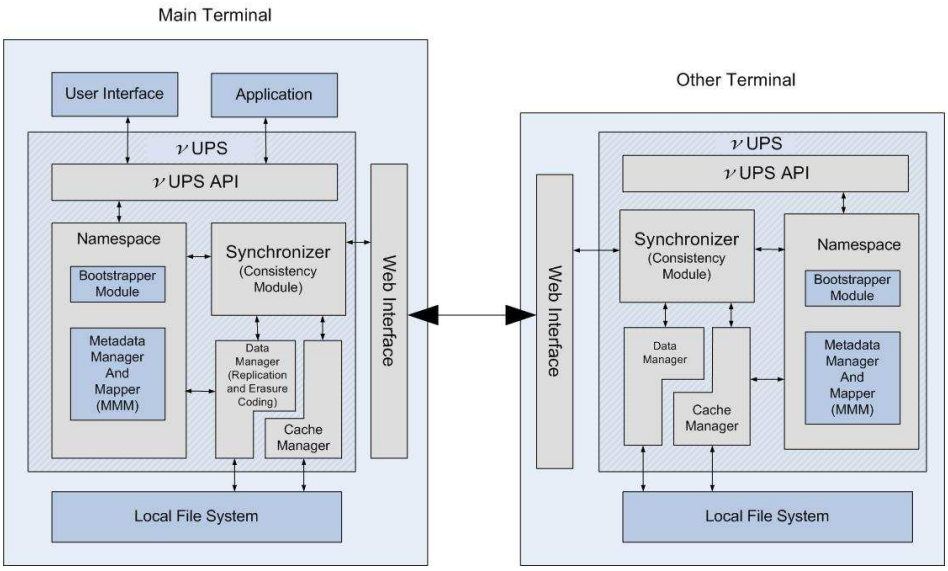


Figure 6.2: vUPS Components

it first needs to load the vUPS namespace, which contains the metadata and the directory hierarchy of the vUPS file system. When a new device wishes to join the system, it also provides the bootstrapping information so that any device can be added or removed from the system.

The second module is the metadata manager and mapper (MMM). Metadata is crucial to vUPS and strong consistency is maintained for metadata. MMM also maps files in the vUPS system to their physical locations in different devices. When a file is created by the user, a new entry is created to the appropriated location in the namespace. Accordingly, a file deletion removes the entry from the namespace.

- *vUPS Synchronizer:*

  Maintaining consistency is essential to any distributed file system. vUPS Synchronizer is responsible for consistency maintenance among multiple copies of a file on different devices. As we shall discuss later, vUPS uses a non-traditional consistency model based on the file types in order to balance the overhead and the performance. Note that because the namespace is loaded on each participating device, the vUPS is also

responsible for maintaining the consistency of namespace.

- *vUPS Data Manager:*

  The data manager deals with the physical storage of the data in the local disk via traditional filesystem APIs. A particular notable function of the vUPS Data Manager is to manage cache for better user response time. As the core of the cache management, a LRU (Least Recently Used) replacement policy is used for cache management.

With these components, they work together in a collaborative manner. Basically, the namespace of the vUPS is often stored in a reliable device or a web site if a bootstrapping site is used. A participating device needs to contact this bootstrapping storage to load the initial namespace as well as the main terminal address.

When a file operation takes place (either by the user or by some application) through vUPS API, vUPS API passes the request to the vUPS Controller, which finds the appropriate file through the MMM.

The MMM then checks with the Synchronizer for cache validation. For read operation, the Synchronizer checks the cache and validates the cache if it is found in the cache. If it is not valid, then the Synchronizer finds the physical location of the data from the MMM. Then the Synchronizer either contacts the local or remote Data Manager to fetch the data. A remote Data Manager is invoked via the web interface. Whenever a request is received from the web interface, the Synchronizer enforces the operation via the local Data Manager. If the operation is write, then the Synchronizer finds the devices that contain the replicas and/or the cached copy of the data. Then the Synchronizer updates its local cache and data (if any) through the Data Manager. It also propagates the update to all the relevant devices via appropriate web service interfaces.

Note that vUPS relies on a storage place to store the initial namespace. This single point of failure can be eliminated by replicating the namespace among all the devices. This may however increase the maintenance overhead.

6.3 Consistency and Availability

Achieving availability and strong consistency at the same time is a major challenge in a distributed file system [77]. Traditional file systems replicate the data across multiple devices to increase availability, which also increases the overhead for consistency. Popular file systems have different policies to make the data consistent among replicas. For example, Coda [78], Ficus [79], and Ivy [80] apply optimistic concurrency control between the replicas. Coda resolves consistency conflict using version controlling. Bayou [81] provides high availability with a weak consistency model. Bayou was mainly designed for intermittently connected devices, which is also suitable for mobile devices. With pervasive network connections today, it is ideal to use continuous synchronization and have quick consistency. For example, cloud storage like Google Drive [56] uses strong consistency, while DropBox [16] applies continuous consistency between the replicas.

Both the optimistic and the continuous consistency models suffer from the scalability issue. In addition, continuous consistency also suffers from the overhead for continuous updates. That is, even a byte change in the data may result in a lot of network traffic (several thousand more than the update itself). Such network overhead is amplified by the number of devices in the system.

Observing the fact that strong consistency is not required for every type of files, vUPS has different replication and consistency policies for different types of files. Today a user often owns much more media data (e.g., audio, video, and image) files than before. For example, it has been shown that a typical Windows user has 11.6% image files of the total storage [82]. This number is increasing due to the vast use of smartphones and tablets, because they are generally used to take pictures or videos. On the other hand, the documents and other files only consist of a small portion of the storage (i.e. 2.5% of the total file system [82]). Note that media files are not frequently changed. Only the corresponding metadata (favorite, ratings, etc.) may be modified. Therefore, we may relax the consistency model for media files as they are often non-editable, while the replicas of

other documents have to be consistent as they are frequently edited. In addition to that, fetching the document files on demand is not expensive compared to that for media files as the media files size [83] is often bigger than that of documents files [82] on average.

Thus, in vUPS, files are divided into two categories: editable and non-editable. The audio, video, image, and program files are considered as non-editable. On the other hand, all other files, including doc, xml, and text files, etc., are categorized as editable files. Note that these non-editable files can also be updated very occasionally, but vUPS only maintains weak consistency among their replicas. In the current implementation, vUPS differentiates different types of files based on their filename extensions.

With two categories of files, vUPS has the following two different policies for different types of files: (1) limited availability with strong consistency; (2) high availability with weak consistency.

6.3.1 High Availability with Weak Consistency

Considering the different types of popular files [84], we categorize the popular video, audio, and image files as non-editable files. Although in our current design these files are recognized solely based on file extensions, any self-learning clustering algorithm can classify the files over time based on the modification history. These non-editable files are seldom modified. Thus, the access to update rate is often high for these files. Moreover, the size of these media files is often larger than the editable files such as doc or xml [83] [82]. Caching and replicating these files on every possible device improves the access time to those files and results in less network traffic, but it also increases the overhead of maintaining consistency between copies.

As these files are seldom modified, vUPS follows a weak consistency model between the copies. A user can request the file from any device. The device may contain the data locally or fetch it from other devices (through the main terminal) and cache it. Whenever a modification takes place in a device, the change is reflected on the local/cached copy. The user does not need to wait for the update being propagated to other devices. Similar to

Bayou, vUPS propagates updates during pairwise contact. This epidemic approach ensures that as long as the participating devices are not permanently disconnected, the writes will eventually reach every copy [85].

For non-editable files, vUPS follows an invalidation protocol to notify the changes to other copies. As an invalidation protocol only notifies the changes, it saves network traffic and latency as the real update is not sent. The application in the devices may or may not pull the update for that file, depending on the policy of that application. vUPS aims to support application-specific conflict resolution, and each application may provide specific dependency check for updates. If the dependency check is valid, then the update takes place. Otherwise, conflict arises and the updates are either merged or marked invalid. Unlike scheduling or database transactions, media files may not have conflicting updates. Thus the merging procedure for media files ensures the latest update to a particular file is applied when any conflict arises. To detect the latest update, vUPS has vClock (vUPS vector logical clock) for each file and each folder. Each file has a vector of the Lamport logical clock [86], where each entry in the vector represents the timestamp for each device associated with that file. Whenever a file is created, a vector timestamp is created with entries for each device where the file is replicated. In addition to that, whenever that file is cached, an additional timestamp entry is added to that vector. If a cached copy is deleted or a replica is removed, the corresponding entry is removed from the vector. For every update from any device, the logical clock for that device in the vector is incremented. Whenever a device joins vUPS, all the data are updated to the latest vector. Thus, the copies are always up-to-date and synchronized if at least one copy of the data is always online. If only one copy of a file is kept connected to the vUPS and all the other copies are offline (that is, only one copy is mutual-exclusively online), then the file is updated if the timestamps of all the vectors are greater than the previous values. If all the entries in the vector are smaller than the new values, then it is not updated. Otherwise, the file is marked as conflicted and both copies are kept for the user to resolve manually. Thus, vUPS ensures the total ordering for the updates between the devices provided at least one copy is always online.

Note that, for non-editable files, the metadata may be changed frequently (a user may change the rating of movies, tag of pictures, etc.), but vUPS considers metadata as editable data, for which strong consistency is maintained among replicas.

Whenever a non-editable file is invoked by a device, the file is cached on the device and in the main terminal according to the caching policy. As the access-to-update ratio is higher, leaving a copy behind will improve the access performance [87]. So, when that file is closed, it is synchronized (if necessary), and a copy is left in the cache. The namespace contains a callback reference to that address for providing response from that copy to other devices.

6.3.2 Limited Availability with Strong Consistency

For copies of the editable files, vUPS maintains strong consistency. As strong consistency is not scalable, vUPS maintains a minimal number of copies to maintain the availability and consistency to get the best of both worlds.

Similar to non-editable files, whenever an editable files is accessed by a device, the file is cached on the device and in the main terminal according to the caching policy. All the modifications take place on the cached copy and are propagated to remote copies (local read/remote write). When that file is closed, it is synchronized (if necessary), and the local cached copy is deleted. The callback reference is also deleted from the namespace once the file is closed. That is, vUPS enforces a strong consistency policy between the copies. It sends the update operation where data should change (active replication).

As the access-to-update ratio is lower, keeping a local copy close to the device may not be better [87] as it may send frequent updates to all the replicas while these updates may not be accessed by the user. In this case, keeping sending these updates can waste bandwidth and may not be scalable. So it deletes all the cached copies once the read/write operation in the cached copy is completed. This approach makes vUPS more scalable. As the average size of editable files is smaller than that of non-editable files, bringing these files to the cache when needed does not affect the performance too much. In addition, for these

types of files, a typical replication policy results in less network traffic.

Assume the failure probability of one machine is p and the number of replication is r. Then, the failure probability of all replicas is:

$$M(n,p,r) = (p^r \times \sum_{i=0}^{n-r} p^i \times (1-p)^{(n-r)-i}) \tag{6.1}$$

Let the unavilability probability of one machine due to network is q and the number of replication is r. Then, the unavailability probability of all replicas due to network failure is:

$$N(n,q,r) = (q^r \times \sum_{i=0}^{n-r} q^i \times (1-q)^{(n-r)-i}) \tag{6.2}$$

$M(n,p,r)$ and $N(n,q,r)$ are independent from each other. So, the availability of the over all system is:

$$A(n,p,q,r) = 1 - (M(n,p,r) + N(n,q,r) - M(n,p,r) \times N(n,q,r)) \tag{6.3}$$

The typical disk failure rate is 4% [88]. If we take this into account, the availability of vUPS can be deduced to be more than 96% for a typical system with four machines with no replication. We can get more than 99% availability by keeping a backup (a replica) of each file, which is comparable to amazon services (with a failing rate 0.1-0.5% [88]).

In case of a terminal failure, the active requests for that terminal is re-directed to other terminals with replicated copies. When the device joins vUPS again, the replica in that device is checked again (with SHA-1) with other replicas and synchronized if necessary.

vUPS is also highly robust in the sense that any terminal can be initiated, allocated, deleted or backed up without interrupting the system. vUPS also allows adding or deleting a disk, allocating storage, moving data, or interrupting system availability. This design allows all data to be remained online and available to clients and other terminals.

6.4 Implementation

We have implemented a prototype of vUPS with Java and HTML. The user may add devices with any operating system and can browse, access, and store data in the vUPS storage consisting of those devices. In addition to the user interface, we have implemented the vUPS APIs in Java for applications. Figure 6.3 shows a screen-shot of vUPS user interface.

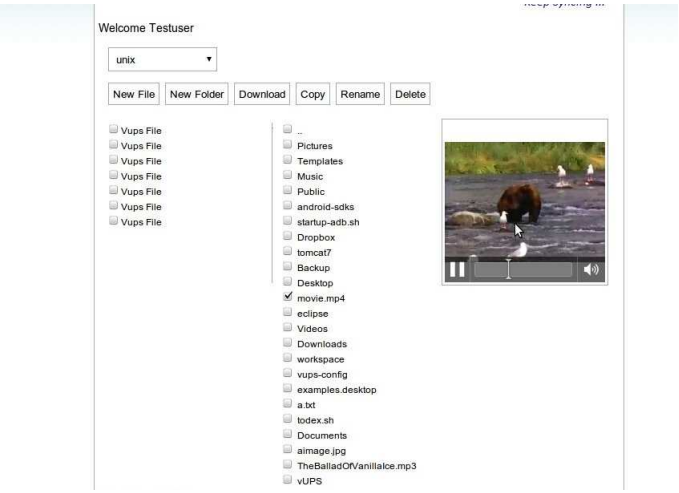


Figure 6.3: vUPS User Interface

The *vUPS APIs* are built with Javascript and HTML5. A local web server is used for bootstrapper right now where a user registers her devices. After login, the user is provided with the main HTML page where they may select any devices, access data, copy between devices, etc. These operations are implemented by Javascript to call the web services in the devices. The *Metadata Manager and Mapper (MMM)* manages the mapping between the vUPS files to their physical locations that are saved in a SQLite database.

The *vUPS Synchronizer* communicates with the devices through the RESTful architecture as mentioned earlier. The *Data Manager* gives an abstraction over the local file system

with the underlying physical storage in the local disk via Java filesystem APIs.

For mobile devices, we also build a vUPS app to access the data. There is also vUPS API for Android to access vUPS from other apps. The current prototype maintains minimal security over the RESTful architecture in HTTP. We use the IP address of the device as the device ID in the current prototype implementation.

In the current implementations, files are simply differentiated based on their name extensions. Extensions like .mpeg, .wmv, .dat, .mov are considered as the popular movie filename extensions. vUPS includes the .mp3 and .mid as the popular audio files, where the .jpeg, .jpg, .gif, .bmp, and .png are considered as filename extensions of image files. Files with these name extensions are all considered as non-editable files. By default, all other files are considered to go under frequent modifications, and vUPS have different polices for them.

6.5 Evaluation

In this section, we evaluate the performance of vUPS based on the prototype we have implemented. In the experiments, we first use the micro-benchmarks to evaluate vUPS. Then we compare the performance of vUPS with the popular application Dropbox. In these experiments, we configure vUPS with three commodity desktops, one laptop, and one smartphone. To compare with the service provided by DropBox, we have also set up a DropBox account. All the desktop machines have 8 GB memory and 2.7 GHz CPU. The laptop has 4 GB memory and 2.7 GHz CPU. The desktop machines run Windows 7, Ubuntu 11 and Mac operating systems, respectively. We use a Google Nexus One phone with 512 MB memory and 1 GHz CPU running Android 2.3 operating system. The DropBox account has 2 GB of storage.

6.5.1 File I/O Performance

First, we study the performance of file reading. In vUPS, a desktop is designated as the main terminal. Files of 1 KB, 10 KB, 50 KB, 100 KB, 200 KB, 500 KB, 1 MB, and 3

MB are read by the smartphone via the main terminal. To emulate the realistic scenario, files are randomly distributed on these devices. When a read request for a file is sent from the smartphone to the vUPS, the main terminal receives the request from the smartphone. The main terminal searches the namespace for the ID of the device that stores the file, and forwards the web address of the resource to the smartphone. The smartphone then completes the read operation by bypassing the main termial. The results are compared against when a direct/local read from the sdcard of the smartphone and when the file of the same size is located on DropBox. We take the average of 100 runs for each file size and the results are shown in Figure 6.4. The x-axis represents the file size. The y-axis shows the response time. We set the network speed as 500 Kbps for local accesses. The maximum and the minimum response time are within 2% of the average with 95% confidence level. As expected, the read time increases with the increase of the file size and neither vUPS nor DropBox is comparable to the local read performance. However, vUPS constantly outperforms DropBox, for which the network speed is unconstrained with an average of 3 Mbps, although the advantages tend to diminish along the increase of the file size.

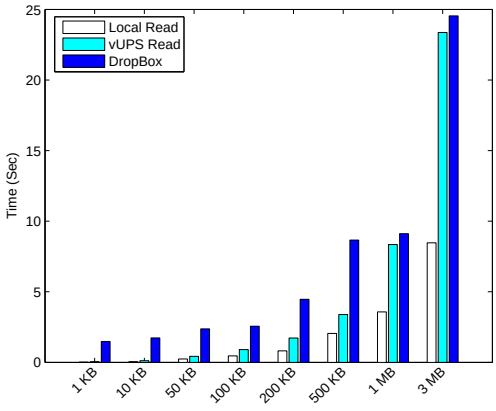


Figure 6.4: File Size vs. Response Time

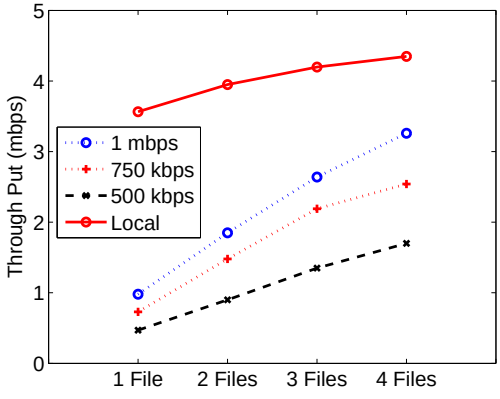


Figure 6.5: Parallel Threads for File Read

To bridge the performance gap for the sequential read in vUPS read, in the next experiment we use parallel threads for multiple file fetching. Figure 6.5 shows the result when the number of parallel file fetching grows from one to four. The x-axis represents the number of files fetched in parallel and the y-axis represents the throughput of the system. For the counterpart of the local read, four files are read by four parallel threads from the sdcard of the smartphone.

To read files in parallel from vUPS, the smartphone contacts the main terminal and requests the files. Each file is stored on a random machine selected arbitrarily. The main terminal returns to the smartphone with the web address of the file resources and the smartphone then fetches those files in parallel. Figure 6.5 shows that the throughput increases with the degree of parallelism. The throughput also increases with the network bandwidth for vUPS when the network bandwidth increases from 500 Kbps to 1 Mbps. When the available network bandwidth is sufficiently large, the gap between the local read and the vUPS read can be significantly reduced.

We have mentioned before that the file operations are handled directly between the requesting machine and the target machine without involving the main terminal in order to relieve the communication bottleneck. To study the performance gain, we compare it to the case when the main terminal fetches the file and then serves it to the smartphone. Figure 6.6 shows the results averaged over 100 tests. The results confirm that it is beneficial for the user and the system to bypass the main terminal whenever necessary.

6.5.2 Performance of Metadata Accesses

Metadata is crucial to vUPS to function properly. To measure metadata access performance, we conduct experiments with the *mdtest* benchmark on vUPS. *mdtest* is developed for large scale file systems with script and C, which is not suitable for our RESTful applications. To fit our system, we modify the *mdtest* and implement the simplified version in Java with threading. We replace the file reading/write/edit with Java system calls. In the experiments, we measure the throughput (the number of operations executed per second)

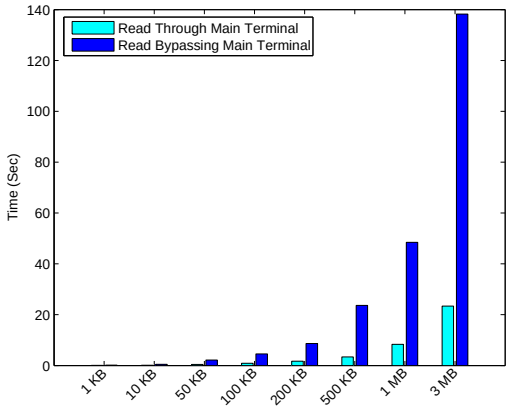


Figure 6.6: Performance Gain by Bypassing the Main Terminal

for creating 1000 files per terminal from the smartphone.

Figure 6.9 shows the throughput for file creation. In the experiments, we compare the time for creating files in our smartphone locally and in vUPS. For local file creation in the smartphone, we create 1000, 2000, 3000, and 4000 files using 1, 2, 3, and 4 threads (a thousand files per thread) in the sdcard of the smartphone. To measure the time of file creation in vUPS, we again create 1000, 2000, 3000, and 4000 files using 1, 2, 3, and 4 threads (a thousand files per thread). Each thread selects one of the devices in the vUPS system. Each thread first contacts the main terminal and requests to create 1000 files in the device.

The result shows that with an increasing number of local file creation requests, the throughput decreases. This is expected because the smartphone thread overhead and the limited file I/O slow down the local file operation, and thus reduce the throughput. On the other hand, for remote devices the performance improves, which is due to the parallel file creation feature in vUPS.

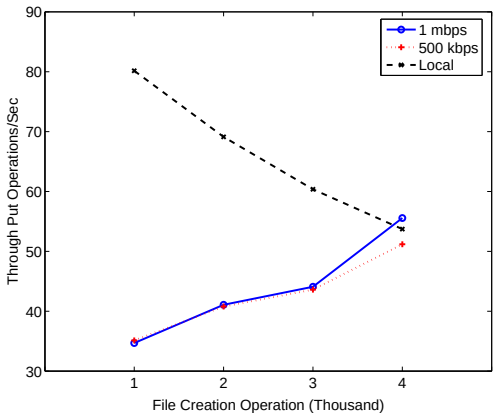


Figure 6.7: Throughput of File Creation

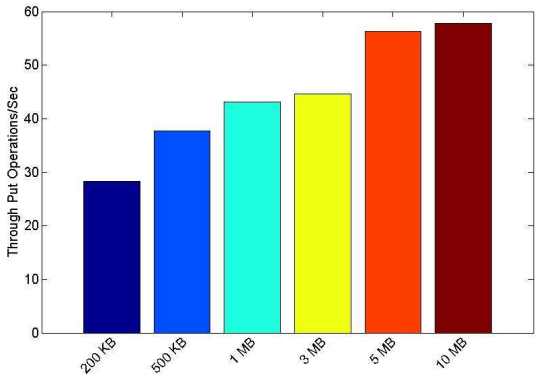


Figure 6.8: Network Bandwidth vs File Creation

6.5.3 Network Impact

To study the impact of network bandwidth, we have also run the *mdtest* file creation of 1000 files with varying network speed. Figure 6.8 shows the response time for 1000 file creation via the smartphone using one thread. The x-axis represents the network speed, while the y-axis represents the system throughput. The figure shows that the network bandwidth does impact the file creation throughput, but the improvement diminishes when the network speed is fast enough. For example, when the network bandwidth is doubled from 5 Mbps to 10 Mbps, the throughput for file creation operation only increases slightly.

To study the impact of network speed on file read operations, we vary the network speed and observe the response time to read a 3 MB file in the smartphone fetched from the main terminal. As expected, Figure 6.9 shows that the read throughput increases with the increase of bandwidth for vUPS while it remains stable for local read and DropBox. Note that because Dropbox is accessed through the public network, we did not constrain the bandwidth of the connection between the smartphone and the Dropbox.

Table 6.1 shows the response time for 5 trials with different network speed and file read or creation operations. From ANOVA [89] tests, we may conclude to reject the hypothesis that the operation type and network speed have no significance on the model. We can also

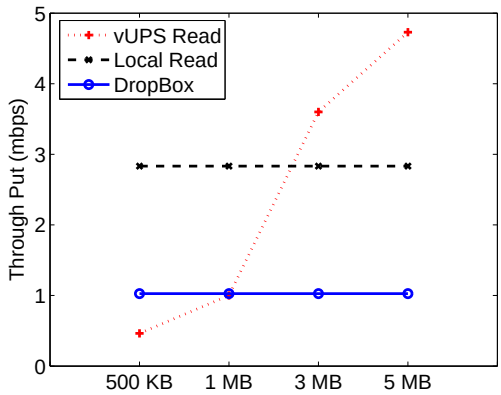


Figure 6.9: Network Bandwidth vs File Read

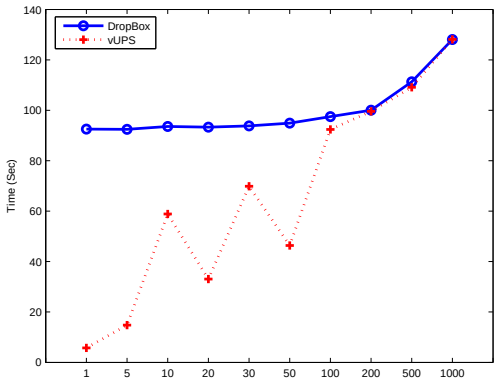


Figure 6.10: DropBox vs vUPS

conclude that the interaction between the network speed and the file operation type does exist.

6.5.4 Comparison to Dropbox

To compare with the DropBox in a more realistic scenario, we have generated 50 files with lambda distribution [90] as discussed in [91] with the size ranging from 1 KB to 512 MB according to that distribution. For DropBox, we synchronize all the files first, then access locally. In vUPS, in case of a cache miss, it downloads the file, otherwise checks the cache and retrieves it unless it is stale. We access these files in random order for 10 to 1000 times.

Figure 6.10 shows the result. The x-axis represents the number of files fetched randomly from these 50 files. The y-axis represents the response time. In the experiment, the DropBox first downloads all the files, and then accesses them randomly. On the other hand, vUPS does not download all the files at first. It downloads the files when being accessed and then caches them for future references. In this way, when the number of accessed file is small, the access time for DropBox is much longer than vUPS as the DropBox first needs to synchronize and download all the files. But when the number of accessed files is larger, the performance of both vUPS and DropBox is similar to each other.

Table 6.1: ANOVA Test for Network and Type of Operation

Operation	500 Kbps	1 Mbps
Read	186	120
Read	204	92
Read	222	164
Read	221	101
Read	228	180
Create	55	58
Create	17	40
Create	37	27
Create	43	32
Create	26	29

In addition to that, the strong consistency between the replicas introduces synchronization overhead. For example, even a single byte update in dropbox generates 37 KB of network traffic measured by wireshark, where vUPS generates only 1KB of data for the same synchronization.

6.6 Summary

The pervasive adoption of mobile devices calls for an effective approach to access users personal data across different devices from anywhere and anytime. While commercial products have offered compelling solutions, users are risking their data security and privacy. In addition, potentially, lots of unnecessary traffic has been wasted during continuous file synchronization. In this work, we have designed and implemented vUPS, a system to transparently and seamlessly integrate a users personal storage space. A web interface is provided to the user with a global view of the files without involving any third party. Experiments based on the implemented prototype system show that vUPS can achieve similar user performance when compared to commodity commercial solutions such as DropBox.

Chapter 7: Conclusion

The technology advancements have made mobile devices being adopted pervasively. To some extent, mobile devices are replacing their original counterparts in our daily life. Such an emerging trend has also led to fast development of various mobile applications. For resource-intensive mobile applications, there is an increasing demand to augment both computation and storage to more powerful counterpart, such as clouds and servers. In this dissertation, we have designed and implemented schemes for both computation offloading and storage augmentation.

For transparent computation offloading, we have addressed the problems of how to offload and what to offload. To tackle the first problem, we have designed and implemented a framework for smart and transparent mobile application offloading. We have implemented a transparent offloading mechanism at the method level. Compared to existing schemes, our solution does not require any modification to the source code or binary, or special compilation, so number of existing applications can be directly offloaded without further efforts.

To deal with the second problem of what to offload, we have designed and implemented a profiler of mobile applications to efficiently partition the applications in an optimal way to find the appropriate methods for offloading. We have designed and implemented a learning based model for estimating the on-device and on-server execution time of applications and make offloading decision accordingly. Compared to the most existing decision-makers utilizing a static policy, our decision maker based on MLP can capture the dynamic nature of the relevant key factors affecting the offloading decision and their relationship.

To augment the storage of mobile devices and enable pervasive access, we have designed and implemented a system to transparently and seamlessly integrate a users personal storage space with cloud with better availability and consistency.

Extensive experiments have been conducted to evaluate the computation offloading and storage augmentation schemes. The experimental results show that our proposed solutions can work seamlessly with existing mobile applications and enhance their performance significantly in terms of response time and energy savings.

Bibliography

Bibliography

[1] "International Data Corporation : Press Release 27 January, 2014," http://www.idc.com/getdoc.jsp?containerId=prUS24645514.

[2] "International Data Corporation : Press Release 30 April, 2014," http://www.idc.com/getdoc.jsp?containerId=prUS24823414.

[3] "Mint," http://www.mint.com/.

[4] "AWS SDK for Android," http://aws.amazon.com/sdkforandroid/.

[5] J. Eriksson, L. Girod, B. Hull, R. Newton, S. Madden, and H. Balakrishnan, "The pothole patrol: Using a mobile sensor network for road surface monitoring," in *Proceedings of The 6th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Breckenridge, Colorado, June 2008.

[6] "Diamedic. Diabetes Glucose Monitoring Logbook," http://ziyang.eecs.umich.edu/projects/powertutor/index.html.

[7] "iPhone Heart Monitor Tracks Your Heartbeat Unless You Are Dead," gizmodo.com/5056167/.

[8] B. Liu, P. Terlecky, A. Bar-Noy, R. Govindan, and M. J. Neely, "Optimizing information credibility in social swarming applications," in *Proceedings of IEEE InfoCom, 2011 mini-conference*, Shanghai, China, April 2011.

[9] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments," in *Proceedings of The 6th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Breckenridge, Colorado, June 2008.

[10] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proc. of EuroSys*, 2011, pp. 301–314.

[11] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," in *IEEE Pervasive Computing*, vol. 8(4), October 2009.

[12] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *Proc. of MobiSys*, San Francisco, CA, USA, June 2010.

[13] M. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proc. of Mobisys*. ACM, 2011, pp. 43–56.

[14] Y. W. Kwon and E. Tilevich, "Power-efficient and fault-tolerant distributed mobile execution," in *Proc. of ICDCS*, 2012.

[15] "Amazon Silk," http://amazonsilk.wordpress.com/.

[16] "Drop Box," http://www.dropbox.com/.

[17] "Sky Drive," http://explore.live.com/skydrive.

[18] "Google Docs," www.docs.google.com.

[19] M. L. Mazurek, E. Thereska, D. Gunawardena, R. Harper, , and J. Scott, "Zzfs: A hybrid device and cloud file system for spontaneous users," in *FAST*, 2012.

[20] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and F. Kaashoek, "Eyo: Device-transparent personal storage," in *Usenix Annual Technical Comference*, 2011.

[21] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services." in *NSDI*, 2010, pp. 17–32.

[22] "DropBox Security Breach," http://www.zdnet.com/dropbox-gets-hacked-again-7000001928/.

[23] "Facebook Security Breach," http://www.telegraph.co.uk/technology/picture-galleries/8939746/Private-photographs-of-Facebook-founder-Mark-Zuckerberg-leaked-after-glitch.html .

[24] 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*. IEEE, 2012, pp. 945–953.

[25] "Aliyun," http://os.aliyun.com/.

[26] I. Giurgiu, O. Riva, and G. Alonso, "Dynamic software deployment from clouds to mobile devices," in *Middleware 2012*. Springer, 2012, pp. 394–414.

[27] "Amazon S3," http://aws.amazon.com/s3/.

[28] "Mark Zuckerbergs pictures leaked," http://www.nydailynews.com/news/national/oops-mark-zuckerberg-pictures-leaked-facebook-security-flaw-article-1.988026?localLinksEnabled=false.

[29] "DropBox security breach," http://www.newsy.com/videos/dropbox-security-glitch-leaves-users-accounts-unprotected/.

[30] "Amazon EC2 outage," http://www.informationweek.com/news/cloud-computing/infrastructure/229402054.

[31] "SME Storage," ehttp://smestorage.com/.

[32] J. Strauss, J. M. Paluska, B. Ford, C. Lesniewski-Laas, R. Morris, and F. Kaashoek, "Eyo: Device-transparent personal storage," in *USENIX Technical Conference*, 2011.

[33] "Daily Main, April 1, 2012," http://www.dailymail.co.uk/sciencetech/article-2078020/Death-point-shoot-Smartphone-cameras-27-cent-photos.html.

[34] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, Dec 2004.

[35] S. Ghemawat, H. Gobioff, , and S.-T. Leung, "The google file system," in *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)*, NY, USA, 2003.

[36] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the sun network filesystem," in *Proceedings of the Summer USENIX conference*, 1985, pp. 119–130.

[37] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5. ACM, 1995, pp. 172–182.

[38] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb, "Simplifying cyber foraging for mobile devices," in *Proc. of Mobisys*, San Juan, Puerto Rico, June 2007.

[39] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case of cyber foraging," in *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, July 2002.

[40] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, and V. H. Tuulos, "Misco: a mapreduce framework for mobile systems," in *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*. ACM, 2010, p. 32.

[41] M. A. Hassan and S. Chen, "Mobile mapreduce: Minimizing response time of computing intensive mobile applications," in *Mobile Computing, Applications, and Services*. Springer, 2012, pp. 41–59.

[42] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: code offload by migrating execution transparently," in *OSDI*, 2012.

[43] "Linpack," https://play.google.com/store/apps/details?id=com.greenecomputing.linpack&hl=en.

[44] "ZXing," http://code.google.com/p/zxing/.

[45] "Mezzofanti," http://code.google.com/p/mezzofanti/.

[46] J. Kleinberg and É. Tardos, *Algorithm design*. Pearson Education India, 2006.

[47] J. H. HOWARD, M. L. KAZAR, S. G. MENEES, D. A. NICHOLS, M. SATYANARAYANAN, R. N. SIDEBOTHAM, and M. J. WEST, "Scale and performance in a distributed file system," in *ACM Transactions on Computer Systems*, 1988.

[48] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992.

[49] D. Borthakur, "Hdfs architecture : Technical report," in *Apache Software Foundation*, 2008.

[50] T. White, "Hadoop: The definitive guide (second edition)."

[51] M. Eshel, R. Haskin, D. H. Manoj, N. Frank, Schmuck, and R. Tewari, "Panache: A parallel file system cache for global file access," in *8th USENIX Conference on File and Storage Technologies*, 2010.

[52] E. B. Nightingale and J. Flinn, "Energy-efficiency and storage flexibility in the blue file system," in *6th conference on Symposium on Opearting Systems Design and Implementation*, 2004.

[53] D. Peek and J. Flinn, "Ensemblue integrating distributed storage and consumer electronics," in *7th conference on Symposium on Opearting Systems Design and Implementation*, 2006.

[54] S. Sobti, N. Garg, C. Zhangv, X. Yu, A. Krishnamurthy, and R. Y. Wang, "Personalraid: Mobile storage for distributed and disconnected computers," in *FAST*, 2012.

[55] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall1, and A. Vahdat, "Cimbiosys: A platform for content-based partial replication," in *Network Systems Design and Implementation*, 2009.

[56] "Google Drive," https://drive.google.com/.

[57] "Droidslator," http://code.google.com/p/droidslator/.

[58] "CyanogenMod," http://wiki.cyanogenmod.org/w/Build_for_passion.

[59] "Dalvik VM," http://source.android.com/devices/tech/dalvik/.

[60] "Virtual Box," https://www.virtualbox.org/.

[61] "MatCal," https://github.com/kc1212/matcalc.

[62] "Picaso," http://code.google.com/p/picaso-eigenfaces/.

[63] "MathDroid," https://play.google.com/store/apps/details?id=org.jessies.mathdroid&hl=en.

[64] "Power Tutor," www.powertutor.org/.

[65] J. Dean and S. Ghemaawat, "Mapreduce a flexible data processing tool," in *Communication of the ACM*, Jan 2010.

[66] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing.* Benjamin/Cummings Redwood City, 1994, vol. 110.

[67] J. org Ott and D. Kutscher, "Drive-thru internet: IEEE 802.11b for Automobile Users," in *Proceedings of IEEE InfoCom*, Hong Kong, March 2004.

[68] M. A. Hassan and S. Chen, "An investigation of different computing sources for mobile application outsourcing on the road," in *Proc. of Mobilware*, June 2011.

[69] "Weka," http://www.cs.waikato.ac.nz/ml/weka/.

[70] "tc," http://unixhelp.ed.ac.uk/CGI/man-cgi?tc+8.

[71] A. J. Nicholson and B. D. Noble, "Breadcrumbs: Forecasting mobile connectivity," in *Proc. of Mobicom*, 2008.

[72] "CPUUsage," http://stackoverflow.com/questions/3118234/how-to-get-memory-usage-and-cpu-usage-in-android.

[73] M. A. Hassan, K. Bhattarai, Q. Wei, and S. Chen, "Pomac: Properly offloading mobile applications to clouds," June 2014.

[74] W. Zhang, Y. Wen, and D. O. Wu, "Energy-efficient scheduling policy for collaborative execution in mobile cloud computing," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 190–194.

[75] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[76] R. T. Fielding, "Architectural styles and the design of network-based software architectures," in *PhD Thesis, University of California, Irvine.*, 2000.

[77] H. Kung and J. T. Robinson, "On optimistic methods for concurrency control," in *ACM Transaction on Database Systems*, March 1981.

[78] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," in *IEEE TRANSACTIONS ON COMPUTERS*, April 1990.

[79] R. G. Guy, J. S. Heidemann, W. Mak, J. Thomas W. Page, G. J. Popek, and D. Rothmeier, "Implementation of the ficus replicated file system," in *USENIX*, March 1990.

[80] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *OSDI*, 2002.

[81] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demer, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *fifteenth ACM symposium on Operating systems principles*, March 1981.

[82] J. R. Douceur and W. J. Bolosky, "A large-scale study of file-system contents," in *Sigmetrics*, May 1999.

[83] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," in *FAST*, 2007.

[84] J. R. Douceur and W. J. Bolosky, "A large-scale study of file-system contents," in *SIGMETRICS*, 2002.

[85] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *PODC*, 1987.

[86] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Communicatio of ACM*, 1978.

[87] A. S. Tanenbaum, "Distributed systems : Principles and paradigms," vol. Second Edition.

[88] "EC2 Failure Rate," http://aws.amazon.com/ebs/.

[89] R. Jain, "The art of computer systems performance analysis," 1991.

[90] J. S. Ramberg and B. W. Schmeiser, "An approximate method for generating symmetric random variables," in *Communications of ACM*, 1974.

[91] K. M. Evans and G. H. Kuenning, "A study of irregularities in file-size distributions," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, San Diego, CA, 2002.

Curriculum Vitae

Mohammed A Hassan received his Bachelor of Science from Bangladesh University of Engineering and Technology in 2006. He was employed as a lecturer in United International University, Dhaka, Bangladesh for two years. His research interests include networking, operating system, and mobile-cloud computing.