Mobile Stand-alone Application Code Off-loading: Architecture and Challenges

Abhishek Dwivedi
NMIMS, Mumbai

Padmaja Joshi
CDAC, Mumbai

Abhay Kolhe
NMIMS, Mumbai

ABSTRACT
Dynamically migrating code for execution from a device to a remote server has been a topic of active research for nearly a few years now. With the advent and the rapid growth of mobile devices, this area has been extended to mobile operating systems. Although mobile devices such as phones and tablets are available with increasing the hardware specifications to include more RAM and CPU cores, they do not often keep up with the demands of ever growing mobile applications. In the context of mobile devices, this field has come to be known as Mobile Cloud Computing (MCC). It combines mobile computing and cloud computing to achieve dynamic code migration of applications to a remote server, achieve reduced power usage and faster execution. In this paper, a review of the most prevalent techniques of MCC that are shaping this field is done. The paper also covers the complete architecture that is used to achieve these results. Along with various architectures, approaches a study of their advantages and disadvantages, as well as the way forward is discussed.

General Terms
Mobile Cloud Computing, code migration, code off-loading.

Keywords
MCC, code offloading, code migration, class-level offload, method-level offload, thread-level offload, code augmentation, Mobile Cloud Computing.

1. INTRODUCTION
The technologies of cloud computing and mobile computing have in the past few years, witnessed tremendous growth independently of each other. Since the past few years, mobile application ecosystems such as Android, iOS, and Windows 8 have become popular for hosting many types of mobile applications [1]. This has led to an entire industry specializing in commercial applications that run on mobile devices. Such applications are providing ever richer functionality on mobile devices.

Cloud computing can be defined as the aggregation of computing as a utility and software as a service where applications are delivered as services over the internet and the hardware and systems software in data centers provide those services [1]. Cloud computing enables an application to utilize computing resources that are remotely located, that is termed as a cloud. These resources have better computing power, memory than single standalone devices.

On the other side, mobility has become an important characteristic of the computing environment. However, mobile applications that require to process real-time data from sensors such as GPS or cameras, or video games and image processing applications are computationally intensive. Such applications demand significant battery consumption as well as a good computing power, and thus restricting the developers in implementing them for mobile devices [1]. It is in this regard that Mobile Cloud Computing (MCC) has been developed as a new paradigm. In recent years, MCC has been harnessed to address these problems. Thus, portions of the mobile application can be offloaded to the cloud to exploit the larger resources of the cloud. This frees the mobile device of at least that part of the computation burden, thus reducing its battery consumption. In this paper, some of the recent techniques used for offloading the code are discussed with the architecture that is followed to achieve this goal.

The paper is organized as follows. The next section provides the background of the mobile cloud computing concept. Section 3 discusses the basic architecture that is common to all mobile code offloading techniques. Section 4 discusses the major approaches to achieve mobile code offloading, their issues, and advantages and disadvantages of each. Section 5 discusses the impact of offloading especially in terms of the power consumption of the mobile device and the effect of network latency. Section 6 concludes the paper with identification of the work that needs to be done in this area.

2. BACKGROUND
The migration techniques discussed in this paper primarily focus on either reducing battery usage, or reducing the CPU cycles of the mobile device [2] [3] [4] for executing them. The applications chosen are not client-server ones but are mainly standalone. The standalone applications are divided in such a way that a portion of it will run on mobile device and the remaining portion may be executed on the remote server to improve battery usage and its performance. For this the portion that can be executed on the remote server needs to be migrated to it. This migration is also affected by the network strength between the mobile device and the remote server. The stronger the network strength, the greater are the chances of a portion of the app's code to be dynamically offloaded to the server or cloud.

The decision to offload the code is also taken by measuring certain metrics about the performance outcomes of executing the code locally, vis-à-vis executing it remotely. Most techniques ensure that anchored classes, i.e. classes which use the mobile device's hardware such as camera, sensors, GPS etc. are not offloaded. There are also provisions to allow code migration via downward task migration [3]. This allows for offloaded classes to exploit the functioning of the mobile device's hardware such as its cameras or sensors.

It must be emphasized that MCC can involve 'pervasive computing', in that data and applications can freely move amongst mobile devices and remote servers [3]. This can result in code migration that can take place across other mobile devices or servers in a cloud. Alternately, MCC may

be implemented on the client-server "push-pull" model, where requests sent by the client are acted upon by a single dedicated server.

3. BASIC ARCHITECTURE

This section details the basic architecture followed while code offloading.

The application whose code is to be augmented is first subjected to a transformation by injecting code in such a way that the architecture is induced into the application. Needless to mention, the injection must also maintain the correctness of the original application from the user's point of view. As a first step of transformation, all classes that can be migrated to cloud or remote server for execution are identified and/ or marked. Typically these are the classes that do not communicate with the hardware of the mobile device, such as cameras, GPS sensor, gravity sensor or are GUI classes [4].

There are two possible approaches to this. In the first approach anchored classes (classes that use hardware of the mobile device and must execute on the mobile device only) are marked whereas in the second one remotable classes are marked. In the first approach, there is a possibility of incorrect behavior if some class is missed through the scrutiny and is not marked as anchored. The second approach may affect the performance of the application but not the behavior [5]. Hence, a majority of the approaches are seen to follow the second approach.

The entire application resides at the client, which is the mobile device in this case. At the server end, the replica of the client's VM is created, with only the remotable part, or the complete application. The code at the server end may be transferred from the client or upon a request from the client server may download it from the play store or a particular location. In the former approach large bandwidth is required and hence, usually not preferred.

At run time, offloading can be done at the class level, the method level, or at the thread level.

3.1 What to Offload?

It is typically observed that offloading improves the performance of the application running on the mobile device as the server or cloud architecture on which the code is offloaded has better computing power as compared to the mobile device. There is a direct correlation between execution speed and Round Trip Time (RTT), as well as between execution time and the latency of the network connection. The higher the RTT, the greater is the latency or waiting period of the application, which in turn increases the runtime, as well as the power consumption of the application.

The offloading should be performed in such a manner that the correctness of the application's execution is maintained. The right flow of execution shall ensure that even after offloading portions of the application to a remote server, the execution is as expected.

One approach is to offload all execution to the server that does not need device-dependent hardware resources (such as camera, motion sensors) because it is assumed that the server has greater computational resources than any mobile device. However, the conditions to offload dynamically may not always be favorable as we shall see later. The favorability to offload may also depend on the section of code to be offloaded. For instance, in some approaches if the section of code utilizes native APIs that are dependent on the underlying OS / hardware, then such code cannot be offloaded. Similarly, most approaches do not offload those portions of the code that use device specific hardware like GPS, camera, motion sensors etc. The execution of such code is left to the device itself.

3.2 Factors Outside the Code

The main factors outside the code that influence the decision to offload the execution of a particular class are the network strength, location of the remote server (as it affects RTT) and the power being consumed by a section of the application. Depending on these, the offloading mechanism may defer offloading to the server in two corresponding scenarios. The first scenario arises if the network strength is low (such as in a 2G or 3G network) or if RTT is above a threshold. In this case, the cost of power of sending data on a weak network, and listening for a response back from the server may altogether be greater than the cost of local execution. The second scenario may arise when the size of the transmitted data is too large. This results in the cost of offloading (in milli Joules) being higher than the cost of running the class on the mobile device [4].

3.3 Dependency Within the Code

Logically the remotable classes that need high computing should be migrated to the remote server or cloud. In addition to these, it may be necessary to offload even those classes that the offloading mechanism does not flag as being heavy on system resources. These classes are those that are very likely to be called by the offloaded classes. If they are not offloaded, the communication between the mobile device and the remote server may increase. This increases the waiting time, thereby increasing the power consumption spent in listening to responses, eventually reducing the speed of execution of the application. Code migration is also performed to offload those classes that are tightly coupled with the offloaded classes [4]. In this technique, a directed call graph is constructed where the vertices represent the classes, and the edges represent the direction of the invocation. All the classes that are grouped together in the call graph are considered to be migrated together.

3.4 Profiler

In dynamic analysis, the profiler service comes into picture. The purpose of the profiler is to compute the runtime costs of the various parts of an application. These costs are typically power consumption and CPU cycles. The profiler also keeps monitoring physical parameters such as power consumption, available bandwidth and network latency. If a class method is deemed to be computationally intensive at runtime, the profiler can mark it to be offloaded to the remote server. However, the final decision to offload rests with the offloading mechanism that shall weigh other factors like network strength and latency also, before offloading a particular method for execution.

3.5 Static Analysis and Clustering

The decision to offload depends on the cost of computation of particular methods within classes. This can be done by a combination of static analysis, and profiling the methods at runtime.

In static analysis, a call-graph of the application is constructed. The nodes consist of classes or particular methods within classes, whereas directed edges represent the invocations [2][4][5]. The callers or callees are typically methods of classes, or classes themselves [4]. Nodes may also

contain corresponding CPU cycles or power consumption figures (in milli Joules). Using this, an estimate of the power consumption, and an estimate of frequency of calls to a class (or a function within a class) is prepared.

Classes that are likely to invoke each other can also be clustered [5]. These are offloaded altogether or made to reside

The basic architecture of this model of mobile cloud computing techniques is shown in Fig 1. A mobile device

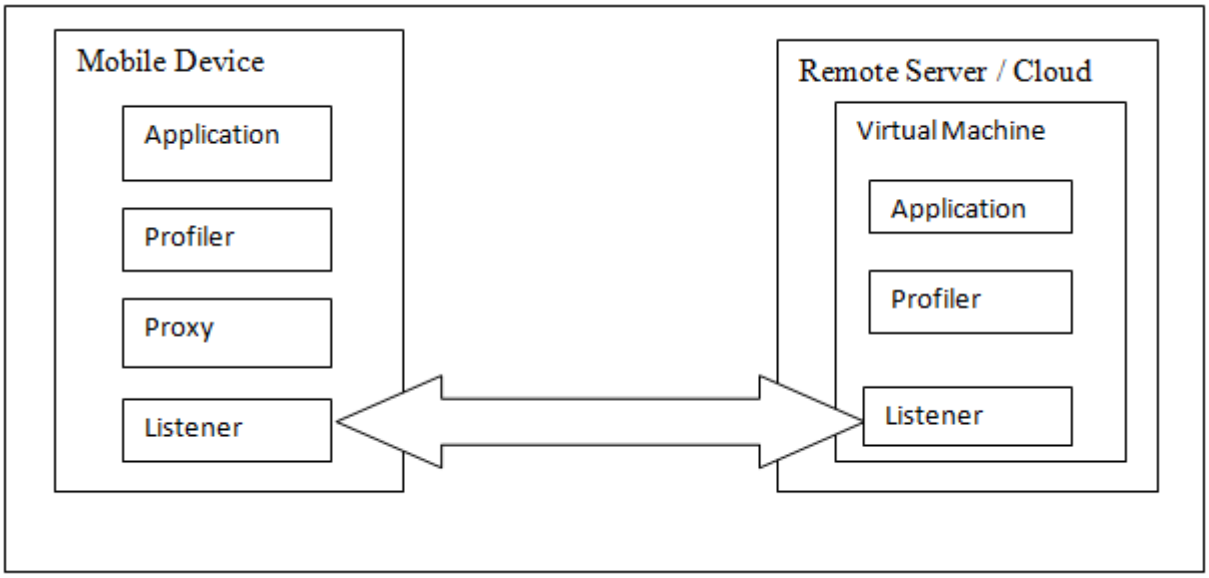

Fig 1: Basic model of code offloading in MCC

at the local device, in order to reduce the latency of communication. For this purpose also, a directed call-graph is constructed.

Another component that takes decision of offloading is profiler. The profiler takes input such as CPU cycles required, network bandwidth available etc. Based on these parameters, profilers provides info if the condition is suitable to offload or not. Using inputs from the static call-graph and the profiler, the system dynamically decides to migrate the instance of a class onto the remote server for execution. This procedure is also termed as dynamic profiling [2].

4. OFFLOADING APPROACHES

Based on the inputs of the profiler, the decision-making unit initiates the process of offloading the code. There are two main paradigms by which this is achieved.

4.1 Class Level/method Level Offload

In this paradigm, the most granular unit of offloading or migration is always the class. Any class (or particular methods within the class) that is deemed to be computationally heavy, is a candidate to be offloaded during runtime, provided certain favorable conditions are met. At runtime, the cost of execution of particular methods may be computed. This cost is typically in terms of power consumption. If the network is found to be strong enough, and if the cost of offloading is determined to be lesser than the cost of local execution, then that portion of the code is offloaded.

commencement of execution of the application at the client.

hosts an application, and also communicates with the remote server/ VM. The server/VM on cloud has either the entire application, or at least the remotable portion of the application hosted on the client as discussed previously.

The Listeners at both ends are used for communication between the mobile device and remote server / cloud. A socket connection (either TCP or a normal socket) is established between the two listeners prior to the

4.1.1 Requirement of proxy creation

As the execution of a class is decided dynamically a proxy class needs to be created for every remotable class. Proxy classes can be implemented using the relevant in-built libraries of Java, or refactoring tools such as Dexmaker[9]. Thus, for every instance of a remotable class, a corresponding proxy object is also generated. All invocations to the remotable instance happen via the proxy object of that instance. If a remotable instance resides locally on the device, the proxy transfers control of execution to the instance. Otherwise, the proxy transfers control to the remote server / cloud for execution. An RPC mechanism can be used for communication as it encapsulates the method invocation, its arguments and other details, and transmits the same to the remote server. After execution, the results are transmitted across to the mobile device.

4.1.2 Synchronization

It is imperative that the instances of remotable classes on the mobile device should be synchronized with the corresponding instances on the server. If any remotable method is invoked, then based on the inputs of the profiler the system may decide that its execution must be migrated to the server. In such a scenario the following sequence of events follow. The client first transmits the latest copy of an object to the server. This transmission makes sure that the copy at the server is now synchronized with the client's copy. Along with the synchronized object, the client also encapsulates the signature of the method being invoked and its argument list to be transmitted to the server. The updated values of the object are then used during the execution of methods at server/cloud end.

After execution, the server sends the modified object to the mobile device. This along with results if any, are transmitted back to the client mobile device. At the client end the results are returned to the appropriate caller for further execution. The transmission of the modified object from the server back to the client ensures that the client also has the latest copy of the object. Keeping the latest copy of an object ensures subsequent execution done locally is smooth in the event that the profiler decides that execution must now be done locally. As mentioned earlier, this situation may arise in case of a drastic drop in network bandwidth, high network latency etc.

4.1.3 Implementations with Class / Method Level Offload

MAUI [4] is one of the base techniques, that has been used as a benchmark for later work in the field. It migrates code at the method-level. However, it requires programmers of the mobile application to annotate the code with certain pre-defined keywords to let the offloader application determine at runtime, which portion can be offloaded to the remote server / cloud. A similar approach is also followed in [8], in which programmers are also required to design their application in terms of services requested by AIDL invocations. The basic premise of MAUI is to combine the approaches of fine-grained code offload, with minimal programmer intervention [4] and changes required to the applications. Microsoft's .Net CLR language has been used in development, though its techniques could equivalently be applied to the Android environment also. MAUI is able to create two versions of the application, one that runs on the mobile device and the other on the remote server. It also uses a profiler to determine the runtime behavior of classes (in terms of milli Joules or CPU time). Then, it factors the network connectivity in terms of strength and RTT to determine if a computationally intensive portion can be offloaded [4]. This computation is done by serializing the state of the offloaded method and determining the cost of transfer over the existing network.

The Profiler keeps monitoring the energy consumption or CPU cycles of the classes currently running. The optimization condition maximizes the energy that a method will consume upon executing locally and if it were to be transferred over the existing network. The constraints are the time to execute locally and the transfer time being less than a pre-defined threshold L [4].

DPartner [5] is one of the more recent implementations, built upon previously known techniques, including MAUI. Like MAUI, this technique also offloads only a portion of the code dynamically at runtime. Unlike MAUI it does not require the programmer to specifically annotate code to be marked for offloading. It automatically scans the class files and marks those classes as remotable, which use the sensor hardware of the mobile device [5].

4.2 Thread Level Offload

This paradigm is in contrast to the class level / method level offload. The main difference is that it operates at the further granularity of a thread. To achieve this, it requires access to and interactions with the heap or stack of the JVM running the application. Therefore, modifications by programmers to the code of the original application are no longer necessary. Instead, it either requires permanent modifications to the underlying JVM, or constructing a customized JVM that implements the offloading mechanism. One particular technique inserts a middleware that runs between the application and the underlying JVM [3].

4.2.1 Implementations with Thread level offload

CloneCloud adopts an approach that is very similar to that of MAUI. However, it offloads even fine-grained code at the method level instead of at the class level [2], by spawning threads that are migrated to the remote server/cloud. The primary goal in this technique is to obtain an improvement in the execution time of the application. The results of improvements in speed and energy consumption obtained by the authors follow the same patterns, as those observed in case of MAUI and DPartner. Before runtime, CloneCloud uses to static analysis to first designate partitions in a program's code that are suitable for offloading. These partitions are marked as execution points, one at the start and the other at the end of the partition. At runtime, when a starting execution point is encountered, execution is first suspended. The suspended thread would be, at that point of its execution be using states. These consist of stack states, heap states and register contents all belonging to the underlying JVM. These captured states are also marked in the phone's VM. The other threads on the phone continue processing.

The necessary states needed to execute the code within the partition (up to the ending execution point) are collected, and spawned into a thread. This thread is then transferred by an RPC mechanism to a remote cloud, where it is executed. The executed thread and its changed states are then transferred back to the mobile device, where it is merged with the original thread. In effect, the new states of the thread received from the cloud are updated in the device's corresponding thread. After merger, execution of the suspended thread resumes just after the end of the partition.

COMET [7] (Code Offloading by Migrating Execution Transparency) is a technique similar to CloneCloud, but by using Distributed Shared Memory (DSM). DSM enables it to offload fine-grained code to multiple nodes onto a cloud. This raises concerns about synchronization between multiple threads running on different cloud nodes. COMET manages synchronization by enabling the device to pull all the changed states of a particular method (or its partition) such as stacks, method-level registers and heap objects, from the cloud nodes. Then, it updates all the dirtied fields locally in a sequence determined by the "happens-before" relationship. In this way, the mobile device shall have the correctly updated states. Some of the advantages of this technique are that it can support multi-threading and migration of a thread at any finer level. Like CloneCloud, COMET too requires modifications in the JVM to achieve its purpose.

In CloneCloud [2] in order to decide which partition must be offloaded, dynamic profiling of the running application is performed i.e. the most heavily used partitions of code are analyzed. The network strength and latency at the point in time are also profiled to determine a cost model of the code to be offloaded [2]. If the cost to offload is lesser than the cost of local execution, then that partition is marked for offloading, . Once a partition is marked to be offloaded, and when the control reaches at the beginning of the partition (the first execution point), execution is suspended and the current states of the method are collected.

In systems such as CloneCloud and COMET, when a thread is received at the cloud, its states are recreated at the cloud, so that the thread is ready for execution. After the thread has

finished execution, the states of the thread are sent back to the mobile device. In CloneCloud, at the device end the thread contexts received from the cloud are overlaid over the suspended thread. The local heap states, registers and stacks are updated with those received from the cloud. COMET first uses the "happens-before" relationships to arrive at the final values of states, before doing this activity. The suspended thread is resumed again for execution. CloneCloud also maintains a table at both the device and cloud end. For every object in the heap, it contains an ID at the device, and same ID at the cloud. The table is synchronized at both ends, to facilitate garbage collection of objects, destroyed at the other end.

exCloud uses a Stack-On-Demand approach to offload code on the cloud [3]. It introduces a middle-ware system that is able to migrate tasks as coarse as VM instances, to as fine as run-time stack frames [3]. It also supports multiple cloud nodes so that parallel programs can be executed, thus providing what the authors argue is true cloud mobility instead of a "push-pull" mechanism in a client-server system [3]. The cloud relies on multiple cloud nodes to execute an application. exCloud however places a middleware between the application and the underlying JVM to achieve its goals.

A system somewhat similar to exCloud, called MobiCloud has also been developed. It extends the functionality of the server to treat mobile devices in an ad-hoc network as a cloud [6]. It is built on top of the MANET network, and facilitates routing, allocation of resources and security [6**]**. It also utilizes a concept known as concept-aware routing, according to which battery power, CPU power, bandwidth are factored to make routing decisions [6]. This helps in efficient migration of the code of a mobile app during runtime.

Determining which portion of the code to migrate is done dynamically by exCloud and COMET. COMET makes every attempt to transmit dirtied data to the cloud. exCloud's Stack-on-Demand approach migrates heap data based on its threshold of loading. The greater the loading, the more likely it is to be migrated to the cloud. Migration can also be performed if there are runtime exceptions. In CloneCloud, static analysis is performed to first scan the code of the entire application, to determine the partitions suitable for offload.

In all the techniques however, certain constraints determine the kind of code that can be offloaded. The code should not access native code i.e. device or platform dependent code (typically written in C). Also, it must not access device hardware such as cameras, GPS or other sensors. The latter constraint is also present in the techniques focusing on class /method level offload.

4.3 Metrics to Determine Offloading

Most techniques discussed in the previous sections, such as MAUI, DPartner, CloneCloud and MobiCloud use optimization techniques to dynamically determine whether a section of the code must be offloaded.

The cost to execute a particular code entity (like a class, a method or code within a method) is typically in terms of energy consumed (in milli-Joules), CPU cycles and memory. This cost is compared to that of the cost of migration, under favorable conditions. The most favorable condition is when network latency is low. If the cost to offload the code entity under current conditions is lesser than the cost were local execution to continue, offloading is performed.

5. IMPACT OF CODE OFFLOADING

Interestingly the Power Saver Mode (PSM) may not always result in optimum performance of a mobile device under dynamic code offloading. The reason is because if the latencies approach the sleep interval, the mobile device expends energy in waiting for a response besides causing the application to slow down [4].

Higher RTT between the device and the remote server increases the time that the device waits for responses from the server, which in turn increases the power consumption of the device. Thus, the performance of offloading systems within 3G networks is expectedly poorer than within Wi-Fi networks. Even within a network, increasing latencies due to RTT values can result in increased power consumption on the part of the mobile device. A similar general pattern of power consumption is followed by most offloading techniques discussed, both under Wi-Fi and 3G networks.

Another and equally important factor is the bandwidth of the network. A higher bandwidth network allows for larger data transfer whereas a lower bandwidth increases the latency of transfer, thus actually slowing down overall execution of the app.

The energy consumption (in milli Joules) is the highest when the entire application runs on the mobile device. As discussed earlier, even within a strong network such as Wi-Fi, increasing RTT directly impacts the energy consumption due to increased latency. Lastly, the energy consumption increases sharply in case the network is a 3G network, thus indicating the effect of reduced bandwidth influencing the system to defer offloading more often and continue execution on the mobile device.

6. CONCLUSION AND FUTURE WORK

This paper reviews the most recent and prevalent work done in the field of Mobile Cloud Computing. The paper details the two major approaches class or method level code offload and thread level code offload. The former can be implemented by modification to the source and may also require programmers to annotate certain methods, though one technique eliminates this. It is also platform and architecture independent. The latter approach requires permanent modifications to the underlying JVM, but has the disadvantage that the application may not be portable across platforms and devices. However, unlike the first approach, this one has finer granularity of code offload, in that sections of methods with only a subset of the method's states can be offloaded remotely for execution. The class-level/method-level approach on the other hand is coarser. Both class-level and thread-level offloading techniques can be extended for multi-threaded applications. But, the finer granularity of thread-level techniques may also come at the cost of native code and native API's being difficult or impossible to offload remotely. This problem is nearly non-existent in the class-level techniques of code migration. However, code that uses device hardware such as cameras and sensors is not offloaded in most techniques.

Static analysis techniques are employed to a much greater extent in class-level approach, to identify code that can or cannot be offloaded at runtime. Most thread-level offloading techniques rely on the demand or usage statistics of JVM states such as heap objects, stacks and registers.

The primary limitation of thread-level offload is that native level states cannot be offloaded, at least not in most of the cases. The class-level offloading techniques deal with coarse-level offloads, which is why they may impose greater penalties on the bandwidth. The way forward may be to combine the positives of both approaches: finer-grained offload may be required in the class/method level approach so as to reduce network overheads, thus improving speeds of execution. This would especially be useful under networks with low bandwidths and/or high RTT. On the other hand, thread-level offloading techniques have to find means of executing native code/APIs remotely. A library of well known APIs and which is defined at the cloud may also help. Protocols to transfer relevant address spaces with as little overheads as possible may have to be developed in such cases.

Finally, the future of MCC promises a lot of advancement and delivery of many more of its goals. The goal will be to bring all the capabilities of a desktop on a mobile device. It will not be surprising if a lot of research work is seen in this domain in the coming years.

7. ACKNOWLEDGMENTS

Our thanks are due to the team at C-DAC Mumbai and NMIMS University, with whose support this study was made possible.

8. REFERENCES

[1] Niroshinie Fernando, Seng W. Loke, Wenny Rahayu. 2013 Mobile Cloud Computing: A Survey. Future Generation Computer Systems 29.

[2] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, Ashwin Patti. 2011 CloneCloud: Elastic Execution between Mobile Device and Cloud. In Proceedings of Eurosys '11.

[3] Ricky K. K. Ma, King Tin Lam, Cho-Li Wang. 2011 ex-Cloud, Transparent Runtime Support for Scaling Mobile Applications in Cloud. In Proceedings of the International Conference on Cloud and Service Computing.

[4] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In Proceedings of MobiSys'10.

[5] Ying Zhang, Gang Huang, Xuanzhe Liu1, Wei Zhang, Hong Mei, Shunxiang Yang. 2012. Refactoring Android java Code for On-Demand Computation Offloading. In the Proceedings of OOPSLA'12.

[6] Dijiang Huang, Xinwen Xhang, Myong Kang, Jim Luo. 2010. MobiCloud: Building Secure Cloud Framework for Mobile Computing and Communication. In Proceedings of Fifth IEEE International Symposium on Service Oriented System Engineering.

[7] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, Xu Chen. 2012. COMET:Code Offload by Migrating Execution Transparently. In Proceedings of 10th USENIX Symposium on Operating Systems Design and Implementation.

[8] Dejan Kovachev, Ralf Klamma. Framework for Computation Offloading in Mobile Cloud Computing. International Journal of Artificial Intelligence and Interactive Multimedia, issue 7, volume 1.

[9] Dexmaker: Programmatic code generation for Android,http://code.google.com/p/dexmaker