A transparent code offloading technique for Android devices

Rômulo Reis de Oliveira, Nícolas Meira da Silva Schirmer, Mateus Machry and Tiago Coelho Ferreto
School of Computer Science
Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
Email: {romulo.reis, nicolas.schimer, mateus.machry}@acad.pucrs.br, tiago.ferreto@pucrs.br

***Abstract*—The integration of cloud computing and mobile devices, known as Mobile Cloud Computing (MCC), allows the adoption of code offloading techniques for improving compute intensive applications' performance and minimize the energy consumed by mobile devices. In this paper we propose a new technique that transparently includes code offloading capabilities to Android devices. It allows all the applications on the device to benefit from it without requiring the Android system recompilation or any modification in the application source code. We evaluated our proposal using a face detection application in different devices, in which the computation-intensive method is executed in a remote server in a local network. Results show gains up to 70% in terms of performance.**

***Keywords-component*—Mobile Cloud Computing; Code offloading; Cloud Computing; Android; Face detection**

I. Introduction

Mobility is a relevant characteristic of modern computing environments. Spreading faster than PCs and being able to support a large number of different applications, mobile devices such as phones and tablets are becoming the primary means for accessing the Internet. Despite being resource-poor, mobile devices are expected to offer the same performance and functionality as desktops. However, compute intensive applications that require powerful processors, abundant memory and long-lasting battery life such as, speech and image recognition, natural language processing and 3D games, are not likely to be suitable in this context. Therefore, mobile devices still cannot replace the traditional desktops, due to the limited CPU speed, memory, storage and battery capacity.

Since server processors are more powerful than mobile ones [13], Mobile Cloud Computing (MCC) has emerged as a way to overcome such constraints on mobile devices. It aims to provide unlimited resources on demand as a service in a transparent way to the user. In this context, code offloading [14], [21], [27], [11], [12] enables the mobile devices to dispatch local computation to more resource-rich computers, which can be the cloud infrastructure, a remote server or a cloudlet [24]. This feature can speed up the application execution and also minimize the use of the battery, memory and local storage. It can also give the user the impression of augmented local resources. Google Photos [15] and Apple iOS's Siri [8] are examples of applications that benefit from code offloading.

Many code offloading techniques have already been proposed [10], [17], [9], [16], [18]. However, even if some of them are user friendly, they are not developer friendly, since they require explicit changes in the applications' source code.

In this paper we propose a transparent code offloading technique for Android devices. Our proposal does not require any changes in the Android system firmware or applications source code. We use the Xposed Framework [6] to transparently modify Android Framework methods and send tasks to be executed remotely in a remote server or in the cloud. Since the modification is performed on the framework layer, no changes are required in the applications.

In order to evaluate our proposal, we implemented a prototype using the Android methods for face detection. We emulated a local cloud using a regular virtual machine running Android-x86, which is accessed by the mobile device through a wireless network. Results show that the proposed technique presents significant gains in application performance, and impacts transparently all applications that use face detection routines.

The remainder of this paper is organized as follows. In Section II we introduce the main technologies related to our paper and in Section III we summarize other works related to code offloading. Section IV outlines the design goals and architecture requirements and prerequisites of our technique. In Section V is explained our proof-of-concept architecture. Methodology, tools and the evaluation are discussed in Section VI. The results are presented in Section VII and we conclude the paper in Section VIII.

II. Background

Android [7] is more than just an operational system, actually, it is a software stack for mobile devices that includes an operating system, middleware, and a set of key applications and services. This software stack consists of distinct layers: Application, Application Framework, Android Runtime, Hardware Abstraction Layer, Native Libraries and Linux Kernel [19].

The most relevant layer in our proposal is the Android Framework, which provides a set of services that are used by applications. This framework implements the concept that Android applications are constructed from reusable, interchangeable and replaceable components. Android provides APIs in the form of Java classes, so application developers are allowed to make use of these services in their applications [25].

The Android Framework is encapsulated together with those layers in a ROM image. Building a custom Android ROM is a challenge, since each vendor usually builds its own private customized Android and the source code of these distributions are kept in secret with all the drivers. There are some Android open source projects that support some devices in which this customized ROM can be installed ("flashed") to the phone. However, they usually have several bugs, since those projects are not supported by the companies.

The open project Xposed Framework [6] is an alternative for Android customization and does not require any change in the ROM. By using root access, the Xposed Framework directly accesses the core Android resources and utilizes them to be able to run different modules on the device. Those modules enables "hooking" method calls, so the developer can override a method and even change the arguments passed by in runtime.

III. Related Work

There are several research works proposing different code offloading techniques for improving the application's performance and minimizing the energy usage by using resources in the cloud [10], [17], [9], [16], [18].

MAUI [10] provides method level code offloading based on the .NET framework and aims to save the devices' battery and improve the application performance by dynamic selecting which methods must be remotely executed. However, the developer must manually annotate the methods that are candidates for offloading as *remoteable*. By this way, a custom compiler generates two separates code bases, one for the mobile device and other the server. Three different applications were used to evaluate MAUI's performance, a face-recognition application, a highly-interactive game (chess game) and a real-time voice translator.

ThinkAir [17] also provides method level code offloading and provides a library and compiler support to make its adoption easier for developers, in addition to a VM manager and a parallel processing module in the cloud. It requires the developers to indicate methods that could be executed in the cloud, and it focuses on minimizing energy consumption of the mobile devices by paralleling method execution using multiple VM images and cloud scalability by dynamic allocating VMs on demand. An N-queens puzzle, face detection application and a virus scan were used to evaluate its performance.

CloneCloud [9] creates a virtual copy from a device, but more computationally powerful. It automatically selects what methods and when they should be remotely executed, optimizing the energy consumption and improving the application performance. It also migrates the code in a transparent way and the developer does not need to modify their application code.

COMET [16] allows multi-threaded applications to use multiples machines. The system allows those threads to migrate freely between machines. Built on top of the Dalvik Virtual Machine, it implements distributed shared memory (DSM) for minimizing the communication between machines. It used an image editor, a turn-based game, a trip planner and a math tool application for evaluation.

uCloud [18] explores the application's components that are independent from each other, but, developers must follow a standard procedure during the application development.

Our technique provides method level code offloading for native methods, not for a specific application method, differently from the others. Through this approach, all applications that use those native methods may benefit from it. Besides that, no modification in the application or customization in the compilation process is required, saving programmer effort.

Differently from [10], [17], [9], [16], [18], in our approach, the decision maker is the user, which can enable or disable the remote execution anytime giving more control to the user.

IV. Architecture and Design Goals

Figure 1 illustrates the proposed architecture for code offloading. The architecture is composed of a number of physical mobile devices as clients and a cloud server hosting VMs instances. There is an entity (proxy) in the Android device that modifies the behavior of a set of methods from the Android Framework. This entity detects when a candidate method is called and handle this call to be executed in the cloud. Each VM runs a service, which is responsible for receiving and replying to the requests from the mobile devices. When an application calls an expensive method from the Android Framework, the task is sent to be processed in the cloud or remote server (App B). Otherwise, the method is locally executed (App A).

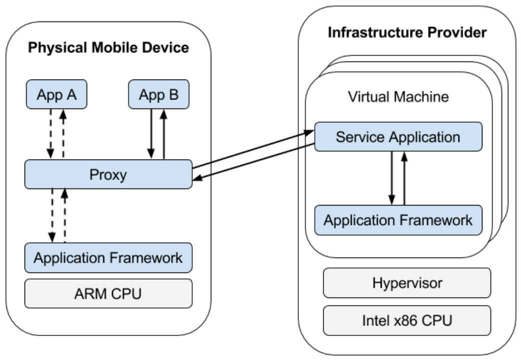

Fig. 1. Proposed architecture for code offloading

The key design objectives for our proposal are:

- It must be practical and easy for users and developers to use this technique.
- No modification in the Android system should be required, since each vendor has its own custom Android and this code is not public.
- All applications should benefit from our approach, if they use a method modified by the module.
- Applications should present a performance improvement when using this technique.

- The user is the decision maker, i.e., he has the control to choose when he wants to use code offloading. However, the technique must support the implementation of algorithms that decides what, when and where to offload the tasks.

V. Proof-of-concept Implementation

Our proof-of-concept prototype uses the open source projects Xposed Framework and Android-x86 [1].

We used Xposed Framework since it allows to install modules that can change method's behavior from the system or applications. No modification in the source code of the Android system or its applications is required, because the changes are done in the memory, which also makes easy to enable or disable a module.

The Android-x86 was used for two main reasons. First, Android was originally designed for ARM processors, however, the open source project Android-x86 has been initiated for porting the system to x86 platform. This port eliminates the overhead generated by the emulation, since it allows the Android system to be installed and run directly in the hardware of a traditional computer. This can boost application performance, because computer hardware is more powerful. Running Android directly in a computer hardware allows to visualize the system by using some virtualization tools, such as: VirtualBox [20], VMWare [26], and QEMU [5]. Second, since Android uses the same Java application framework/SDK in both processors, the application portability does not require any modification in the application.

The overall proof-of-concept implementation is depicted in Figure 2. Android VM runs an application service, which is responsible for receiving and replying the requests from the mobile devices, where the Xposed Framework is installed and our custom module is enabled. This module modifies the behavior of a predefined set of native methods from the Android Framework, making them send the request to the cloud. So, every time an application calls a native method that has been modified, this method sends a request to the cloud with the method name and parameters.

The service application, which is running in the VM instance (server), uses sockets to communicate with the devices. When a request is received, it calls the native method required with the parameters from the request, then the service replies to the device with the result.

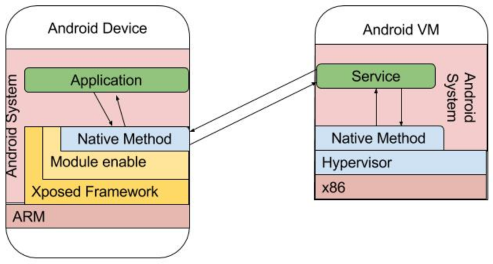

Fig. 2. Prototype implementation

VI. Methodology, Tools and Prototype Evaluation

For evaluating the performance of this architecture we implemented a prototype and ran a set of experiments to analyze the following metrics: execution time, network usage, and memory usage. Profiling Android is not easy, since there is not a unique tool that provides accurate information of CPU, memory, energy and network usage of an application. For profiling the network and memory we used Dalvik Debug Monitor Server (DDMS).

The prototype has three main components: (i) Face Detection application, (ii) Server Application, and (iii) Xposed Framework module. The Face Detection application uses Android framework methods to detect faces in an image. This application is executed in the smartphone and is implemented as a regular application (there are no annotations for offloading). The Server Application runs as a service application in an Android-x86 VM to handle offloading requests from mobile devices.The Xposed Framework module connects both components. It modifies the behavior of the method *detect* from class *vision.face.FaceDetector*. Instead of executing the code to detect faces in the picture on the device, it sends a request to the Server Application and waits for its response, in order to get back to the application with the appropriate result.

A class must implement the interface Serializable to enable its objects to be sent by the network. Since the classes Frame and SparseArray, used by face detection methods, do not implement this interface, we had to handle this by manually serializing those classes using the GSON library [4]. We chose GSON, because among the few open-source projects that can convert Java objects to JSON, most of them require that the developer place Java annotations in the classes, which is not possible in our context, since we do not have access to the source code. Most of them also do not fully support the use of Java Generics.

In the virtual machine, we used Android 6.0.1 [2] from the Android-x86 open source project without modification in the source code. We just created a virtual machine using VirtualBox and installed Android. Just the server application was installed in the Android VM. We used VirtualBox 5.1.4 for running the VM in the computer. We used two devices: Smartphone X and Smartphone G. The smartphones, computer and VM specifications are presented in Table I.

A wireless router ASUS model RT+-N10 (150Mbps High Speed) was used to enable the communication between the smartphone and the VM host.

The Xposed Framework installation requires root access, bootloader unlocked and TWRP installed. After the Xposed Framework installation, we installed our custom module and enabled it. As mentioned before, the module modifies the method *detect* from the class *vision.face.FaceDetector*. According to the Android official documentation [3], this method finds faces given an image, the result is a *SparseArray* of faces.

In our evaluation we used 2 representative images that vary in file size, dimension and in number of human faces

TABLE I
COMPONENTS SPECIFICATIONS

Name	Operational System	CPU	RAM	Storage
Computer	Ubuntu 16.04 64 bit	Intel Core i7-4790 3.6 GHz	16 GB	1 TB
VM	Android-x86 6.0.1	4 v. cores	6 GB	8 GB
Smartphone G	Android 5.1.1	4x1.4 GHz Cortex-A53	1 GB	8 GB
Smartphone X	Android 5.1.1	4x1.5 GHz Cortex-A53 4x2.0 GHz Cortex-A57	3 GB	32 GB

(Table II). The images are available online [22], [23] and are showed in Figures 3 and 4. We used both images on two different devices: Smartphone X and Smartphone G. Before running each experiment, the face detection application was killed and the device was rebooted.

We also explored two different scenarios. In the first one (Scenario 1) we assumed that the VM has no information about the smartphone data, so the file of the image needs to be sent to the VM to be processed. In the second scenario (Scenario 2), we assumed there is a local copy of the smartphone data needed by the VM for processing the method, so, the smartphone sends a request to the VM with an identification of the image and not the image file.

Fig. 3. Image A

For each image, we ran the face detection application three times in each smartphone. In the first time, we used the local computational resource of the device. Since the test was performed locally, no network traffic was generated. In the second and third time, we explored scenario 1 and 2 respectively. The execution of the method *detect* was performed on the remote Android VM hosted in a computer.

VII. Results

The first metric analyzed is execution time. The comparison of execution time for local and remote executions, in both scenarios, can be seen in Figure 5 and 6.

For both images in scenario 1, Smartphone X took longer to run the method remotely than to run it locally. In scenario 2, the execution time was 29,63% faster for image A and 31,88% faster for image B.

However, when running the method remotely using smartphone G, the execution time performed better in both scenarios and images. In scenario 1 it was 40,37% faster for image

Fig. 4. Image B

A and 37,10% faster for image B compared with the local execution time. In scenario 2, the execution time was 72,03% faster for image A and 70,02% for image B. Smartphone G can benefit more from code offloading because it has a less powerful hardware than smartphone X. Scenario 2 showed the best performance in this context.

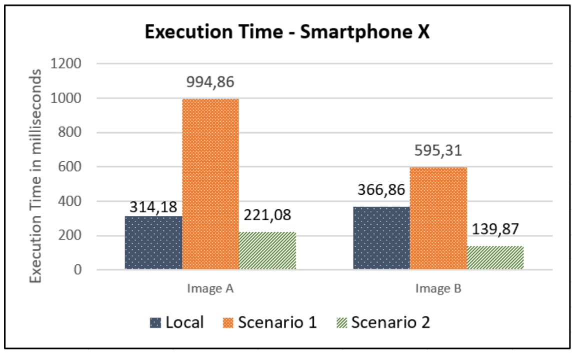

Fig. 5. Total Execution Time

We profiled the application to get the network usage. The result is summarized in Table III. The data transferred should be the same for both smartphones, however, in some cases, the number of faces can be different due to ACCURATE_MODE set in the FaceDetector in the service application. Due to the higher capacity of the VM, we enabled the ACCURATE_MODE flag to increase in accuracy when executing the method remotely. Smartphone X found 23 faces in image A running the method locally, while running the method remotely the application service found 24 faces in scenario 1 and 25

TABLE II
IMAGE SPECIFICATIONS

Name	Width x Height (px x px)	Size (KB)	Number of faces
Image A	640 x 450	660	29
Image B	640 x 502	73	5

TABLE III
NETWORK TRAFFIC

	Smartphone X				Smartphone G			
	Scenario 1		Scenario 2		Scenario 1		Scenario 2	
	Image A	Image B	Image A	Image B	Image A	Image B	Image A	Image B
RX (bytes)	5.412	1.543	4.729	1.127	4.812	1.165	4.729	1.127
TX (bytes)	20.580	10.892	584	376	9.534	4.616	584	376

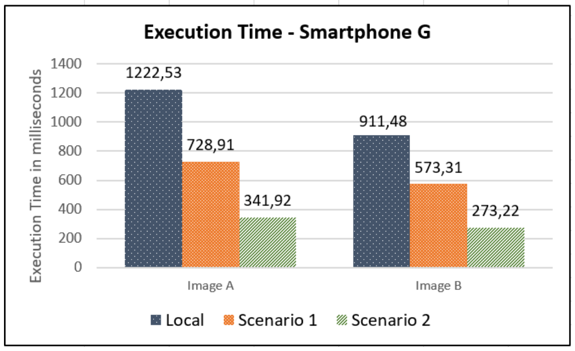

Fig. 6. Total Execution Time

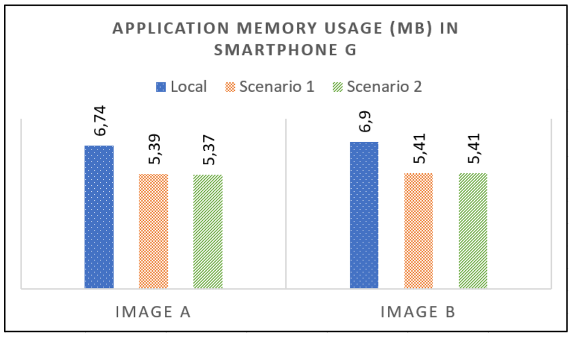

Fig. 8. Application Memory Usage - Smartphone G

in scenario 2. Smartphone G found 22 faces in image A by running the method locally, against 25 by running it remotely.

We also profiled the application to get the memory usage. Figures 7 and 8 represent the application memory allocated after the execution of the method *detect*. In all cases, less memory was allocated when executing the method remotely. Smartphone X allocated in average approximately 40% less memory for detecting the faces from image A and in average approximately 7,95% for image B. Smartphone G allocated in average approximately 20,18% less memory for detecting the faces from the image A and in 21,59% for the image B.

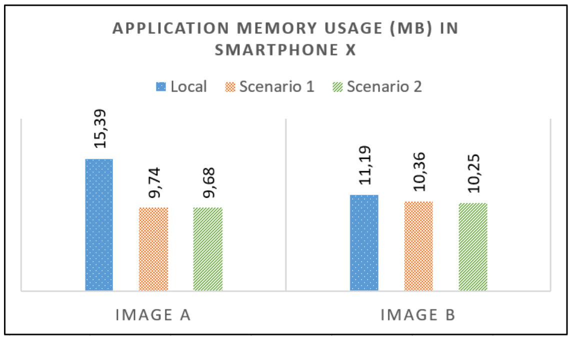

Fig. 7. Application Memory Usage - Smartphone X

VIII. CONCLUSION AND FUTURE WORK

In this paper we presented a new technique for code offloading, where no modification is required in the applications' source code nor changes in the Android firmware. This technique is transparent, practical and not intrusive for users and developers. It also allows the user to enable or disable the module anytime, but it is possible to implement a decision maker in the module what was not our focus. The resources from the cloud are more powerful and all the applications can benefit from our technique, if they use/call the modified methods by the module. Running a task remotely is not always faster than locally. For improving performance, the methods from the Android Framework must be computationally "expensive" and not require heavy data transference between the smartphone and the cloud.

Our results showed that this technique can improve execution time for some cases and also minimize memory allocation. We still intend to explore other methods from Android that can benefit from our technique and analyze other metrics, such as CPU and battery usage. Since we assumed an environment where the server is close to the device, we also intend to investigate the behavior of this technique using a cloud computing infrastructure far from the device in order to verify the impact of the communication delay. A mathematical model for indicating when the user could benefit from our technique is under research.

ACKNOWLEDGMENT

We gratefully acknowledge Motorola mobility for its support to this research

REFERENCES

[1] Android-x86. http://www.android-x86.org/.

[2] Android-x86 - porting android to x86: Releasenote 6.0-r1. http://www.android-x86.org/releases/releasenote-6-0-r1.

[3] Google apis for android: Facedetector. https://developers.google.com/android/reference/com/google/android/gms/vision/face/FaceDetector.

[4] Gson. https://github.com/google/gson.

[5] Qemu. http://wiki.qemu.org/Main_Page.

[6] Xposed framework. http://repo.xposed.info/.

[7] O. H. Alliance. Android open source project. https://source.android.com/.

[8] Apple. Siri. http://www.apple.com/ios/siri/.

[9] B. Chun, S. Ihm, P. Maniatis, and M. Naik. Clonecloud: Boosting mobile device applications through cloud clone execution. *CoRR*, abs/1009.3088, 2010.

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

[11] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[12] N. Fernando, S. W. Loke, and W. Rahayu. Mobile cloud computing: A survey. *Future generation computer systems*, 29(1):84–106, 2013.

[13] J. Flinn. Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 2012.

[14] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya. Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine*, 53(3):80–88, March 2015.

[15] Google. Google photos. https://photos.google.com/.

[16] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA, 2012. USENIX Association.

[17] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Unleashing the power of mobile cloud computing using thinkair. *CoRR*, abs/1105.3232, 2011.

[18] V. March, Y. Gu, E. Leonardi, G. Goh, M. Kirchberg, and B. S. Lee. cloud: Towards a new paradigm of rich mobile applications. *Procedia Computer Science*, 5(Complete):618–624, 2011.

[19] A. Misra and A. Dubey. *Android Security: Attacks and Defenses*. An Auerbach book. CRC Press, 2016.

[20] Oracle. Virtualbox. https://www.virtualbox.org/.

[21] M. Othman, S. A. Madani, S. U. Khan, et al. A survey of mobile cloud computing application models. *IEEE Communications Surveys & Tutorials*, 16(1):393–413, 2014.

[22] Pixabay. Image a. https://pixabay.com/en/smile-laugh-portrait-close-joy-1491429/.

[23] Pixabay. Image b. https://pixabay.com/en/family-kids-happy-people-mother-521551/.

[24] M. Satyanarayanan, R. Schuster, M. Ebling, G. Fettweis, H. Flinck, K. Joshi, and K. Sabnani. An open ecosystem for mobile-cloud convergence. *IEEE Communications Magazine*, 53(3):63–70, 2015.

[25] N. Smyth. *Android Studio 2.2 Development Essentials - Android 7 Edition:*. CreateSpace Independent Publishing Platform, 2016.

[26] I. VMWare. Vmware. http://www.vmware.com/.

[27] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):23–32, 2013.