






Improving the Trade-Off between Performance and Energy Saving in Mobile Devices through a Transparent Code Offloading Technique

Rômulo Reis¹^a, Paulo Souza¹^b, Wagner Marques¹^c, Tiago Ferreto¹^d and Fábio Diniz Rossi²^e

¹*Polytechnic School, Pontifical Catholic University of Rio Grande do Sul
Ipiranga Avenue, 6681 - Building 32, Porto Alegre, Brazil*

²*Federal Institute of Education, Science and Technology Farroupilha, Brazil*

Keywords: Mobile Cloud Computing, Code Offloading, Mobile Devices, Face Detection, Android.

Abstract: The popularity of mobile devices has increased significantly, and nowadays they are used for the most diverse purposes like accessing the Internet or helping on business matters. Such popularity emerged as a consequence of the compatibility of these devices with a large variety of applications. However, the complexity of these applications boosted the demand for computational resources on mobile devices. Code Offloading is a solution that aims to mitigate this problem by reducing the use of resources and battery on mobile devices by sending parts of applications to be processed in the cloud. In this sense, this paper presents an evaluation of a transparent code offloading technique, where no modification in the application source code is required to allow the smartphone to send parts of the application to be processed in the cloud. We used a face detection application for the evaluation. Results showed the technique can improve applications performance in some scenarios, achieving speed-up of 12x in the best case.

1 INTRODUCTION

The recent computing and telecommunication advances boosted the adoption of mobile devices. These devices have been used for the most diverse purposes, such as entertainment and personal care. However, as the complexity of mobile applications increases, the demand for computational resource increases as well. So, there is a concern in supplying this increasing demand without sacrificing the device's batteries (Zhou et al., 2015).

Currently, mobile devices use diverse software and hardware technologies to deliver greater convenience to their users. But, these devices still present a poor performance in comparison to desktop computers. This can affect the user experience when some application is running resource-intensive tasks like natural language processing. In addition, there is a concern about the trade-off between application


performance, battery consumption, and temperature, since increasing the computational capacity of these devices implies higher temperature and battery consumption.


In this context, Mobile Cloud Computing (MCC) emerges with the proposal of delivering greater processing power to mobile devices without adversely affecting aspects in the temperature and the energy consumption of these devices (Akherfi et al., 2016). MCC brings to mobile devices the unlimited high processing capacity provided by cloud computing. It also makes code offloading techniques feasible. Code offloading allows parts of an application to be sent and executed in the cloud. Once the code is processed in the cloud, the result is sent back to the mobile device. As a consequence of these advantages, several applications like Google Photos¹ and Apple Siri² have adopted this technique (Sanaei et al., 2014).


Several code offloading techniques have been developed (Benedetto et al., 2017; Thakur and Verma, 2015). However, most of them require explicit changes in the application source code, which limits their use, since this code is rarely available to anyone.


¹Available at: <<https://photos.google.com>>.


²Available at: <<https://www.apple.com/ios/siri>>.

^a  <https://orcid.org/0000-0001-9949-3797>

^b  <https://orcid.org/0000-0003-4945-3329>

^c  <https://orcid.org/0000-0003-3304-5611>

^d  <https://orcid.org/0000-0001-8485-529X>

^e  <https://orcid.org/0000-0002-2450-1024>

So, a transparent code offloading technique for Android devices was proposed (de Oliveira et al., 2017) to eliminate this limitation. This technique allows the use of code offloading and does not require any modification in the application code. Experiments were conducted to prove the viability of this technique. However, the experiments were performed in a local environment, using a local network to connect the smartphone and a virtual machine, which is more close to a mobile edge computing environment.

In this sense, this paper presents an evaluation of this transparent code offloading technique in a real-world cloud environment. The main objective was to evaluate whether this technique was a feasible option in a cloud computing environment and how efficient it could be. So, we used the same face detection application from (de Oliveira et al., 2017) to conducted a set of experiments, but using virtual machine instances from a public cloud provider (Google Cloud Platform) instead of a local VM. A secondary objective was to provide an evaluation of the impact of Internet latency when this code offloading technique is adopted. So, we used a set of virtual machines with heterogeneous capacities and from different regions of the world.

The remainder of this paper is organized as follows: In Section 2 we present the concepts of Mobile Cloud Computing and Code Offloading. In Section 3, the architecture of the transparent code offloading technique is explained. In Section 4 we present the specifications and configurations of the experiment environment. Section 5 is reserved for the explanation and discussion of results. In the Section 6 we present the final considerations.

2 BACKGROUND

Cloud computing is defined as a paradigm that allows access through the Internet to a shared set of configurable computational resources with unlimited capacity. Such resources can be quickly provisioned and released with minimal effort of management or interaction with the service provider. (Mell et al., 2011). Mobile Cloud Computing (MCC) enabled the mobile device to benefit from the unlimited resources provided by cloud computing (Thakur and Verma, 2015; Shiraz et al., 2013).

Since mobile applications are becoming more complex and supporting more and more functionalities, more computational resources are required to run those applications, which leads to greater battery consumption. Therefore, the code offloading technique is used to send parts of an application or even the en-

tire application to run in the cloud. This can improve application performance, as well as extend the battery life of mobile devices (Thakur and Verma, 2015; Benedetto et al., 2017; Chun et al., 2010; Qian and Andresen, 2015; Flores et al., 2015).

There are several proposals for code offloading techniques as an alternative to providing better performance and energy consumption. Some of them require the developers to modify the application code in order to define the partitioning and migration of parts from an application, so only the part with the greatest demand for computational power is sent to the cloud. Others approaches require full application migration to the cloud, but this implies communication overhead, which can increase the execution time of applications.

Unlike these related studies, De Oliveira et al. (de Oliveira et al., 2017) proposed a transparent code offloading technique for Android devices. They used Xposed Framework to transparently modify Android framework methods and send tasks to be executed remotely on a server. In this sense, the proposal does not require any changes to the firmware source code or Android system applications. To evaluate the proposal, an Android application was developed with the demand for computing resources using the native methods of the Android operating system. The authors emulated a local cloud using a traditional virtual machine running Android-x86, which is accessed by the mobile device over a wireless network. The results showed that this technique can improve execution time for several application cases, as well as minimize memory allocation.

3 ARCHITECTURE DESIGN

The environment assumed by the transparent code offloading technique is composed of *i*) Android instances of virtual machines running in a local workstation or in the cloud and accessible to *ii*) physical Android devices, which are connected to a network. In this architecture, each mobile device has installed the Xposed Framework, which can modify the behavior of a set of Android Framework methods. Each mobile device also has an Xposed Framework custom module enabled. This custom module has a set of Android framework methods or application methods. When this module is enabled, it detects when a specific method is requested and manipulates this request to run in the cloud.

Figure 1 illustrates an overview of the architecture design, where an Android device is running two applications: App A and App B; and computational

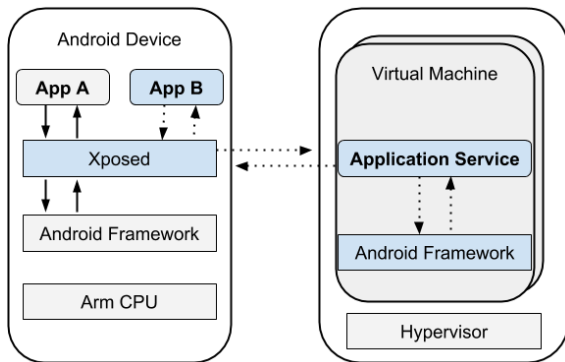


Figure 1: Architecture design overview.

infrastructure running Android instances, which has an application that plays the role of a service that process tasks received from the devices. In this case, Both applications are invoking some method from the Android framework. However, App B is invoking a method which was pre-defined to be offloaded in the custom module. Therefore, just the method invoked by App B is hooked by Xposed and sent to the cloud.

In the same context, each virtual machine runs a service that is responsible for receiving and responding to requests from mobile devices. When an application calls a method that demands more computing resources and it is defined in the Xposed framework module, the task is sent for processing in the cloud. Otherwise, the method is executed locally. Therefore, such architecture does not impact the operation of applications and methods that are not pre-defined in the module.

By using this architecture, application developers do not need to modify an existing application or create an application that sends its methods to the cloud, since the Xposed framework module is responsible for this. In the same sense, if there is a third party application that invokes the same method used by App B, the method would be offloaded as well. Therefore, the use of Xposed framework allows the code offloading technique to be performed through a transparent way to the application developer since the developer of an application does not need to program the application's communication with the server.

The transparent code offloading technique is a solution that allows the device benefits from cloud computing resources and the resource code of the Android or application is not available. At the same time, it is agnostic to Android distribution, since they have the same Android framework. Hooking Android native method allows several applications to benefit from code offloading. In addition, the device users can choose when they want to use this code offloading technique. This technique supports the implementa-

tion of decision algorithms to evaluate which part of the application and when to offload this part. However, this is out of the scope of this paper, since our goal is not to evaluate decision algorithms for code offloading.

4 EVALUATION SCENARIO

We have conducted a set of experiments in order to provide an evaluation of the transparent code offloading technique in a cloud computing environment. The environment for this evaluation has three main components: (i) Face Detection Application, (ii) Application Service Server, and (iii) Xposed Framework Custom Module.

The face detection application is the same from De Oliveira et al. (de Oliveira et al., 2017). This application uses the native Android framework method *FaceDetection* to detect faces of people in an image. We have chosen this method because it demands high computational power. This application is implemented as a regular application (there are no annotations for offloading) to be executed in the smartphone.

Application service server runs inside a virtual machine with Android-x86. This service application handles offloading requests from mobile devices. The Xposed Framework custom module connects both components. It modifies the behavior of the method *detect(Frame frame)* from class the *vision.face.FaceDetector*. Then, instead of executing the code to detect faces in the picture, the smartphone sends a request to the application service server and waits for its response, in order to get back to the application with the appropriate result. The images are available in the storage of the VM instance.

The FaceDetector Builder is set in the accurate mode in the application service server. This makes *detect(Frame)* takes longer to be executed, however, it should be more accurate, finding more faces than any other mode available. Besides, the execution time of the method on the VM is not supposed to take longer than locally on the mobile device, since the cloud has more processing power.

Figure 2 illustrates the sequence diagram of our experiment. When the face detection application calls *FaceDetector.detect(Frame)*, the Xposed Framework acts as a man in the middle, intercepting this method invocation and executing the method from the module we have developed instead of the original one. Then, it sends the ID of the image to the server. Each image has a predefined ID, enabling the application service to identify the required image. This allows the application server to handle different images. Then, the

service application receives the image ID, creating a *Frame.Builder* for the predefined image, which is locally available in the internal memory, and executes *detect(Frame)* method. Finally, it sends back to the smartphone the number of detected faces.

The method from the module receives the number of detected faces from the application server. This allows the module to create a loop for receiving the faces from the server. Right after the number of faces is sent, the application service cast each face into a string in JSON format using the Gson library and sends them, one by one, to the smartphone. Upon receiving each face, the module has to extract the Face object in JSON format from a string and append it in a generic *SparseArray*. Once all the faces are received, this *SparseArray* is returned to the application. We have used JSON since it is not possible to make a Java class serializable in this context.

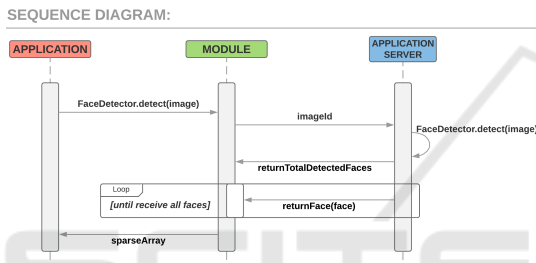


Figure 2: Sequence Diagram of the Experiment.

The experiment data gathered was based on the following metrics: time to establish the connection, time to receive faces detected from the server, the time taken by the server to detect faces, and memory usage. For this evaluation, we selected 5 images with different dimensions and number of faces. The images are presented in Figure 3 and the images specifications are available in Table 1.

Table 1: Images Specifications.

Name	Width x Height (px)	Size (KB)	Number of Faces
Image 1	1024x768	1300	2
Image 2	1024x768	914	4
Image 3	1024x768	1600	5
Image 4	1024x768	674	5
Image 5	1024x768	1270	29

Table 2: Latency and average distance between the cloud servers and the smartphone.

Cloud Region	Latency	Approximate Distance from where the experiments were conducted
Sao Paulo (Brazil)	25.7ms	12521km
Sydney (Australia)	406ms	1142.9 km

We used Ravello as an infrastructure provider. In this sense, we created and instantiated a virtual machine with Android-x86 running the application service, which performs face detection in the cloud. To

verify the impact of different computing resources available on the virtual machines on the cloud, we used the three virtual machines sizes described in Table 3.

Internet latency can influence application performance that uses code offloading techniques. Therefore, we deployed the virtual machines in two different regions (Sao Paulo and Sydney) to present the impact caused by Internet latency regarding execution time. Then, we verify the latency of the connection with the servers in the different regions and present the approximated distance between the servers and from where the experiments were conducted. The results are presented in Table 2. For each image we ran the experiment with different virtual machine sizes, running in different regions. Table 4 presents the configuration of each experiment.

In the mobile device was installed the developed application with a FaceDetection method, and the Xposed framework. The framework modified the method FaceDetection from class FaceDetector. Then, through this application, instead of executing the application to detect faces on the mobile device, it sent a request to the server application to perform this task.

The mobile device used in the experiments has the following configuration: Android 5.1.1, CPU 4x 1.1 GHz Cortex-A7, 1GB of RAM, and 16GB of storage. It is important to emphasize that the Xposed Framework installation requires root access to the mobile device, bootloader unlocked, and TWRP installed.

After the Xposed Framework installation, we installed our custom module and enabled it. Each experiment was performed ten times for each configuration presented in Table 4. The virtual machines were running the Android 6.0.1 from the open source project Android-x86 with no modification in its source code. In this virtual machine, we installed the application service server for receiving the requests from the smartphones and performs the face detection.

Table 3: Virtual Machines Specification.

Cloud VM Size	Specifications		
	Processor	Memory	Disk
Small	1vCPU	4GB	8GB
Medium	2vCPU	4GB	8GB
Large	4vCPU	4GB	8GB

5 RESULTS AND DISCUSSION

We analyzed three metrics to evaluate the efficiency of the transparent code offloading technique (de Oliveira et al., 2017). The first metric was the *total time*, that is the time taken by the face detection



Figure 3: Images adopted in the experiments.

Table 4: Experiment Specifications.

Image	Environment						
Image 1	Smartphone	Cloud VM		Cloud VM		Cloud VM	
		Small Instance		Medium Instance		Large Instance	
Image 2	Smartphone	Sao Paulo Sydney		Sao Paulo Sydney		Sao Paulo Sydney	
		Cloud VM		Cloud VM		Cloud VM	
Image 3	Smartphone	Small Instance		Medium Instance		Large Instance	
		Sao Paulo Sydney		Sao Paulo Sydney		Sao Paulo Sydney	
Image 4	Smartphone	Cloud VM		Cloud VM		Cloud VM	
		Small Instance		Medium Instance		Large Instance	
Image 5	Smartphone	Sao Paulo Sydney		Sao Paulo Sydney		Sao Paulo Sydney	
		Cloud VM		Cloud VM		Cloud VM	

application on the smartphone to show the detected faces. Total time includes the time taken to instantiate the Java objects required by face detection method and the communication time with the cloud servers in the code offloading scenarios. The second metric we analyzed was the *time to detect faces*. We considered this metric in order to evaluate the technique in a more accurate way, being able to analyze the performance impact brought by running the face detection method in the cloud, without considering external factors like communication delay. The third metric we analyzed during the experiments was the *memory usage*, that is the amount of RAM consumed by the smartphone. Given the fact that memory is one of the hardware components responsible for delivering performance to the smartphone users, we analyzed such metric to verify the memory gains that offloading applications to the cloud could bring. Moreover, memory also affects the device power consumption, so reducing the memory usage can also implicate in reducing the smartphone battery usage.

5.1 Overall Application Performance

As we can see in Figure 4 (a), using code offloading technique and the virtual machine instance in Sao Paulo increased the overall application performance according to the virtual machine specification. The best result in this region was achieved by using the medium instance, where using code offloading improved the application performance, when considering the total execution time. The medium size instance resulted in a speed up of 4.8x, while the small instance resulted in a speed up of 3.9x and 4.4x for the large instance. In this sense, we can perceive that executing the face detection method in the large virtual machine does not guarantee the best performance.

The face detection application relies on CPU processing, and as a consequence, using 2 vCPUs instead of 1 vCPU has improved the application performance in more 92%. However, increasing the virtual machine capacity to 4 vCPUs led to a 15ms overhead. This is because this application does not require too much processing to take advantage of 4 vCPUs of the large instance, so both the medium and the large instances bought close results. Through such findings, we can conclude that the medium instance has the specifications that best fit the demand for resources of the face detection application.

Executing the face detection method in the cloud server located in Sydney (Figure 4 (b)) decreased the overall application performance by approximately 344% due to the 406ms of communication delay between this cloud server and the smartphone (Table 2).

5.2 Time to Detect Faces

The results of the time to detection faces also indicated the viability of the use of code offloading through the cloud. In all cases, the virtual machines instances presented better results than the mobile device to run all application without using code offloading. All results are illustrated in Figure 4 (c) and in Figure 4 (d). The worse outcome was obtained performing the code offloading with a small cloud instance on Sydney with image 5. However, this result also presented a speed up of 3.5x in execution time if compared with the mobile device execution time result. Such value was obtained since that virtual machine has less computing resources than the other instances and the image tested present 29 faces to be detected.

The best result was collected using image 1 when was obtained performance speed up of 12x using a cloud large instance. In the same sense, image 1 presents just two faces to be detected, and the large instance presents more computing resources than small and medium instances. Then, it provides better performance during face detection by the cloud instance. In addition, if we consider just the region of Sao Paulo, the medium virtual machine presents the better performance when used to detect faces of image 1, where it was 1209% faster in the execution time.

The worst result was captured during the execution of image 5 using the small instance of the virtual machine with a speed up of 4.7x, since the image needs more computing resources to detect 29 faces. Then, when the large instances were adopted, we obtained a better execution time. The results also illustrate the impact of the number of faces in the images in each instance of the virtual machine, where it is provided better performance using large virtual machines with a small number of faces in the images.

5.3 Memory Usage

We also analyzed the memory allocated after the execution of the method *detect*. Figures 4 (e) and (f) illustrate the average RAM allocated and, as expected, less memory was allocated when executing the method in the cloud. We got RAM savings up to 55,68% for image 4 using a small instance in Sydney. This happens because, by running the method remotely, it prevents the use of memory required to run the method locally. This can lead making the battery last longer. Considering the average smartphone memory allocated for all experiments using cloud infrastructure and compared with the memory allocated running the application only locally, the experiments has allocated

42,07% less memory for Sydney and 25,87% less memory for Sao Paulo.

6 CONCLUSION

Oliveira et al. (de Oliveira et al., 2017) proposed an approach for using code offloading where no modification is required on the application source code nor changes in the Android system image. It is transparent and easy to use for Android device users and developers. It also allows the user to enable or disable the use of code offloading. However, the authors have run their experiments in a local network using virtual machines through VirtualBox. In this paper, we brought the transparent code offloading technique for Android devices to the cloud in order to verify the viability of the proposed architecture using the cloud infrastructure.

In this sense, we adopted Ravello as the cloud provider and several metrics were collected. The first metric was the time the application on the smartphone needs to perform the face detection without the use of cloud instances. We presented the evaluation of the total time to perform the face detection using the cloud, which includes aspects such as the time to instantiate the face detection method and the time to communicate with the cloud instances. In the same context, we considered the time the applications took to detect faces without considering external factors like communication delay.

Two regions were considered and five images were adopted during all experiments. Virtual machines with different computing resources were used as well. Our results showed that this technique can improve performance in terms of execution time and minimize the RAM usage by using cloud computing. They also showed that this technique is highly impacted by network infrastructure and Internet access. Results indicated performance speed up of 12x% when using the cloud to perform code offloading.

The resources from the cloud are more powerful and all applications could theoretically use this technique. However, not all of them can benefit from this technique, since not always running a task remotely is faster than locally, keeping that in mind for improving performance, the methods from the Android Framework must be computationally expensive and not require massive data transference between the smartphone and the cloud. We still intend to explore other methods from the Android Framework that can benefit from this technique.

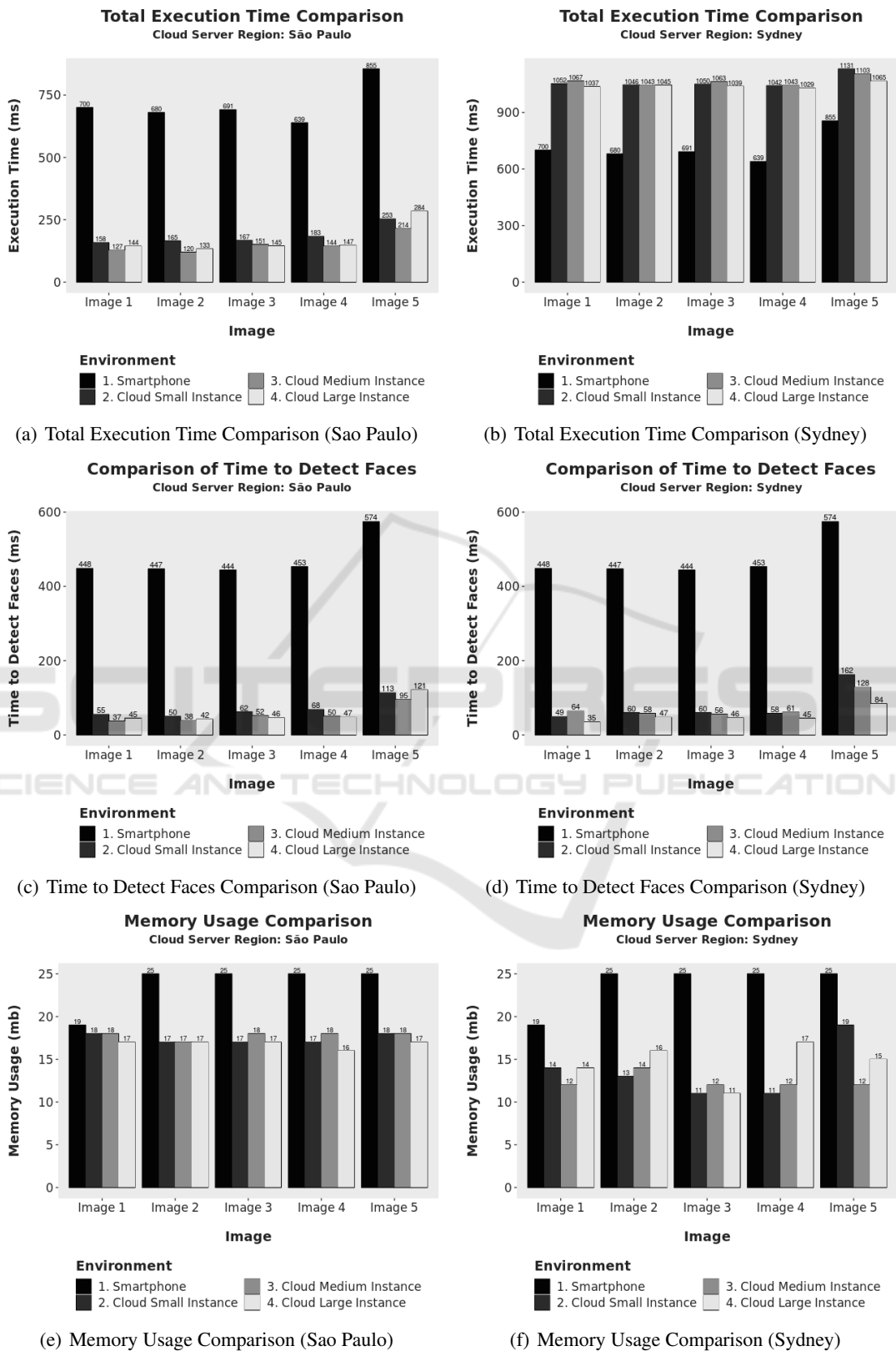


Figure 4: Comparison regarding performance and resources usage between offline processing and code offloading.

REFERENCES

- Akherfi, K., Gerndt, M., and Harroud, H. (2016). Mobile cloud computing for computation offloading: Issues and challenges. *Applied computing and informatics*.
- Benedetto, J. I., Neyem, A., Navon, J., and Valenzuela, G. (2017). Rethinking the mobile code offloading paradigm: from concept to practice. In *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pages 63–67. IEEE.
- Chun, B.-G., Ihm, S., Maniatis, P., and Naik, M. (2010). Clonecloud: boosting mobile device applications through cloud clone execution. *arXiv preprint arXiv:1009.3088*.
- de Oliveira, R. R., da Silva Schirmer, N. M., Machry, M., and Ferreto, T. C. (2017). A transparent code offloading technique for android devices. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2017 13th International*, pages 1078–1083. IEEE.
- Flores, H., Hui, P., Tarkoma, S., Li, Y., Srirama, S., and Buyya, R. (2015). Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine*, 53(3):80–88.
- Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing.
- Qian, H. and Andresen, D. (2015). Extending mobile device's battery life by offloading computation to cloud. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, pages 150–151. IEEE Press.
- Sanaei, Z., Abolfazli, S., Gani, A., and Buyya, R. (2014). Heterogeneity in mobile cloud computing: taxonomy and open challenges. *IEEE Communications Surveys & Tutorials*, 16(1):369–392.
- Shiraz, M., Gani, A., Khokhar, R. H., and Buyya, R. (2013). A review on distributed application processing frameworks in smart mobile devices for mobile cloud computing. *IEEE Communications Surveys & Tutorials*, 15(3):1294–1313.
- Thakur, P. K. and Verma, A. (2015). Hybrid process cost evaluation method in mobile code offloading. In *Next Generation Computing Technologies (NGCT), 2015 1st International Conference on*, pages 149–152. IEEE.
- Zhou, B., Dastjerdi, A. V., Calheiros, R. N., Srirama, S. N., and Buyya, R. (2015). A context sensitive offloading scheme for mobile cloud computing service. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 869–876. IEEE.