Hindawi
Scientific Programming
Volume 2020, Article ID 5304612, 9 pages
https://doi.org/10.1155/2020/5304612

Research Article

MCAF: Developing an Annotation-Based Offloading Framework for Mobile Cloud Computing

Yilian Zhou,[1] Ligang He ,[2] Bin Wang,[3] Yi Su,[2] and Hao Chen[1]

[1]*College of Computer Science and Electronic Engineering, Hunan University, Changsha, China*
[2]*Department of Computer Science, University of Warwick, Coventry, UK*
[3]*ZTE Corporation, Shenzhen, China*

Correspondence should be addressed to Ligang He; ligang.he@warwick.ac.uk

Received 29 November 2019; Revised 24 February 2020; Accepted 7 May 2020; Published 16 July 2020

Academic Editor: Daniele D'Agostino

Offloading computation from mobile to remote cloud servers is a promising way to reduce energy consumption and improve the performance of mobile applications. However, a great challenge arises as automatic integration of powerful computing resources in remote cloud infrastructure and the portability of mobile devices. In this paper, we develop a Java annotation-based offloading framework, called MCAF, for android mobile devices. This framework is designed and committed to simplifying the development of android applications enabled with the offload capability. All the developers need to do is to import the SDK library of our MCAF and annotate the computation-intensive methods. MCAF can automatically extract the annotated source code and generate the code that will be run in the Cloud. Moreover, the codes of making the offloading decisions are automatically inserted into the original source code. We also conducted the real experiments to show the applicability of our MCAF.

1. Introduction

Advances in the portability and capability of mobile devices such as smartphones, together with increasingly faster and widespread wireless networks, a large number of mobile APPs such as shopping, gaming, information management, and so on have been made. Smartphones have changed human lives and become indispensable gadgets in modern society. Despite the fast increase in hardware performance of mobile devices, it is still limited comparing with their desktop counterparts and cannot meet the ever-increasing demand from end-users and APP developers (e.g., CPU, storage, and battery life). The limited hardware resources impede further improvement of Quality of Services (QoS) [1] for users and also further expansion of mobile APP categories. Integrating mobile devices with powerful Cloud platform [2, 3] to provide the Mobile Cloud Computing (MCC) offers a promising solution to overcoming the challenge [4, 5].

MCC is able to save the energy consumption of mobile devices and/or improve the running performance of the mobile applications by offloading part of executions from mobile devices to cloud servers. Although various offloading frameworks have been developed, a big obstacle to hinder the wide adoption of these offloading frameworks is that it relies heavily on the programmers to develop the mobile applications in a manner designated in the offloading framework. In this work, we aim to design and develop an offloading framework which simplifies the development of mobile applications. Our offloading framework, called MCAF, makes use of Java annotation, which is different from the existing ones in the following aspects.

In all existing offloading frameworks, the programmers either have to annotate the source code or write the source code in a specific manner, and the framework developers need to develop the bespoke compiler to compile the source code embedded with the offloading ability. To use MCAF, there is no need to develop a new compiler, and only the existing Java compiler is needed to realize the offloading process.

Moreover, most existing frameworks such as Cuckoo [6] require the programmers to implement the code to be run on the Cloud. In contrast, MCAF aims to minimize the effort of conducting the offload. The programmers who employ MCAF do not need to implement the code running on the Cloud but write the source code as normal for the mobile device and add the Java annotations that specify the code offloading. MCAF automatically extracts from the source code the part that is to be run on the Cloud according to the Java annotation. The extracted code is then encapsulated automatically as a stand-alone Java program and is offloaded to the Cloud when the conditions are met.

Another major difference between MCAF and most existing frameworks is that MCAF offloads the task at the granularity of the method level. MCAF extracts the source code of a method that is to be offloaded and then wraps up the method as a stand-alone program.

Overall, MCAF aims to reduce the efforts of the framework developers and the programmers in realizing the offloading capability in mobile Apps. All that the users of MCAF have to do is to import the MCAF SDK library and add annotation for computation-intensive methods. In this paper, we present the design and implementation of our offloading framework. In order to showcase the applicability of our approach, we also developed two mobile applications for real-life scenario by applying our offloading framework. Experiments have also been conducted to evaluate the effectiveness of our framework.

Note that in practice confidentiality and the integrity of the transmitted data should be ensured. There are the existing measures in the literature [7, 8] to address the security issues in the offloading framework. For example, in the work presented in [7], the data are encrypted and signed before transmitting, and are then decrypted in the Cloud. The encryption and decryption costs are taken into account when making the offloading decisions. Ensuring security is not the focus of this paper. MCAF can make use of the existing security measure in the literature (such as the one presented in [4]) to ensure confidentiality and the integrity.

The remainder of this paper is organized as follows. In Section 2, we provide the background of Java annotation and the related work. In Section 3, we present the design of MCAF and also a case study to show the exact workings of MCAF. MCAF is evaluated in Section 4. This paper is concluded in Section 5.

2. Background

2.1. Annotation. Annotation is an important feature in Java since it relives Java developers from the pain of cumbersome configurations [9]. It shifts the responsibility of writing the boilerplate Java code from the programmer to the compiler. Annotations were first introduced in Java 5.0 and more advanced features are gradually supported in later Java versions.

Annotation is a form of metadata and can be used to describe any information about the elements (packages, classes, methods, fields, arguments, variables, etc.) in a program [10]. Annotations do not directly affect the program semantics. However, since annotation does affect the way in which the programs are treated by tools and libraries, they can in turn affect the semantics of the running program. Annotations can be read from source files, class files statically, or be read reflectively at runtime. For example, annotations, such as Deprecated, Override, or NotNull, can be used to describe the constraints or the usages of the elements in a method during the compilation.

Although typically an application programmer never have to define an annotation type, it is not hard to do so. The declaration of an annotation type is similar to that of a normal interface. An at-sign (@) precedes the keyword "interface." The declaration of a method will also define the elements of the annotation type. The declaration of a method must not have any parameters and can have default values. Return types are restricted to primitives, String, Class, enum, annotation, and arrays of these types. The following is an exemplar declaration of an annotation type. Once an annotation type is defined, program developers can use it to annotate source code. In general, annotations will be processed during the compilation. The command-line utility APT, an annotation processing tool, is used to find and execute the annotation processors based on the annotations inserted in the source files.

2.2. Related Work. Many researchers believe that combining mobile computing with clouds is a promising solution to overcome the battery limitation of smartphones and extend the performance of smartphones [11]. Indeed, much recent work has focused on building the frameworks that enable the offloading of mobile computations to the cloud [12, 13].

Cuervo et al. proposed an offloading system called MAUI, which can enable fine-grained energy-aware code offloading from smartphone to the remote server [14]. In MAUI, a method that is possible to be offloaded is declared as a "remotable" method. MAUI then models the offloading problem as an integer programming problem and finds the optimal offloading decision by solving the integer program. Microsoft.NET Common Language Runtime (CRL) is the programming language used by MAUI. However, the drawback of MAUI is that it needs the developers to modify the source code run on the local mobile and implement the source code run in the Cloud.

Compared with MAUI, CloneCloud [15, 16] goes one step further and does not ask the programmer to label (e.g., declare) the offloadable methods. CloneCloud automatically identifies the offloading costs by analyzing the source code both statically and dynamically at runtime. It then runs an optimizer to partition the tasks between mobile devices and the Cloud.

MAUI and CloneCloud are the systems for offloading parts of the existing program running on the mobile device to the Cloud. Cuckoo [6, 17] is the application development framework. Cuckoo can be used by programmers to develop the mobile Apps that have the offloading ability. Cuckoo provides a simple programming model and allows a single interface with a local and a remote (on the Cloud) implementation. The Apps developed using Cuckoo will decide at runtime automatically whether the local or the remote implementation is invoked for an interface.

DPSF [18] is another offloading framework based on Java. It first analyses the call-graph of the application offline and determines which classes can be offloaded. It then partitions all Java classes into two parts during the compilation. The classes in one part are run locally in the mobile phone while the other will be offloaded to run on the server. The entire application is deployed in both local mobile and remote server. The application starts running in local mobile. When it runs to a method in a class that is offloaded, it makes use of the RMI (Remote Method Invocation) mechanism in Java to invoke the method in the Java application deployed in the remote server. Although DPSF also makes use of the existing ability in Java to realize offloading, there are the following differences from our MCAF.

First, DPSF is essentially a static offloading framework because it, whose classes are offloaded, has to be determined at the programming stage through the offline profiling, and then the source code has to be adjusted to fulfil the offloading. DPSF is dynamic only in the sense that when it begins to run the application but detects that the network connection is poor, it will run the original Java application. Once the original application or the application with the offloading ability starts running, it will run to completion. There is no other mechanism in DPSF that makes the offloading decisions for individual classes during the application execution. In MCAF, although we annotate which methods can be offloaded at the programming stage, the offloading decisions can be made for a particular method. If the offloading condition is met, the offloading is then carried out for that method.

Second, DPSF offloads the workload at the granularity of the class level, while it is at the method level in MCAF.

Third, when using DPSF, the programmer has to change the source code, for example, changing the modifier of non-private fields to private and generating the public getter/setter methods for them, generating a proxy class for each offloaded class, and so on. In MCAF, the programmer only annotates the methods through Java annotation scheme, but does not change the original source code, which reduces the programmer's effort and is less error-prone during the development stage. The source code for running the offloaded methods in the Cloud server is automatically generated by MCAF, not written by the programmer. If the annotation is ignored, the application behaviour remains unchanged.

3. MCAF

3.1. MCAF Modules. The Mobile Cloud Annotation Framework (MCAF) consists of the following modules: (1) Annotation Handler: extracting all information related to the annotated methods; (2) Cloud Proxy: realizing the communications between the local mobile device and the Cloud; (3) Offloading Decider: implementing the offloading strategies; and (4) Code Rewriter: automatically wrapping up the source codes of the extracted method as stand-alone Java programs and compiling them as classes files.

3.1.1. Annotation Handler. APT is the existing annotation processing tool. In MCAF, APT is used to identify which method is annotated and save the annotation information. The saved annotation information contains the accessing modifiers, the return type, and the parameters type of the annotated methods. However, the parameter values and the method code are not included. One of the responsibilities of Annotation Handler is to complement the annotation information saved by APT. Annotation Handler extracts the segment of the source code in each annotated method. Specifically, a regular matching expression is constructed in Annotation Handler, which is used to search the annotation information saved by APT and to match the part of source code which implements the annotated methods. The extracted code segment will be packaged as the stand-alone Java program and compiled to the class files by the Code Rewriter module. The class files will be transferred to and run on the Cloud according to the offloading decision made by Offloading Decider.

3.1.2. Offloading Decider. After the processing of Annotation Handler, all annotated methods are identified, and the annotation information and the source file of these methods are saved. When an annotated method is invoked, the Offloading Decider module calculates the offloading cost of this method. If it is beneficial to offload, the bytecode of this method, which is obtained by the Code Rewriter module, is retrieved and transferred to the cloud by the Cloud Proxy module. If the Offloading Decider decides not to offload, the method will be executed in the local mobile device.

Note that the offloading strategy is not the focus of this paper. MCAF is a framework to realize the offloading ability. The developer can plug any existing offloading strategy into the Offloading Decider. The Offloading Decider interfaces with other components in MCAF through its output, which is either True or False. If the output is true, the method in question will be offloaded. Otherwise, the method will be run locally. Algorithm 1 in Section 3.2 presents an example of this.

3.1.3. Cloud Proxy. The Cloud Proxy module is responsible for offloading-related communications between local mobile device and the Cloud. Firstly, when the Offloading Decider decides to offload an annotated method, Cloud Proxy transfers the bytecode and the arguments of the method to the Cloud. Second, the execution results of the offloaded method are returned to the local mobile device and passed to the invoked method.

3.1.4. Code Rewriter. Code Rewriter generates two types of source code (class). After Annotation Handler extracts the code segment of the annotated methods, Code Rewriter generates the stand-alone source code for each annotated method. The source code is then compiled to a Java class, whose name is the name of the annotated method appended by a word "Class." For example, if the name of the annotated method is "Add," the name of the new class is "AddClass." The generated classes may be transferred to and run on the cloud after the offloading decision is made. In the class, a member function is defined which includes all instructions of the annotated method. By invoking the member function of the newly generated class on

```
 (1) public class A_ extends A {
 (2)    int Add(int x, int y) {
 (3)       isAddClassNeedUpload = Decider.getClassUploadStatus();
 (4)       if (isAddClassNeedUpload) {
 (5)          isAddClassRemoteExists = Decider.getRemoteExistsStatus();
 (6)          if (!isAddClassRemoteExists) {
 (7)             ClientProxy.uploadClass("xx/xxx/AddClass.Class");}
 (8)          return ClientProxy.getResult("Add", x, y);
 (9)       }
(10)       return x + y;
        }
     }
```

Algorithm 1: Generating a new class "A_" that extends from class "A" and then overwriting the "Add" method by inserting the offloading instructions.

the cloud, we will obtain the same results of the annotated method as it runs on the local smartphone. We call this type of class the *Cloud execution* class.

The second type of source code (class) generated by Code Rewriter is run in the local smartphone. In this source code, the new class inherits from the class where the annotated method is located. The name of this new class is the name of the inherited class appended by an underscore symbol"_." For example, if the inherited class is "A," the name of the new class is "A_." Code Rewriter overrides the new class "A_" by automatically inserting the code which makes the offload decisions and the code that interacts with the Cloud, such as uploading the "AddClass" generated above, and its input parameters to the Cloud and receiving the return results from the Cloud. If the decision made by the added decision making code is false (i.e., do not offload), the original instructions of the annotated method will be run in the local smartphone. We call this new class the *policy and local execution* class.

3.2. A Case Study of MCAF.

In this subsection, we present an example to illustrate the workings of MCAF.

We first define a new annotation named "Upload" in Algorithm 2.

Target and *Retention* are two predefined annotation types in Java. The *Target* type with the value of *ElementType.METHOD* in line 1 is used to specify that the annotation is applied at the method level. The Retention type with the value of *RetentionPolicy.CLASS* in line 2 is used to specify that the added annotation is retained by the compiler at the compile time (but is ignored by the Java Virtual Machine). Line 3 defines a new annotation type called *Upload*. Defining a new annotation type is similar as defining an interface in Java except that the keyword *interface* is preceded by the sign @. In the body of the *Upload* annotation type, three annotation type elements are declared: type, module, and valueType. The *type* element is used to identify the *Upload* annotation. The *module* element is used to specify the module in which the annotated method is located (assume the annotated method is in the "app" module). valueType is the data type of the input parameters of the annotated method. We developed an Annotation.jar package. The "Upload" annotation type is defined in Annotation.jar. Annotation.jar also implements the functionality of the Code Rewriter component.

After the Upload annotation type is defined, we can use it to annotate a method. In Algorithm 3, a class with the name of "A" is defined, which includes an "Add" method. The "Add" method takes two input parameters with the integer type, x and y. Before the "Add" method, the newly defined "Upload" annotation type is inserted, in which the valueType element takes the actual types (i.e., integer) of the two input parameters, x and y. The other two elements, "type" and "module," take the default values.

When the code in Algorithm 3 is compiled, the compiler realizes that the "Add" method is annotated by "Upload." Two types of source code (two classes) will be generated.

On one hand, the compiler will invoke the Code Rewriter functionality implemented in Annotation.jar to generate a new class named "AddClass" as shown in Algorithm 4. If the offload decider decides to offload the "Add" method, the "AddClass" will be uploaded to the Cloud for execution.

On the other hand, the compiler will also invoke the Code Rewriter functionality to generate another new class named "A_," which extends from class "A" (the "Add" method is located in class "A"). Then the "Add" method is overwritten by inserting a set of instructions shown in Algorithm 1 (from line 4 to line 10) in the "Add" method. In Algorithm 1, line 3 calls the offload decider to determine whether the "Add" method should be uploaded. If the output of the Offload Decider is True (checked in line 4), the method needs to be offloaded and lines 5–8 will be run. Lines 5 and 6 check whether the "Add" method has been offloaded (the App may be executed repetitively). If it has been offloaded, it means that the code of the "Add" method exists in the Cloud and there is no need to upload the code of the method again. If the method has not been offloaded before, it calls the Cloud Proxy to upload the "AddClass" (line 7), which contains the code of the "Add" method that will be run in the Cloud, and then wait for the Cloud to return the result of running the "Add" method (line 8). If the out of the Offload Decision is False, the calculation is performed locally (line 10).

When the Cloud server receives the "AddClass.Class" uploaded by the smartphone, it invokes the "Add" method in the AddClass.Class through the Java reflection mechanism. The segment of code that run the "Add" method in the Cloud

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.CLASS)
public @interface Upload {
   String type() default "Upload";
   String module() default "app";
   String[] valueType() default {};
}
```

Algorithm 2: Defining a new annotation type named "Upload."

```
public class A extends MainActivity {
   @Upload(valueType = {"int," "int"})
   int Add(int x, int y) {
      return x + y;
   }
}
```

Algorithm 3: Annotating the "Add" method in the "A" class.

```
package org.cloud.annotations.MainActivity;
public class AddClass {
   public int Add(int x, int y) {
      return x + y;
   }
}
```

Algorithm 4: Generating the "AddClass" to be run on the Cloud.

```
Object[] parameters = new Object[]{x, y};
Class[] parametersType = new Class[]{int.class, int.class};
Object[] parameters = (Object[]) entityBean.parametersValue.toArray();
Class[] parametersType = entityBean.parametersType.toArray(new Class[entityBean.parametersType.size()]);
Class<?> cl = Class.forName("org.demo.data.AddClass");
Method method = cl.getDeclaredMethod("add," parametersType);
Object object = cl.newInstance();
ret = (int) method.invoke(object, parameters);
```

Algorithm 5: The Cloud server executes the "Add" method in the AddClass.Class through the Java reflection scheme.

is shown in Algorithm 5, in which x and y will take the values uploaded by the smartphone.

3.3. System Architecture. In this section, we present the system architecture for implementing the Android application with the offload capability. First, we introduce what type of methods can be annotated using our scheme. Android applications run their bytecode on the Dalvik Virtual Machine (Dalvik VM). There are some differences between Dalvik VM and Java Virtual Machine (JVM). Dalvik VM has optimized the execution of the android code. For example, UI graphic, camera, and sensors cannot be executed on JVM because their execution depends on the Android Runtime Environment. The methods which can be annotated must meet the following conditions: (1) the annotated methods can only contain the pure java code. Note that the percentage of the code that is offloaded will not be significantly reduced due to this condition since the methods that are most likely to be offloaded are computation-intensive ones and all arithmetic operations can be programed in pure Java code. (2) The annotated methods should not use the global variables. If the global variables have to be used, they need to be transferred to the Cloud server together with other method parameters. (3) The annotated methods have to be public methods.

3.3.1. Build Process. Figure 1 shows the process of building an Android application using our framework from the source code to the apk file. The only thing that the developers

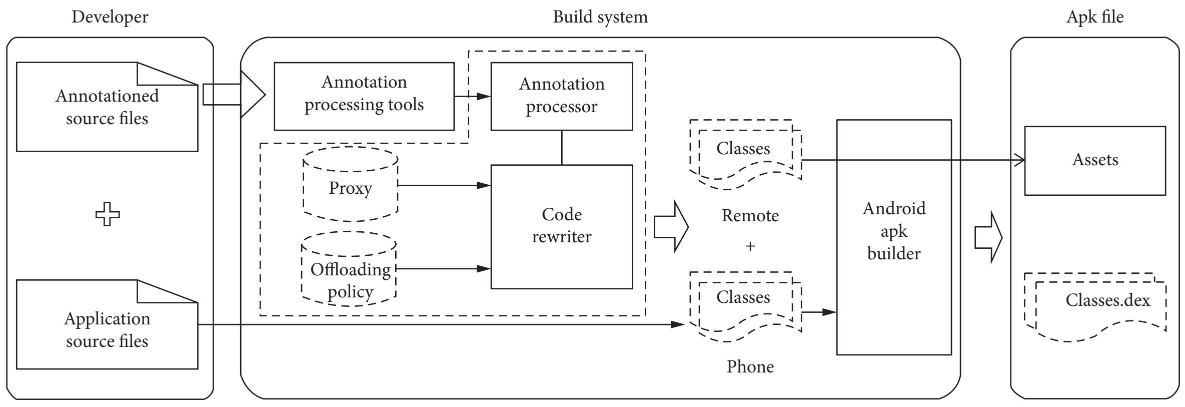

Figure 1: The build process of an android application with MCAF.

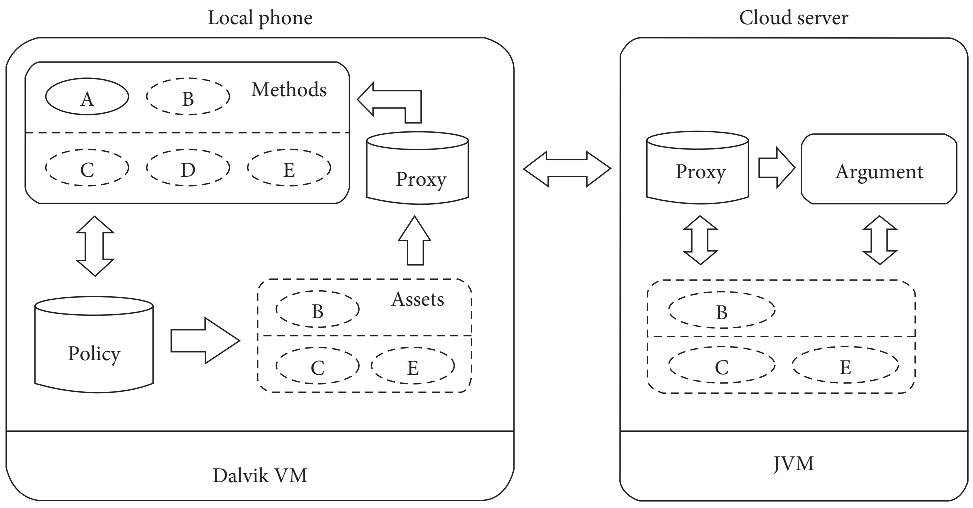

Figure 2: The runtime architecture of android application generated by the framework.

need to do is to add the annotation, which divides the source files into two categories: annotation source files and the original source files. In this process, Annotation Process Tools (APT) can find all annotated methods in annotated source files. Then, it generates the *Cloud execution* classes that can be executed on the Cloud and the classes that run on the local smartphone through the framework SDK.

3.3.2. Runtime Architecture. Figure 2 shows the runtime architecture of the application. In the smartphone, the runtime architecture is mainly divided into four parts. The first part is Methods, which contains the methods that will be executed when the android application runs. In Figure 2, there are two classes in the methods part, which are divided by the dotted line. Methods A and B are in the same class, while methods C, D, and E are in the other. In addition, we use the dotted ellipse to represent the annotated methods, which are methods B, C, and E. The second part is Policy, which makes the offload decisions for the annotated methods. The third part is assets; it is a resource folder and contains the source code of the annotated methods, which can be uploaded and executed on the Cloud. The last part is Proxy; it is responsible for communication with the cloud. In the Cloud server, the module argument is used to receive the arguments from local smartphone when an annotated method is executed in the Cloud.

In the Cloud server, the argument module is responsible for managing the transmission of the parameters between the smartphone and the server. When an annotated method is invoked in the local phone during runtime, the method will wrap the parameter values into the serialized file stream, which is transmitted to the server through the network and is deserialized to the parameters value on the server. By using the serialization, we can transmit the complex type parameters easily in addition to basic types of data. On the Cloud server, we allocate the memory space to store the executable files uploaded from the smartphone. These files are invoked through the Java reflection mechanism. The required parameters during the execution are provided by the argument module. When the Cloud server obtains the execution results, we will use the serialization mechanism to pass the results to the smartphone.

4. Evaluation

The aim of developing the offload framework is to save the energy consumption of the smartphone where the mobile

Table 1: The execution time of the matrix computation application.

Power n	Offloading (ms) Transmission	Offloading (ms) Cloud	Local phone (ms)	Speedup (local_run/offloading_run)
20	237	61	6912	23.19
40	232	77	13742	44.47
60	232	102	20610	61.71
80	214	128	27468	80.32
100	210	152	34551	95.44

Table 2: The execution time of the Image Segmentation application.

Times	Offloading (ms) Transmission	Offloading (ms) Remote	Local phone (ms)	Speedup (phone/offloading)
2	969	20	3727	3.85
4	984	27	7485	7.61
6	1003	29	11201	11.17
8	1065	43	15022	14.11
10	1067	93	18710	17.54

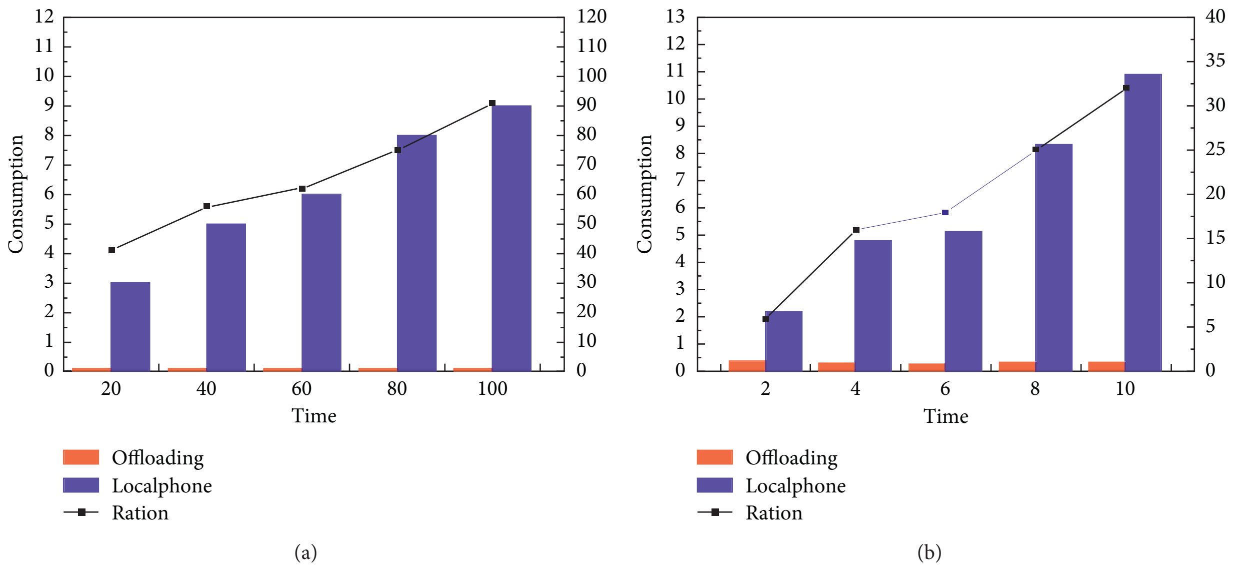

Figure 3: (a) The energy consumption of the matrix computation application; the x-axis is the number of times the matrix multiplication is performed (i.e., the power n). (b) The energy consumption of the image segmentation application; the x-axis is the number of times the image segmentation is performed.

App is running while maintaining the performance of the App. In this section, we evaluate the performance and power consumption of our MCAF. We conducted real experiments on a smartphone, which is Vivo X20 with 1.8 GHz CPU, octa core processors, 4 GB RAM and Android OS 7.1.1. We used a desktop as a Cloud server, which runs ubuntu 14.04 with a 3.40 GHz CPU and 16 GB RAM. The Cloud server and smartphone are interconnected by wifi.

We implemented two applications and ran them either on the local phone or on the cloud server through offloading. The first application is to perform matrix computation, which is a computational intensive application. In this application, we generated the square matrix with the size of 100×100. The elements of the matrix are random numbers between -10 and 10. The application computes the n-th Power of the matrix. First, we ran the application on the smartphone. Then, we ran the application by offloading it to the Cloud server. We also measured the power consumption of the smartphone and the execution time of the application in both cases. How to design smart offloading strategies is not the focus of this work. In the experiments, the application is always offloaded when the wireless connection is available.

4.1. Execution Time. Table 1 lists the execution times of the application when running in the smartphone and when offloading. The offloading execution time consists of 2 parts, including the communication time of uploading the code and the parameters to the Cloud and the execution time on the Cloud server. As the table shows, the communication time is relatively stable. This is because the data transmitted and the network speed during the offloading process change little. The execution time on the Cloud server increases gradually as the power n (i.e., the number of matrix multiplication) increases. We also calculate the speedup of running through offloading over running in the smartphone. As the computation increases, the speedup increases too. The results show that our offload framework can significantly improve the running performance of the application.

The second application we used in the experiments is image segmentation, which partitions a digital image into multiple segments. It is also a computation-intensive application. The execution times of the application in local phone and in the Cloud server are shown in Table 2. The input size of the test image is 300×300 pixel. As the table shows, the application incurs more communication time compared to the matrix computation application. The results still show good speedup when the application is offloaded.

4.2. Energy Consumption. To evaluate the energy consumption of these applications, we used software called Trepn Profiler [19]. It is an on-target power and performance profiling application for mobile devices. In the Trepn Profiler, the power measurement for a single app can be achieved with a feature called Show Deltas. Figures 3(a) and 3(b) show the energy consumption of these two applications when they are executed locally and through offloading. It can be observed from the figure that offloading offers significant energy saving. The amount of energy saved increases as the computation size increases.

5. Conclusion

This paper presents a Java annotation-based offload framework called MCAF for mobile cloud computing. MCAF is used to upload the annotated source code to the Cloud server at the granularity of methods. By using MCAF, developers do not need to implement the Cloud-side service, which can be generated automatically by MCAF during the compilation. The codes of making the offloading decisions are also automatically inserted. All that the developers need to do is to import the SDK library of our MCAF and annotate the computation-intensive methods. We conducted the real experiments to showcase the applicability of MCAF.

It is possible to extend this work to allow the offloading of the Android-related methods (i.e., the methods that are not programed in pure Java code, but call the functions in Android-related libraries). In order to realize the offloading of android-related code, the following work should be conducted: (i) extracting the offloaded method and generating a stand-alone apk that encapsulates the offloaded method; (ii) setting up a VM in the Cloud and deploy the Dalvik VM environment in the VM; (iii) developing a functional component in the Cloud VM to enable the communication between the Dalvik VM and the mobile. The development of the above framework is expected to involve much more engineering work than deploying a standard Java VM as in the current MCAF. We will carry out careful investigations as to whether it is worth the effort. After all, only computation-intensive tasks are most likely to be offloaded, which can be coded in pure Java code.

MCAF is originally designed for offloading the workload from smartphones to the Cloud server. We plan to explore the application of MCAF in fog-edge computing. In fog-edge computing, data processing is moved to edge devices, which are closer to where the data are generated compared with Cloud computing. As long as the edge devices are more powerful than smartphones, there is no reason why MCAF cannot be used to achieve the offloading of the workload from smartphones to edge devices in fog-edge computing.

Data Availability

The data supporting the results of this work are available upon request from the corresponding author.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was partially funded by National Natural Science Foundation of China under Grants 61972137 and 61772183 and the Worldwide Byte Security Co. Ltd.

References

[1] J. Mei, K. Li, and K. Li, "Customer-satisfaction-aware optimal multiserver configuration for profit maximization in cloud computing," *IEEE Transactions on Sustainable Computing*, vol. 2, no. 1, pp. 17–29, 2017.

[2] K. Li, C. Liu, K. Li, and A. Y. Zomaya, "A framework of price bidding configurations for resource usage in cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2168–2181, 2016.

[3] C. Liu, K. Li, C. Xu, and K. Li, "Strategy configurations of multiple users competition for cloud service reservation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 508–520, 2016.

[4] H. Qi and A. Gani, "Research on mobile cloud computing: review, trend and perspectives," in *Proceedings of the 2nd IEEE International Conference on Digital Information and Communication Technology and It's Applications (DICTAP)*, pp. 195–202, IEEE, Bangkok, Thailand, May 2012.

[5] 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*, vol. 13, no. 18, pp. 1587–1611, 2013.

[6] C. A. I. Long, K. K. H. Kunasekaran, V. Ramakrishnan et al., "Task offloading to the cloud by using cuckoo model for minimizing energy cost," in *Proceedings of the 2016 International Conference on Information Engineering, Management and Security (ICIEMS 2016)*, Tirupur, India, 2016.

[7] B. Huang, Y. Li, Z. Li et al., “Security and cost-aware computation offloading via deep reinforcement learning in mobile edge computing,” *Wireless Communications and Mobile Computing*, vol. 2019, Article ID 3816237, 20 pages, 2019.

[8] V. Viduto, C. Maple, W. Huang, and D. López-Peréz, “A novel risk assessment and optimisation model for a multi-objective network security countermeasure selection problem,” *Decision Support Systems*, vol. 53, no. 3, pp. 599–610, 2012.

[9] W. Cazzola and E. Vacchi, “@Java: bringing a richer annotation model to Java,” *Computer Languages, Systems & Structures*, vol. 40, no. 1, pp. 2–18, 2014.

[10] http://tutorials.jenkov.com/java/annotations.html.

[11] J. Liu, K. Li, D. Zhu, J. Han, and K. Li, “Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems,” *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 2, p. 36, 2016.

[12] Y. Wang, I.-R. Chen, and D.-C. Wang, “A survey of mobile cloud computing applications: perspectives and challenges,” *Wireless Personal Communications*, vol. 80, no. 4, pp. 1607–1623, 2015.

[13] M. A. Khan, “A survey of computation offloading strategies for performance improvement of applications running on mobile devices,” *Journal of Network & Computer Applications*, vol. 56, pp. 28–40, 2015.

[14] E. Cuervo, A. Balasubramanian, D. Cho et al., “MAUI: making smartphones last longer with code offload,” in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ACM, San Francisco, CA, USA, pp. 49–62, June 2010.

[15] B. G. Chun, S. Ihm, P. Maniatis et al., “Clonecloud: elastic execution between mobile device and cloud,” in *Proceedings of the 6th Conference on Computer Systems*, ACM, Salzburg, Austria, pp. 301–314, April 2011.

[16] L. P. R. Mali and L. G. D. Naik, “A review on distributed application processing framework-clone cloud,” *International Journal of Engineering and Computer Science*, vol. 4, no. 1, 2015.

[17] R. Kemp, N. Palmer, T. Kielmann et al., “Cuckoo: a computation offloading framework for smartphones,” in *Proceedings of the 2010 International Conference on Mobile Computing, Applications, and Services*, pp. 59–79, Los Angeles, CA, USA, October 2010.

[18] D. Kong, T. Qi, T. Yang et al., “A dynamic computation offloading framework for Android,” in *Proceedings of the 5th IEEE International Conference on Broadband Network & Multimedia Technology*, pp. 134–138, IEEE, Guilin, China, November 2013.

[19] https://developer.samsung.com/game/trepn.