Received: 15 May 2018 | Revised: 18 July 2018 | Accepted: 24 September 2018

DOI: 10.1002/ett.3526

SPECIAL ISSUE ARTICLE

WILEY

The importance of granularity in multiobjective optimization of mobile cloud hybrid applications

Aamir Akbar | Peter R. Lewis

The Aston Lab for Intelligent Collectives Engineering (ALICE), Aston University, Birmingham, UK

Correspondence
Aamir Akbar, The Aston Lab for Intelligent Collectives Engineering (ALICE), Aston University, Birmingham, UK
Email: akbara2@aston.ac.uk

Funding information
Abdul Wali Khan University Mardan (AWKUM)

Abstract

Mobile devices can now support a wide range of applications, many of which demand high computational power. Backed by the virtually unbounded resources of cloud computing, today's mobile cloud (MC) computing can meet the demands of even the most computationally and resource-intensive applications. However, many existing MC hybrid applications are inefficient in terms of achieving objectives like minimizing battery power consumption and network bandwidth usage, which form a trade-off. To counter this problem, we propose a data-driven technique that (1) does instrumentation by allowing class-, method-, and hybrid-level configurations to be applied to the MC hybrid application and (2) measures, at runtime, how well the MC hybrid application meets these two objectives by generating data that are used to optimize the efficiency trade-off. Our experimental evaluation considers two MC hybrid Android-based applications. We modularized them first based on the granularity and the computationally intensive modules of the apps. They are then executed using a simple mobile cloud application framework while measuring the power and bandwidth consumption at runtime. Finally, the outcome is a set of configurations that consists of (1) statistically significant and nondominated configurations in collapsible sets and (2) noncollapsible configurations. The analysis of our results shows that from the measured data, Pareto-efficient configurations, in terms of minimizing the two objectives, of different levels of granularity of the apps can be obtained. Furthermore, the reduction of battery power consumption with the cost of network bandwidth usage, by using this technique, in the two MC hybrid applications was (1) 63.71% less power consumption in joules with the cost of using 1.07 MB of network bandwidth and (2) 34.98% less power consumption in joules with the cost of using 3.73 kB of network bandwidth.

1 | INTRODUCTION

In recent years, there has been a growing interest to bind virtual resources to low-power devices such as smartphones.[1] To make mobile devices virtually limitless in terms of processing power, energy, and storage space, the integration of cloud

computing technology[2] and mobile devices is often done. This interdisciplinary domain is called mobile cloud computing (MCC).[3] It is a distributed and augmented program execution model, upon which mobile cloud (MC) hybrid applications are developed.[4] Such applications have the advantage of using the vast majority of services provided by both mobile and cloud computing. The MC hybrid applications are becoming even more frequently relied upon. For example, to find a route between two locations, a navigation application installed on a mobile device can use the stored maps, but to find the real-time traffic on that route, the application would need the cloud service.

A battery-operated mobile device (ie, a smartphone) can be connected to the cloud via a WiFi or a mobile data network (3G/4G). From a mobile device point of view, the decision to execute a computationally intensive task locally on the device can demand high battery power consumption. For example, when an MC hybrid app is doing image processing or solving a long and complex mathematical expression, it consumes the device battery power. From a network connection point of view, the decision to execute the computationally intensive task remotely on the cloud can be affected by a high bandwidth usage. For example, when an MC hybrid app offloads to the cloud too often or a large amount of data is transferred between the two endpoints (mobile device and cloud server). Therefore, attaining an optimal configuration or a set of optimal configuration of an MC hybrid application, while not affecting the overall performance of the app, is considered one of the challenging areas in MCC.[5]

We consider the effective partitioning of MC hybrid application as a multiobjective optimization problem. In multiobjective optimization, there are a number of objectives to be optimized simultaneously, and, typically, the objectives are conflicting with each other. There exists a natural trade-off between objectives, which creates a set of Pareto-optimal solutions,[6] those that are not dominated by any possible other solution in the solution space. Optimizing multiple objectives, we might obtain Pareto-optimal configurations in a configuration set of an MC hybrid app. The objectives we consider are the following:

1. Minimize power consumption: number of joules consumed by the application on a mobile device during one execution.
2. Minimize network bandwidth: data sent and received between a mobile device and a cloud endpoint during one execution (MBs).

In our previous work,[7] we have shown that the MC hybrid applications have an efficiency trade-off between battery power consumption and network bandwidth usage. This exists because (1) performing computationally intensive tasks on a mobile device can be inefficient in terms of battery power consumption and (2) sending data to and receiving from the cloud can also be inefficient in terms of bandwidth usage cost and battery power consumption by the transmitter chip on board (ie, WiFi) inside the mobile device. To address this problem, we present a technique to find and apply the optimal configurations of an MC hybrid application. The optimal configurations are considered to be the ones in which the application has minimal battery power consumption and minimal network bandwidth usage. A *configuration* is a valid mapping of all distinct offloadable modules of an MC hybrid app to a mobile and the cloud server endpoints. We use a binary string to represent a configuration. Previously,[7] we presented initial steps toward establishing this technique. In this paper, we extend that work to consider method-level and hybrid-level configurations and provide a more in-depth analysis of its behavior and performance.

Furthermore, to evaluate this technique, we consider Android-based applications. We assume that an app is composed of a set of collaborative code units called modules (ie, classes or methods in Java). By using a simple MC hybrid application framework, we make the modules offloadable so that they can be executed both locally on a mobile device and remotely on the cloud server. In general, when designing an MC hybrid application, we are faced with a decision about which module of the app should be executed locally and which one remotely. As there exists an efficiency trade-off between the objectives, the set of Pareto-optimal configurations of an MC hybrid application can be achieved. We follow the following basic steps to achieve the Pareto-optimal configurations.

1. The first step is to create the initial set of configurations for an MC hybrid app, which is based on three different level of granularity of the modules of the app.
2. After the configurations are created, the efficiency of each of the configurations is then measured by doing offline profiling. We consider the efficiency of a configuration as the recorded measured power consumption and network usage after one complete run of the MC hybrid app for that configuration.
3. At the end, a final set of configurations is obtained that consists of (1) statistically significant and nondominated configurations in collapsible sets and (2) noncollapsible configurations.

These steps provide a data-driven approach to finding the Pareto-optimal configurations. With an MC hybrid app instrumentation, the modules of the app are executed on two different endpoints (mobile and the cloud), depending on a configuration. Measuring the efficiency of the configurations generates the data, which is used to derive the Pareto-optimal configurations of the MC hybrid app.

The rest of this paper is organized as follows. Section 2 presents the motivation and background. Section 3 discusses the required steps to achieve the efficient MC hybrid applications. Section 4 introduces the case studies. Section 5 explains the workflow we use to find the Pareto-optimal configurations. Section 6 explains the experiments conducted and their analysis. Section 7 concludes this paper, and the future work is discussed in Section 8.

2 | MOTIVATION AND BACKGROUND

Mobile devices (eg, smartphones and tablets) are frequently used these days. Their popularity has increased mainly because of their support for a wide range of applications such as image processing, video processing, games, and online social networks. With the recent advancement of technology in mobile devices, they are now also equipped with services like location awareness, context awareness, and the use of sensors (Gyro, accelerometer, etc). However, the software applications that are developed for mobile devices can have a disadvantage of achieving the same level of user experience as in desktop software systems due to limited resources (ie, battery life) available on mobile devices.[5] Backed by virtually unlimited resources, the applications developed for a mobile device can offload their computationally intensive tasks to a remote computing location (ie, a cloud, an edge, or a fog).[8] This can save the battery life of a mobile device while using the network bandwidth.

2.1 | Mobile cloud, edge, and fog computing

Cloud computing has gained a very high popularity by providing high performing, flexible, low cost, and on-demand computing services.[2,9] A mobile device that has Internet connectivity (ie, WiFi, 3G/4G) can access the cloud by using MC hybrid applications. However, due to a long distance in terms of network topology between a mobile device and the conventional cloud, MC hybrid applications can suffer from high latency. Alternatively, mobile edge computing (MEC) has recently emerged[10] and promises to provide low latency due to be in proximity of mobile devices. An edge device is local to the mobile devices (in terms of network topology) where data are generated and collected. However, MEC has limited computing capabilities compared to the conventional cloud computing.[11] Fog computing[12] is a virtualized platform that provides computing services between mobile devices and the cloud data centers, typically, but not exclusively located at the edge of the network. While fog computing minimizes latency and reduces the amount of data sent to the cloud, it poses security and privacy concerns.[13]

2.2 | Mobile applications partitioning

To integrate a stand-alone mobile application with the remote computing service (ie, the cloud), the source code of the app is first partitioned into offloadable modules (eg, classes or methods). The partitioning can be done either a priori (at development time) or a posteriori (at runtime). So that the most computationally intensive modules at the code level can be identified and offloaded for remote processing.[14,15] After the partitioning is done, a decision mechanism of the underlying MC application framework decides when and what modules to offload to the cloud.

2.3 | MC application frameworks

An MC application framework provides tools and Application Programming Interfaces (APIs) to integrate a mobile application with the cloud. The integration might be done through a middleware[16] that provides an abstraction between a mobile device and the cloud and controls every aspect of the communication in between. In our previous work,[7] we used a simple middleware-based framework, which was based on *Firebase*.* Using Firebase or another such kind of system as middleware comes with a number of problems that we had highlighted. They work in an event-driven fashion, which calls back to the thread that starts its on-event handler. For example, if the user interface (UI)/main thread is waiting to get the results from the cloud, the callback from the Firebase handler will be blocked due to the inherent characteristic of the

*https://firebase.google.com/

Android system. In this work, we use a socket-based framework instead. They work in a suspend-offload-receive-resume fashion. More details about our framework are provided in Section 5.1.

Cuervo et al[17] proposed *MAUI*, an MC hybrid app framework. MAUI combines the benefits of two approaches: it maximizes the potential for energy savings through method-level code offloading while minimizing the changes required to applications. The partitioning is done using method annotations (ie, @Remoteable) at the developmental time. At runtime, the offloadable methods are identified using the Microsoft.Net Reflection API. The MAUI solver decides whether a method can be offloaded to the cloud or kept executing on the mobile device. Chun et al[18] proposed *CloneCloud*, which offloads computationally intensive parts of the application code to a device clone operating in a computational cloud to enhance application performance. Unlike MAUI, CloneCloud operates at thread-level granularity. It uses a combination of static analysis and dynamic profiling to partition applications automatically at a method-level granularity while optimizing execution time and energy use for a target computation environment. As there are no annotations in the source code, so the partitioning mechanism is used offline, which aims to pick which parts of an application's execution to retain on the mobile device and which to migrate to the cloud. Kosta et al[19] proposed *ThinkAir*, which addresses two key issues that were lacking in MAUI and CloneCloud: parallelism of methods execution using multiple virtual machine images on cloud and adaptation of online method-level offloading.

Furthermore, the design of MC hybrid applications is normally based on achieving one or more than one particular objective(s) such as energy efficiency, bandwidth usage, and execution time. Khan et al[20] classified MC hybrid application frameworks into four broad categories: (1) performance-based, (2) energy-based, (3) constraint-based, and (4) multiobjective-based frameworks. Deng et al[21] and Guo et al[22] proposed code-offloading frameworks that are based on two objectives: minimizing power consumption and execution time of mobile applications. Naqvi et al[23] proposed a multiobjective optimization framework (*MAsCOT*), which uses probabilistic graphic models for a self-adaptive decision support for code offloading. Nakahara and Beder[24] proposed a bi-objective optimization framework (*CoSMOS*), which analyzes each optimization parameters (energy consumption and execution time) separately using cost function and self-adaptive reinforcement. Achieving two or more objectives at the same time might not be possible. For example, minimizing bandwidth usage may prevent the objective of minimizing energy consumption as the transceiver (RF) chip also uses power. This creates the efficiency trade-off between the two objectives.

2.4 | Search-based multiobjective optimization framework

In this paper, we consider minimizing battery power consumption and network bandwidth usage as two objectives. To achieve efficient MC hybrid applications, we presented previously a multiobjective optimization technique[7] to find and apply the optimal configurations of Android-based MC hybrid application. We evaluated the technique using offline profiling of Android-based MC hybrid applications but only with a class-level granularity. In this work, along with the class-level, we also consider the method-level and hybrid-level granularities of applications. Like the MAUI framework, we use Java annotations for partitioning mobile applications source code into multiple granularity levels. Using a custom workflow, we use an exhaustive search algorithm to find efficient configurations of MC hybrid applications. We also do more in-depth statistical analysis to refine the final configuration set. In the next section, we discuss the proposed method in more details.

3 | ACHIEVING EFFICIENT MC HYBRID ANDROID APPLICATIONS

In this section, we discuss our technique to achieve efficient MC hybrid Android applications. The computationally intensive modules (ie, classes or methods) of an app, when executed on a mobile device, consume battery power. We identify such modules at the code level and make them able for remote execution, using an MC application framework. The framework executes the app by taking a configuration. We create a range of different configurations for an app, which are based on (1) the number of modules in the app, (2) execution of modules across the two endpoint (ie, a mobile device and the cloud), and (3) granularity level of a module.

3.1 | Modular configurations

A modular configuration (or simply a configuration) is a binary string created for executing an MC hybrid app using an MC application framework. In a configuration, each binary digit of the string represents an offloadable module of the app.

The state of the digit tells the MC framework to execute the module either on a mobile device endpoint (0) or on the cloud server endpoint (1). The number at which a module executes during runtime is assigned its position (or index) in the configuration string. A module may execute multiple times (during runtime) but its position in the string is determined by the number at which it executes for the first time. Based on the application's code granularity, we can obtain different levels of configurations.

3.2 | Granularity of configurations

Granularity is the extent to which an app can be broken down into different modules. The computationally intensive tasks of the app are, therefore, divided into these modules. We identify these modules at the code level and make them offloadable by using the APIs of an MC hybrid application framework. The modules might be fine grained (ie, methods of classes) or coarse grained (ie, classes of an app).

To create a class-level (coarse grained) configuration, the offloadable modules treated as classes will be mapped to the binary string. For example, an arbitrary configuration, 0101, maps 4 classes of an app. The first and third modules with this configuration will be executed on the mobile device (0) and the second and fourth modules will be offloaded to the cloud (1), when the app executes using the MC framework. For $n = 4$, a set of class-level configuration can be obtained and its cardinality would be $2^n = 16$. Similarly, a method-level configuration (fine-grained) will map the methods of an app as the offloadable modules. For example, a configuration, 01000100010011, maps 14 methods to binary string.

A hybrid-level configuration has mixed granularity and will execute the offloadable modules in a mixed combination of selected methods and classes. A hybrid-level configuration can also be represented by a binary string such as 1010:01011010. Unlike the class- and method-level configurations, the binary string is composed of two parts that are separated by a colon sign (:). The digits residing on its left side (first part) represent the offloadable classes and on its right (second part) are a combination of both classes and methods. The state of a digit determines whether the corresponding class will be executed as a coarse grained (0) or as a fine grained (1). The state of a digit in the second part describes whether the corresponding offloadable module (a class or a method) will be executed on the mobile 0 or on the cloud 1. To make a hybrid-level configuration set, the cardinality of the set will depend on the total number of offloadable classes and methods of the app.

3.3 | Collapsible configurations

A collapsible configuration is defined as the one that can be collapsed into a same or a different granularity-level configuration. The collapsible configurations are identical in terms of executing the same offloadable modules on either a mobile device or remotely. For example, a hybrid-level configuration 1100:0001110 can be collapsed into (1) a hybrid-level configuration 0011:0111100, (2) a method-level configuration 0001111100, and (3) a class-level configuration 0110. As we can see in these four configurations, the mapped offloadable modules (whether coarse grained or fine grained) will execute on the same endpoint, no matter what the configuration level is. We find the collapsible configurations in the three configuration sets. This gives us different sets, each having the collapsible configurations. We make the collapsible sets in order to find nondominant configurations and their statistically significant configurations. We will discuss the nondominant configurations in Section 3.6 and the statistically significant configurations in Section 3.5.

3.4 | Efficiency of configurations

We consider the efficiency of a configuration as the recorded power consumption and network bandwidth usage of the configuration when it is applied for a single run of the app. It is measured at runtime when the MC hybrid framework is executing an MC hybrid app with a configuration. In the following sections, we discuss how it is measured.

3.4.1 | Battery power consumption

Mobile devices operates on a limited supply of power available from the battery; therefore, power consumption should be used carefully.[25] The offloadable modules of an MC hybrid app when execute on a mobile device use the device components for computing. The components draw power from the battery to operate. We assume that the computation power is P^C. Alternatively, the modules can be executed remotely on the cloud. As stated in the works of Lin et al[8] and Fernando et al,[26] an offloadable module executes on the cloud in three phases: (1) the *RF* sending phase, (2) the

cloud computing phase, and (3) the *RF* receiving phase. We assume that the power consumption due to communication (RF sending and RF receiving) is P^{RF}. The total power consumption, P^{T}, for a configuration can be as follows:

$$P^{T} = P^{C} + P^{RF}. \tag{1}$$

When no code offloading is used, the communication power (P^{RF}) will be zero and the total power consumption (P^{T}) will be equal to the total computing power (P^{C}).

3.4.2 | Network bandwidth usage and latency

The network bandwidth is used when an MC hybrid application is using code offloading. However, code offloading can be inefficient if the app relies too much on it. It will use the maximum network bandwidth as well as consuming the battery power (P^{RF}). The total network bandwidth usage for a configuration is recorded as total bytes transmitted (*Tx*) and received (*Rx*) between a mobile device and the cloud, when an MC hybrid app is executed.

The network latency is a critical measurement in code offloading. If the packets are dropping due to low signal strength or there is a congestion in the network, then the performance of the MC hybrid app will be degraded. As we are doing offline profiling to find efficient configurations of applications in a lab environment, the network latency in such case is noticed to be negligible.

3.5 | Finding statistically significant configurations

Finding statistically significant configurations is important because in the collapsible sets, the observed difference between the efficiency of any two configurations may not be significant. In other words, the efficiency of any two collapsible configurations happened to be different due to some uncontrolled variables. For example, the Android system was busy in scheduling a system-related task and the MC hybrid application took more computation time or the data packets were dropping due to a network congestions. As the uncontrolled variables cannot be avoided entirely so, due to their presence, to select efficient configurations, the hypothesis testing becomes essential. The result of a hypothesis test will give us statistically significant configurations in the collapsible sets.

3.6 | Selecting Pareto-efficient configurations

To achieve an efficient MC hybrid application, a filter needs to be applied to the collapsible configuration sets. The aim is to only pick the nondominated configuration(s) along with their statistically significant configuration(s), if present, in each of the collapsible sets. The rest of the configurations in all the collapsible sets are removed due to the fact that they are dominated and are not statistically significant.

A final set of configuration is created by combining all the collapsible sets along with the configurations, which were not collapsible. This set has a mix of configurations of the three granularity level. The nondominated configurations in the final set are the Pareto-efficient configurations. These configurations are superior to the others when both of the two objectives are considered.[27] In addition, these configurations optimize the efficiency trade-off and provide efficient alternatives to the MC hybrid app in terms of battery power consumption, network bandwidth usage, and latency.

4 | CASE STUDIES

We target two Android-based applications to apply the technique of achieving efficient MC hybrid application (discussed in Section 3). We first analyze the source code of the applications and find the computationally intensive tasks implemented in modules (classes and methods). To make these modules offloadable, we use custom-made Java annotation to put before the methods in classes that are composed of the computationally intensive tasks or are suitable for code offloading. The annotation is provided by the MC application framework that we use. We will discuss more about the framework in Section 5.1. When the MC framework executes the applications, the offloadable modules will execute either on a mobile device or on the cloud, depending on the current applied configuration. As discussed in Section 3.2, three different levels of configuration sets for the MC hybrid applications can be created. They are discussed in details in the following sections.

4.1 | ImageEffects

ImageEffects is a prototype application we created. The idea was to use a test app similar to Instagram that does the image processing and can store the images on a remote file storage server. ImageEffects has built-in image filters to apply to an image that is taken by using the device's camera inside the app. After the filter is applied to the image successfully, it can then be uploaded to the remote storage server. The mobile battery power is consumed when performing computationally intensive tasks, ie, image processing. The network bandwidth is used for uploading/downloading images to/from the remote storage server and using code offloading. Moreover, the battery power is consumed (1) by the mobile device's WiFi chip during the uploading/downloading process and (2) by the mobile device's LCD screen when displaying the application's UI.

The ImageEffects performs four distinct tasks. All the tasks are carried out in different modules of the app. A sequence diagram in Figure 1 shows all the tasks and how they are performed. The first task is to apply a filter to an image, which is to change each and every pixel of the original image. The second task is to generate a thumbnail of the filtered image, which is scaling down the image to a lower resolution. The thumbnail is then set to be viewed on the application's UI. The third task is to upload the image to the storage server and download it back. The fourth and last task is to calculate the hash codes of the original filtered image and the downloaded image and compare them to check that the downloaded image has not been changed or tampered. These tasks are executed in the background using the Android's AsyncTask library after an input from the user.

The ImageEffects has a total of four classes and ten methods in these classes, which are suitable for code offloading. The first and fourth classes have three methods each and the second and third classes have two methods each. For a total number of four class-level modules, $n = 4$, the cardinality of the class-level configuration set for ImageEffects is $2^n = 16$. For ten method-level modules, $n = 10$, the cardinality of the method-level configuration set is $2^n = 1\,024$. Lastly, the

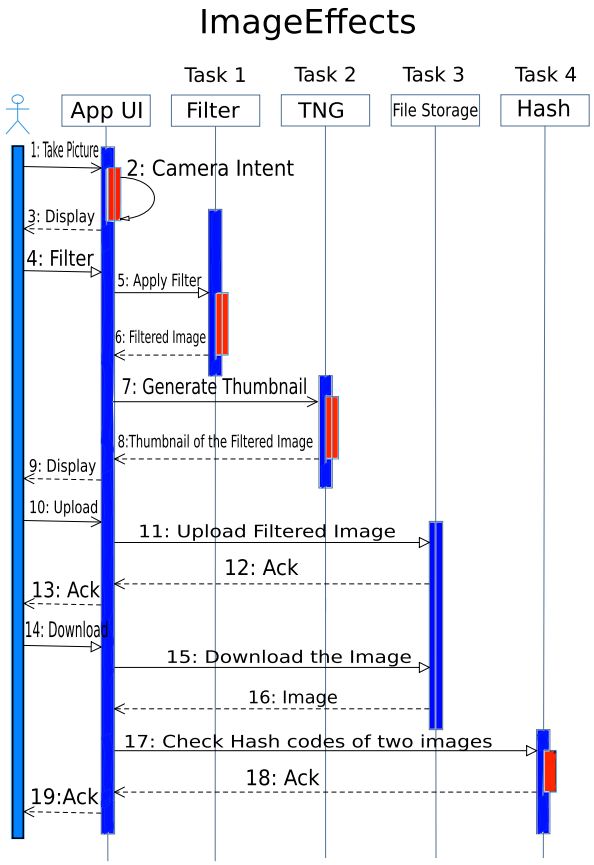

FIGURE 1 Sequence diagram of the ImageEffects mobile cloud hybrid application. It is a prototype and Instagram-like app that we developed. The four distinct tasks of ImageEffects are performed in different granularity-level modules, where each executes either on a mobile device or on the cloud depending on a configuration. UI, user interface

cardinality of the hybrid-level configuration set for the ImageEffects is 2560. We have created a small Java-based tool[28] that can create all the valid hybrid-level configurations for an MC hybrid app by providing the number of classes and methods.

The total number of valid and possible configurations (combining all granularity level) that can be created for ImageEffects are 3600. Some of these configurations are collapsible and will end up in the collapsible sets, as discussed in Section 3.3. For example, a class-level configuration 0000 is collapsible into a method level 0000000000, a hybrid level 0001:00000, another hybrid level 0111:00000000, and 14 other hybrid-level configurations. Similarly, a method-level configuration 0010110111 is only collapsible into a hybrid level 1110:001011011. Creating the collapsible sets helps us find the statistically significant and nondominated configurations in the sets. In Section 5, we will discuss how we filter these collapsible sets for the statistically significant and nondominated configurations. The filtered configurations will then end up in a final configuration set, as discussed in Section 3.6.

4.2 | Mather

Mather[†] is an open-source Android application, which is based on the *Math.js*[‡] library to perform expression-based computations. In addition to basic arithmetic, Mather also supports complex mathematical expression evaluation, user-defined functions, matrices, etc. When executed, Mather utilizes a mobile device's CPU for computation. The more complex the expression to evaluate, the more CPU utilization and, therefore, the more battery power consumption. To save the battery power, the task of mathematical evaluations can be executed on the cloud.

We analyzed the source code of Mather and found its two classes (ie, *MathParser.java and MathItem.java*), each having three methods, computationally intensive and suitable for code offloading. We also found that Android's WebView is used in one of the methods (*eval*()), which is run by the main UI thread. In the Android system, when the main thread is running, a request from the remote endpoint cannot be completed. For example, when *eval*() executes on a mobile device and calls another method that is to be executed on remotely according to the configuration. The result from the remote endpoint would not be handled as the main thread of the Android will still be running and in a waiting mode. This will cause the Mather in a *not responding* mode. Therefore, only one-third (75%) of the configurations for Mather are applicable.

For two classes ($n = 2$), the cardinality of the class-level configuration set for Mather is $2^n = 4$, in which only three are applicable. As in each of the classes, there are three offloadable methods. Therefore, for a total of six methods ($n = 6$), the cardinality of the method-level configuration set is $2^n = 64$. The applicable method-level configurations are only 48. Finally, for the two classes and six methods, we apply the valid 32 hybrid-level configurations. Some of the configurations are collapsible and will end up in the collapsible sets. For example, those configurations that run the modules only on a mobile device are collapsible (ie, 00,000000,01:0000,10:0000). Similarly, the following configurations will end up in a same collapsible set: 111000,01:1000, 10:1110,10.

Finally, we measure efficiency of the configurations for Mather and filter the statistically significant and nondominated configurations from the collapsible sets. The final configuration set is created at the end, in which the efficient configurations are searched for Mather. We will discuss how we do it in Section 5.

5 | THE WORKFLOW—TO FIND THE EFFICIENT CONFIGURATIONS

In this section, we describe how to set up a workflow to search for Pareto-optimal configurations for the two MC hybrid applications (ImageEffects and Mather). The workflow should follow the steps (discussed in Section 3) to achieve efficient MC hybrid Android applications. We will use the workflow to measure the efficiency of the configurations created for both MC hybrid apps. We will also discuss the MC application framework, which the MC hybrid apps will use to execute and effectively use the code offloading.

5.1 | MC application framework

An MC application framework executes MC hybrid applications on mobile devices and uses code offloading. The framework provides APIs for the code offloading, which are used to make the modules of mobile apps offloadable.

[†]https://github.com/icasdri/Mather
[‡]http://mathjs.org/index.html

For the two MC hybrid applications (ImageEffects and Mather), we implemented a simple MC application framework similar to the one used in the work of Flores and Srirama.[29] This framework provides an open-source code-offloading API, which is available on Github.[§] Using the API, we put an annotation (*@Cloud*) before the methods of the apps. A simple Java-based converter is provided, which reads the Java files, explores the classes of the apps, and transforms the annotated methods into offloadable modules. In addition, using this converter, two copies of the application are created: one for running on a mobile device (client) and second for remote execution (server). The API provides a socket-based communication between the client and the server.

The framework receives a configuration as a command-line argument, parse it and then executes the modules of the MC hybrid apps according to the digits sets in the configuration. Depending on the configuration level, the digits set as 1 indicate (1) all the offloadable methods of the classes will be executed on the cloud (class level), (2) the offloadable methods of the classes will be executed on the cloud (method level), and (3) the offloadable classes and methods of other classes will be executed on the cloud.

5.2 | Executing the MC hybrid applications with configurations

The MC hybrid apps (ImageEffects and Mather) are now able to use the code offloading after implementing the MC application framework. As discussed in Section 4, the total number of configurations created for ImageEffects and Mather is 3600 and 83, respectively. To find efficient configurations, executing the apps manually and repeatedly with all of these configurations would clearly be time consuming and practically not feasible. Therefore, an automatic process is required that executes the apps repeatedly and each time with a new configuration.

We wrote a Python-based script for the automatic process, which implements an exhaustive search algorithm shown in Algorithm 1. The script runs on a PC and executes the MC hybrid apps repeatedly and each time with a new configuration. It uses an open-source library, *AndroidViewClient*,[¶] to interact with the MC hybrid applications on a smartphone, connected to the PC via a USB cable. This library provides higher-level operations and the ability to obtain a tree of Android UI views present at any given moment on the device or emulator screen and performs operations on it. Alternatively, Android's own library *MonkeyRunner* can also be used for the same purpose. The script also records the execution time of each run of the MC hybrid app with a configuration.

Algorithm 1 Exhaustive search algorithm executes MC hybrid apps with configurations and measures efficiency of the configurations

```
1:  device ← connectToDevice()
2:  if device == null then
3:      return
4:  end if
5:  S ← Configuration Set
6:  count ← 1
7:  while count ≤ n do
8:      device.shell(Monitor App, S[count])
9:      device.shell(MC App, S[count])
10:     while true do
11:         app ← device.shell(ps | grip α)
12:         if app == null then
13:             break
14:         end if
15:     end while
16:     device.touch(x,y)                          ▷ Stop the Monitor app
17:     count ← count + 1
18: end while
```

[§] https://github.com/huberflores/CodeOffloadingAnnotations
[¶] https://github.com/dtmilano/AndroidViewClient/wiki

We do the offline profiling by running the script. The script is provided with all the configurations (S), for both MC hybrid apps. The exhaustive search algorithm iterates through the configuration sets, passes them to the apps, and executes them. During the runtime, the efficiency of the configurations is measured with the help of an instrumentation mobile app we created, *Monitor*. The script controls the Monitor app and the MC hybrid apps by sending commands to them. After an MC hybrid app completes one run, it destroys itself and the script finds it out inside the inner while loop. It then stops the profiler in the Monitor app by sending it the stopping command. The recorded measurements by the Monitor for the current configuration are received on the PC and stored in an SQL database. This process is repeated until all the configurations are applied and their measurements are recorded.

5.3 | Measuring the efficiency of configurations

As discussed previously (in Section 5.2), the efficiency of the configurations is recorded by the Monitor app during the execution time of the MC hybrid applications. When the profiler is stopped in the Monitor app by the automation script, a string having all the recorded measurements (power consumption, bandwidth usage, and execution time in seconds) for the current configuration is created. It is then sent to the PC over a socket connection, which is then parsed and the recorded values are stored in the SQL database.

For measuring power consumption of the MC hybrid app, we implemented an open-source tool, PowerTutor,[#] in the Monitor app. It is based on the component power management and activity state introspection.[30] To estimate the power consumption of the components, PowerTutor uses PowerBooter. It is an automated power model construction technique that uses built-in battery voltage sensors and knowledge of battery discharge behavior to monitor the power consumption while explicitly controlling the power management and activity states of individual components. These components include CPU, WiFi, 3G, GPS, LCD display, and audio interface. In addition, PowerTutor considers the energy consumption of each app to be independent. In other words, PowerTutor assumes that, ie, app A consumes the same amount of energy with or without app B running. In this way, power consumption is measured (based on statistics) for each component and for each UID/application. Implementing the PowerTutor, the Monitor gives us the total power consumption as stated in Equation 1. For recording the network bandwidth usage per UID in Android-based devices, we used the built-in Android library, *android.net.TrafficStats*. As the data transmitted Tx and received Rx are recorded against the UIDs of active (in running mode) applications, measurements for an MC hybrid app can be extracted by providing its UID to the Monitor app.

5.4 | Filtering the configurations

The aim of the filtering is to uncover a significant difference between different collapsible configurations of the MC hybrid apps when it actually exists. We do statistical tests to uncover the difference. As discussed in Section 3.3, we create collapsible sets of configurations. We measure the efficiency of configurations in these sets for n times, where n is a sample size. The sample size is important because larger sample sizes give us more confidence over any expected difference, given the noise in the system. The filtering process involves comparing the samples of any two configurations in the collapsible sets to uncover the difference. Moreover, there exist outliers of power or bandwidth usage values in the samples. Their presence leads to substantial distortions of parameter and statistic estimates when using statistical tests. Therefore, we first eliminate the outliers from the samples before using the statistical tests.

5.4.1 | Outliers elimination

We noticed during the manual analysis of the profiling data, received from the Monitor app, that, for some configurations, the power consumption values, of some samples, deviate markedly from others. Therefore, to eliminate such outliers, we do outliers tests. A number of outliers tests have been proposed.[31] Some of the more commonly used tests are the Grubbs' test, Tiejen-More test, and Generalized Extreme Studentized Deviate test. We wrote a Java-based tool that implements the Grubbs' test. It accesses all the n samples of the efficiency of the configurations and applies the Grubbs' test repeatedly until all the deviated power consumption values are removed. It then takes new measurements to replace the eliminated ones in the samples. Now that we have the efficiency of all the configurations recorded with no outliers, we will do the statistical tests. This will filter the statistically significant configurations in the collapsible sets.

[#] https://github.com/msg555/PowerTutor

5.4.2 | Hypothesis testing

As discussed in Section 3.5, we perform the hypothesis test to find the statistically significant configuration. A hypothesis cannot be proved but can only be accepted/rejected based on a statistical test result. Therefore, the test will either accept or reject a null hypothesis. If the probability value of the test for two configurations is extremely low, the null hypothesis is then rejected. This shows that the configurations are statistically significant.

We establish the null hypothesis by assuming that any two configurations in the collapsible sets have no real difference in terms of their recorded power consumption and bandwidth usage values. The difference in the means of power and bandwidth happened merely due to the uncontrolled variables. We noticed that the measured power consumption data of the samples of the configurations is not normally distributed. Therefore, nonparametric hypothesis tests are best to apply on such data. We created a Java-based tool that iterates through all the collapsible sets and applies Wilcoxon rank-sum test[32] (a nonparametric statistical hypothesis test) to the configurations in each set. The tool stores the statistically significant configurations from the sets. When the tests are completed, the tool then searches the nondominated configurations in the collapsible sets, as discussed in the following section.

5.5 | Nondominated configurations

This is the final step in our workflow, which is to search for the nondominated configurations in the collapsible sets. A collapsible set has one or more than one nondominated configuration(s). They are efficient than others in terms of the lowest power consumption or network bandwidth values, as discussed in Section 3.6. The Java-based tool (discussed in Section 5.4.2) finds them and stores them in a final configuration set. This set is composed of (1) statistically significant and nondominated configurations in collapsible sets and (2) noncollapsible configurations. The Pareto-efficient configurations for the MC hybrid apps (ImageEffects and Mather) are then searched in their final configuration sets, which we will discuss in the next section.

6 | OFFLINE PROFILING AND RESULTS

In this section, we explain the offline profiling of the two different MC hybrid apps, ie, ImageEffects and Mather, which were introduced in the case studies. We will use the workflow, discussed in Section 5, to conduct the experiments. By carrying out the offline profiling, we will be able to find out the efficient configurations of the MC hybrid apps using the MC application framework. At the end of this section, we discuss the obtained results from the profiling with analysis.

6.1 | Experimental setup

We created a lab environment to carry out the profiling, which was composed of the following entities: (1) two endpoints, one for execution the mobile application and one for the cloud application; (2) a PC that runs the automation script (discussed in Section 5.2), the image storing server program for ImageEffects (written in Java), a Java-based client that gets the measurements from the Monitor app, and store them in an SQL database; (3) a small office or home office network, which connects the PC and the two endpoints.

We used Motorola Moto G4 smartphone for the mobile endpoint. For the cloud endpoint, we set up an Android-x86 virtual environment on the PC. The MC hybrid apps and the Monitor app were installed on the endpoints. The smartphone was connected to the PC via a USB cable. We execute the automation script on the PC, which runs the exhaustive algorithm to find the efficiency of all the configurations for the MC hybrid apps. In order to get n samples of the configurations, we executed the script for n number of times.

6.2 | Understanding the MC application framework

We created a small Android-based MC hybrid app (*Demo*), which implements the framework. It consists of two Java classes, where each has two offloadable methods. Each method evaluates a different and randomly selected arithmetic expression. These expressions are (1) given a number x, method "A" finds factors of all the numbers from 1 to a x; (2) given a number x, method "B" finds the factorial of x; (3) given a number x, method "C" finds the multiplication of $x * x$ matrix, having randomly generated numbers from 1 to 100; and (4) given a number x, method "D" evaluates the

expression tan(tan(tan(x))). Based on the three levels of granularity (discussed in Section 3.2), a total of 36 configurations for the demo app were obtained.

We performed three tests with the Demo app. The computation level of the app was increased subsequently in each test, which was achieved by increasing the input value of parameter x in each method. In all the three tests, we obtained 100 samples of the configurations and then filtered the configurations to obtain the final set. The statistically significant and nondominated configurations in the collapsible sets and noncollapsible configurations for the Demo app, for all the three tests, are plotted on 2D graphs in Figure 2. We produced two different dot graphs (Figure 2A and Figure 2B) of the tests, in which the configurations of the Demo app before and after applying the filter are respectively shown. The data points on the graph are the configurations, where the x-axis value is the mean of the total power consumption and the y-axis value is the mean of the network bandwidth usage. We do the analysis of the results in the following section.

In each subsequent test, the computation level is increased, but the network bandwidth usage remained the same. After the offline profiling of the Demo application, we observed the following features from the results that reflect the behavior of the framework.

6.2.1 | All-zero and all-one configurations

These are the configurations with which an MC hybrid app executes all the offloadable modules either on a mobile device (all zero) or offloads (all one) to the cloud. In all-zero configurations, the battery power is consumed as the modules execute on a mobile device and there is no network bandwidth usage. The all-one configurations come with the cost of using the network bandwidth and consuming the battery power by the transmitter chip. We can see in Figure 2A, when the computation level of the Demo app was low (test-1), that the all-zero configurations were more efficient. As the computation level of the app was increasing (test-2 and test-3), the all-one configurations were turning to be more power efficient than all zeros with the cost of maximum bandwidth usage. The all-zero configurations can be seen in cluster-0 of test-1 and test-2 and cluster-1 of test-3. The all-one configurations can be spotted in cluster-4 of test-1 and test-2 and cluster-3 of test-3.

6.2.2 | Clusters of configurations

We can see in Figure 2A the configurations of the Demo app formed into different clusters after their efficiency was measured. The clusters are formed based on the amount of battery power consumption and network bandwidth usage. The configurations that have nearly the same network bandwidth usage formed into horizontal type clusters. These configurations have the same number of modules mapped as one, which is to use code offloading. As shown in Figure 2A, we can see four clusters of configurations in test-1 and test-2, which are numbered on the bases of number of modules using code offloading.

When the computation level was low (test-1), the configurations in the clusters were near to each others. When the computation level was increased (test-2), the distance between the configurations in the clusters also increased. This is due to the fact that the modules that were doing complex computation consumed more battery power and moved to the right of their clusters. This behavior is more prominent in test-2 (Figure 2A), where the spread of results of the configurations inside the clusters are more expanded.

When the computation level was maximum (test-3), the clusters of configurations become more prominent in vertical form rather than the horizontal form, as shown in Figure 2A. Those configurations that consumed more battery power are now in the right cluster (cluster-3). This is because the modules they mapped were more computationally intensive. As the third method of the demo app is $x * x$ matrix multiplication, which is the most computationally intensive method, therefore the configurations with the third digit set to zero landed into cluster-3. In cluster-1, the configurations that consumed less battery power are present. These configurations offloaded the computationally intensive modules to cloud. Therefore, they consumed less battery power with the cost of using more network bandwidth.

6.2.3 | Pareto-optimal configurations

As discussed in Section 5.5, the final configuration set for the demo app for all the three tests are created. This set contains the statistically significant and no-dominated configurations in the collapsible sets and the noncollapsible configurations. The final configuration set are plotted on the 2D graphs in Figure 2B. We can see that, when the computation level is low (test-1), running all the modules on the mobile device (all zero) with class-level configuration is the right choice. In other words, there is no need of using code offloading or executing the app with fine or hybrid-level granularity configurations

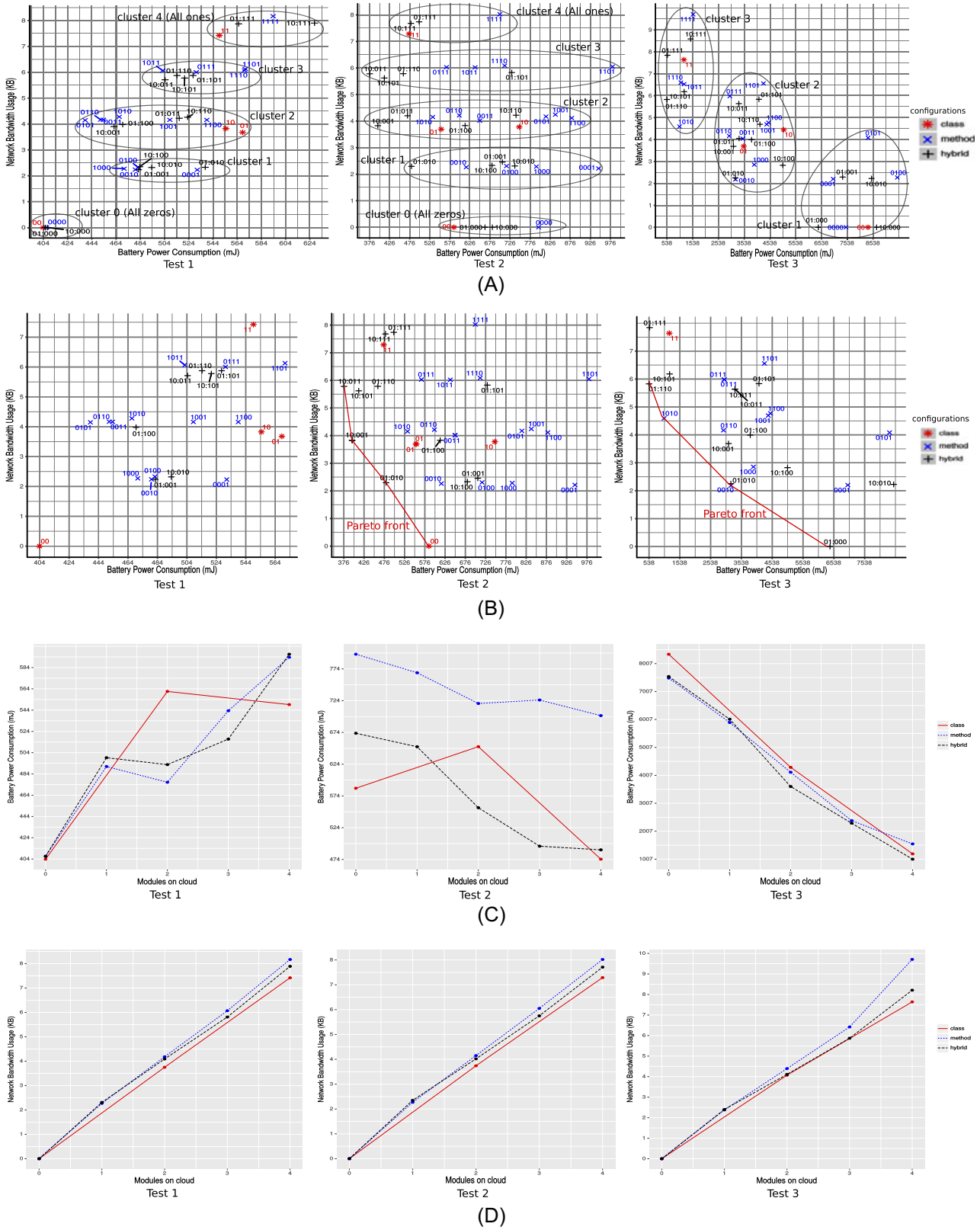

FIGURE 2 Plots showing the results obtained from three different profiling tests. The tests were carried out with a prototype Android-based application (demo) to understand the underlying mobile cloud application framework. The computation level of the app increases in each test subsequently. A, All configurations of the app forming in clusters. We can see that the configurations turning into vertical-shaped clusters from horizontal after increasing the computation level of the app; B, The filtered configurations having the Pareto-efficient configurations making the Pareto front. The plots in C and D show the power consumption and network bandwidth usage per number of modules running on the cloud, respectively

when the app is not doing too much computation. However, when the computation level increases (test-1 and test-2), we get a Pareto front (shown with a red line). The configurations forming the Pareto front are the Pareto-optimal configurations. These are efficient in terms of battery power consumption and network bandwidth usage. The efficient configuration in Test-1 and the Pareto-optimal configurations of test-2 and test-3 of the demo app are shown in Table 1. We can see a mixed level of granularities of the efficient configurations, which shows that multilevel granularity is important for achieving optimization in MC hybrid applications.

The mean execution time of the of demo app for the Pareto-optimal configurations is also stated as execution time in Table 1. We can see that, as the computation level increases (from test-1 to test-3), the execution time of the app with all-zero configurations increases. The execution time of the hybrid-level efficient configuration (01:110) in test-3 is the same as the class-level efficient configuration (00) in test-1. In other words, with a maximum computation level (test-3), offloading three out of four modules (01:110) took the same time as running all the four modules (00) on the mobile device and with less computation level (test-1). This shows that the performance of an MC hybrid app can be improved while using multilevel granularity for achieving optimization in MC hybrid applications.

6.2.4 | Power and bandwidth consumptions per modules running on the cloud

We observed the behavior of the Demo app executing the same number of modules on the cloud with different configurations. The mean power consumption and bandwidth usage per number of modules offloaded are shown in Figures 2C and 2D, respectively. We can see in Figure 2C, when the computation level was low (test-1), using code offloading was *energy inefficient*. The power consumption was increasing per number of modules using code offloading. But, as the computation level was increasing (test-2 and test-3), we can see that the code offloading was becoming *energy efficient*. The high number of modules executing on the cloud was consuming less battery power of the mobile device. This implies that running an MC hybrid application with all-zero configurations, when the computation level of the app is low, can result in less power consumption than running on all-one configurations. On the other hand, when the computation level of the app is high, executing it with all-one configurations can result in less power consumption than running on all-zero configurations.

The network bandwidth usage of the demo app, running different number of modules on the cloud, for different level of configurations is shown in line graph in Figure 2D. The bandwidth used by the modules of the app using code offloading was the same in the three tests. It is due to the fact that the input parameter "x" of the methods was an integer value. In addition, the result generated by the modules was an integer value. The small difference of the bandwidth usage, which can be seen in the graphs, is due to the network or platform overhead.

6.3 | Results for ImageEffects and Mather

In the previous section, we have discussed the behavior of the MC application framework using a demo app. This gave us insight to better understand how the two MC hybrid apps (ImageEffects and Mather) will result while executing them using the MC application framework. The total number of configurations for ImageEffects was 3040. For Mather, we had a total of 83 applicable configurations. We were able to get 30 and 100 samples of all the configurations of ImageEffects and Mather, respectively. In the following sections, we discuss the results for each of the apps separately.

6.3.1 | ImageEffects

The mean of power consumption and bandwidth usage of the configurations of ImageEffects are plotted on a 2D graph and shown in Figure 3A. We can see the configurations forming into two vertical-type clusters, which is similar to the behavior discussed in Section 6.2.2. The configurations that consumed more battery power are in the right-side cluster. The majority of these configurations are those that execute the modules composed of the image-processing task on the mobile device. We can also see that the all-zero configurations, which use no code offloading, are at the bottom of the graph. These configurations used only small amount of network bandwidth - only for uploading/downloading images to the image server. The all-one configurations are in the left-side cluster, except the method level (1111111111), as they remotely execute the modules and consumed less battery power. The all-one method-level configuration (1111111111) is near the top of the right-side cluster. This is due to the flip-flop pattern of data send and receive over the network for the maximum time (in case of 1111111111 is ten times), which use more battery power and network overhead. The flip-flop

TABLE 1 Pareto-efficient configurations of the mobile cloud (MC) hybrid applications obtained are listed. They are obtained as a result of offline profiling of the MC hybrid apps. The number of samples obtained of the configurations, and the efficiency of each configuration is stated. The configurations are of different granularities

MC Hybrid App	Configuration	Granularity Level	Samples	Efficiency of Configurations				
				Exec. Time (sec)	Battery Power Consumption (mJ)		Network Bandwidth Usage (KB)	
				Mean	Mean	Standard Deviation	Mean	Standard Deviation
Demo app (test-1)	00	class	100	2.99	279.645	70.7518	0	0
Demo app (test-2)	00	class	100	3.02	614.6129	98.5792	0	0
	01:010	hybrid	100	3.00	538.8065	83.726	2.2568	0.0871
	10:001	hybrid	100	3.00	378.0	74.2911	3.8861	0.0836
	10:011	hybrid	100	3.01	344.4839	74.5848	5.6719	0.0997
Demo app (test-3)	01:000	hybrid	100	9.35	5241.2903	1162.5894	0	0
	01:010	hybrid	100	5.99	2204.6774	510.2685	2.3155	0.0682
	1010	method	100	5.01	958.9355	158.1123	4.4542	0.0683
	01:110	hybrid	100	2.99	504.2258	89.6743	5.84	0.0803
ImageEffects	1010:00000000	hybrid	30	10.03	2666.685	587.5	133.0	0.8
	1000:010000	hybrid	30	8.13	1166.69	267.0	669.7	1.3
	1011:110000100	hybrid	30	8.02	1066.7	308.9	1065.7	1.4
	1000100000	method	30	8.09	967.569	279.3	1201.3	1.5
Mather	01:0000	hybrid	100	20.06	16756.871	3096.783	0	0
	001000	method	100	20.12	14737.9677	2970.083	3.072	0.026
	10:0110	hybrid	100	20.07	10895.452	2754.441	3.728	0.047

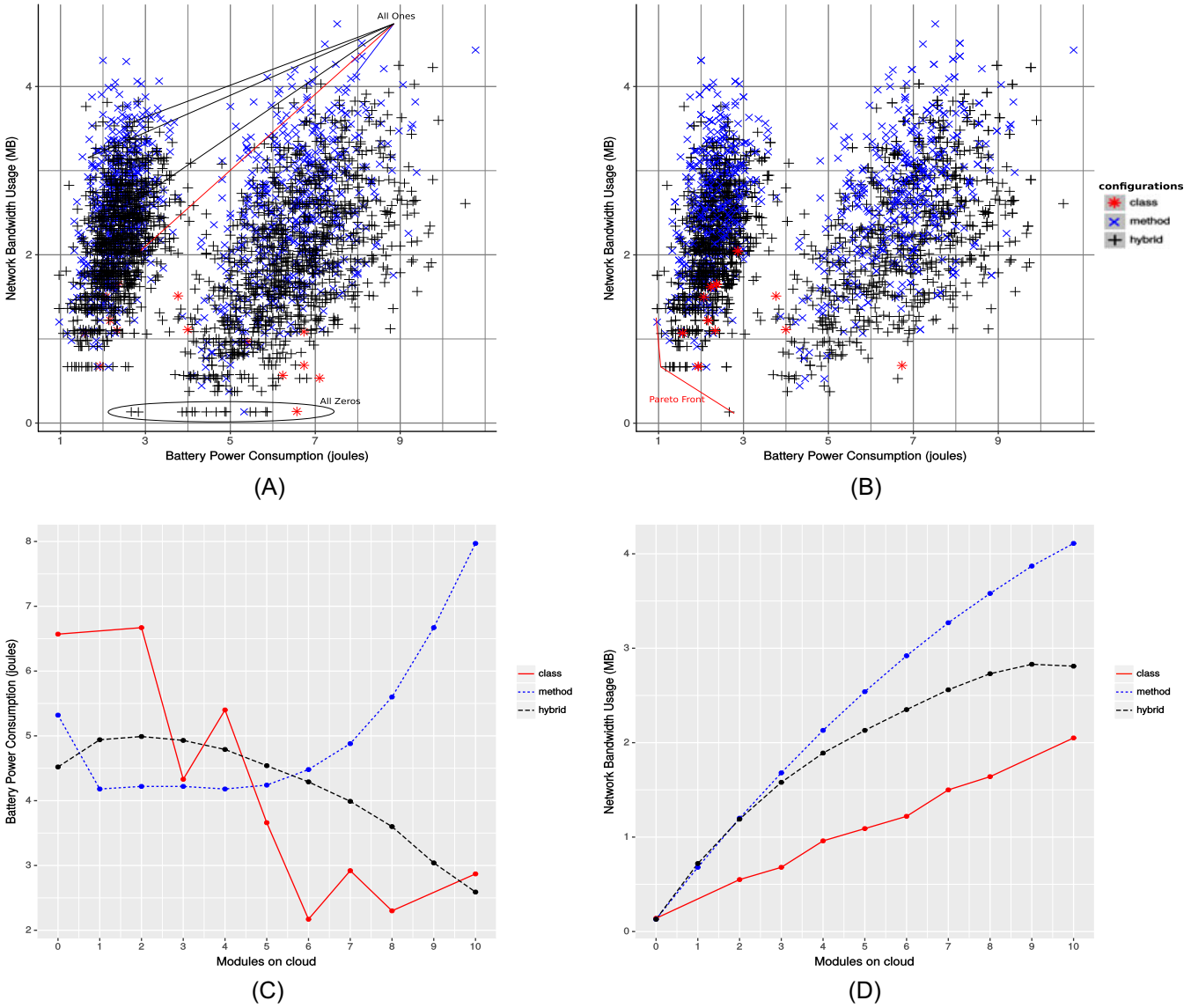

FIGURE 3 Plots showing configurations for ImageEffects mobile cloud hybrid app. A, All configurations of mixed granularity forming in two vertical-shaped clusters. The configurations running the computationally intensive modules of the app are in the right-side cluster; B, Filtered configurations in the collapsible sets along with the noncollapsible configurations. The Pareto-efficient configurations making the Pareto front are also shown; C, Battery power consumption; D, Network bandwidth usage per number of modules running on the cloud. Each line represents different granularity level of the configurations running the modules

pattern in all-one class-level configuration (1111) is the lowest (four times). Therefore, we can see it uses less network bandwidth than other all-one configurations.

The filtered configurations in the collapsible sets along with noncollapsible configurations are shown in Figure 3B. We can see the Pareto-optimal configurations forming the Pareto front. They are also listed in Table 1. These configurations are efficient in terms of minimum network bandwidth usage and battery power consumption. Moreover, we can see that they are of different granularity level (method level and hybrid level), which shows that multilevel granularity is important for optimization of MC hybrid apps.

Figures 3C and 3D show the power and bandwidth consumption of multiple granular configurations. We can see that running maximum number of modules on the cloud consumed less power and bandwidth for class and hybrid-level configurations. The method-level configurations, due to the flip-flop pattern, consumed more data and power when running maximum modules on the cloud. However, they are efficient in terms of battery power consumption when running between one and five modules on the cloud.

6.3.2 | Mather

The Mather evaluates mathematical expressions, as discussed in Section 4.2. For experimental purpose, we evaluate a $10 * 10$ matrix multiplication. The results obtained are plotted on graphs in Figure 4, where the all-zero and all-one configurations are also labeled. As in the demo app and ImageEffects, the all-zero configurations of Mather are also at the bottom as they do not use network bandwidth for code offloading. The filtered configurations in the collapsible sets along with the noncollapsible configurations are shown in Figure 4B. We can see the Pareto-optimal configurations form the Pareto front and these configurations are also stated in Table 1. They are of method and hybrid levels of granularities. The resultant shape of the Pareto front for Mather is a concave shape. While this is not typically expected for min-min problems, it is possible here because of the following.

1. For Mather, only 75% of the configurations were feasible, which means that the search space was restricted.
2. The two objective functions are not independent. This dependency is due to the fact that the transmitter also uses power when sending or receiving data. Therefore, when a configuration sends/receives more data, it also uses more power and will be shifted toward the upper-right of the graph. This can result in a concave Pareto front.

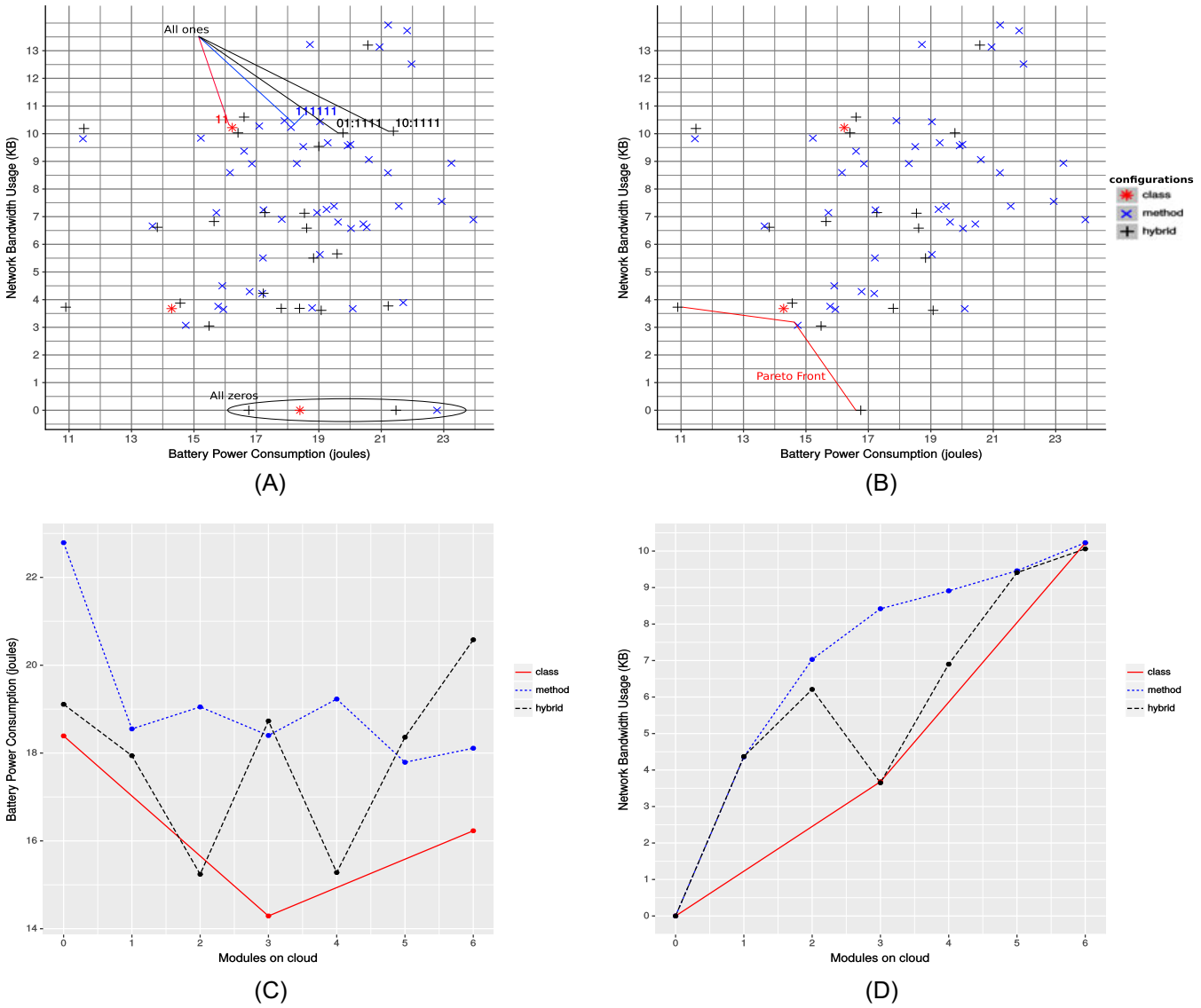

FIGURE 4 The feasible configurations for Mather mobile cloud hybrid app are plotted in these graphs. A, All configurations of mixed granularity for Mather; B, Filtered configurations in the collapsible sets along with the noncollapsible configurations; C, Battery power consumption per number of modules running on the cloud; D, Network bandwidth usage per number of modules running on the cloud

Figures 4C and 4D show the power consumption and bandwidth usage by number of modules offloaded to cloud with multiple granular configurations, respectively. Due to the fact that all the configurations in search space for Mather were not applicable, the power consumption forming a saw-tooth effect per number of modules on cloud.

7 | CONCLUSION

In this paper, we presented a technique for developing efficient MC hybrid applications by optimizing the battery power consumption and network bandwidth usage. Using this technique, an application's source code is modularized during development time using an annotation before methods. These methods are then converted into offloadable modules. The modules are then executed on either a mobile device or the cloud server depending on a configuration. A configuration is a binary string that represents the modules and their executing endpoints. Based on the three level of applications granularities, a configuration is of three types, ie, (1) class-level (coarse grained), (2) method-level (fine grained), and (3) hybrid of coarse and fine grained. By doing offline profiling and using a simple MC application framework, an MC hybrid application is executed repeatedly with its configurations. The battery power consumption and network bandwidth usage are then measured at execution time.

We conducted experiments by applying this technique on two different Android-based applications. ImageEffects is a prototype and Instagram-like app we created and Mather is an open-source app. The experimental results indicate that these applications produced Pareto-optimal configurations. These configurations are efficient in terms of minimizing the two objectives, ie, battery power consumption and network bandwidth usage. We assessed the results with a prototyped *Demo* app. We created this app to understand the behavior of the underlying MC framework. From the results, it was observed that a simple mobile application that performs less computation is efficient when executed with a class-level configuration that does not use code offloading. When an MC application performs more computationally intensive tasks, the code offloading becomes energy efficient. The efficient configurations in this case are the Pareto optimal and of mix level of granularities. This was true for both ImageEffects and Mather.

We also observed that the impact of using our technique is a reduction in battery power consumption with the cost of using network bandwidth. This is measured from the mean power and data of configurations shown in Table 1. The Pareto efficient and all-zero configuration for ImageEffects, which runs all the modules on the mobile device, are 1010:00000000. This configuration has the highest power consumption in all the Pareto-efficient configurations. The second Pareto-efficient configuration, which offload modules to the cloud but has the lowest power consumption, is 1000100000. The reduction of power, if the second configuration is used instead of using all the modules on a mobile device, is calculated to be 63.71% in joules. This power reduction has come with the cost of using 1.07 MB of network bandwidth when the second configuration is used. Similarly, for Mather, if the non–all-zeros configuration that has the lowest power consumption (10:0110) is used, the reduction in power is calculated to be 34.98% in joules with the cost of using 3.73 KB of network bandwidth.

8 | FUTURE WORK

In order to validate the idea of optimizing the battery power consumption and bandwidth usage in MC hybrid applications using multi granular configurations, we presented our technique and discussed the experimental results. We will now discuss the future directions.

- Scalability: the time it takes to exhaustively search the space of configurations depends on the number of modules. Thus, the larger the configuration set, the more time the exhaustive search will take. An estimated time for more than 20 modules could be more than a month. Therefore, in future work, evolutionary algorithms such as genetic algorithms will be used instead, which take less time to converge, albeit to approximate optimal solutions.
- Two-step searching: as in the current work, the exhaustive search algorithm searches for each and every configuration of any level of granularity, which takes long time. In the future work, a more intelligent search algorithm can be implemented. The idea is to first find the Pareto-efficient configurations in the class-level granularity, where the search space is comparatively small. Based on only these good configurations, their collapsible method and hybrid-level configurations will then be searched for efficient configurations.

- Self-adaptive and self-aware MC application framework: using a simple MC application framework that is capable of using code offloading can benefit offline profiling but its capabilities are not enough to use for online profiling. The future work will consider self-adaptive and self-aware MC application framework, which will be using switching at runtime between the Pareto-optimal configurations of an MC hybrid app. The switching thresholds will be based on packet losses caused by interference and network congestions. The objective to use such a framework is to minimize network latency along with power and bandwidth consumption at runtime.

ACKNOWLEDGEMENTS

This work would not have been possible without the financial support of the Abdul Wali Khan University Mardan (AWKUM) in Pakistan. We would also like to show our gratitude to our colleague, Dr Elizabeth Wanner, for sharing her insights during the course of this research. Finally, we are very thankful to all the researchers in the ALICE research group with whom we discussed our work and considered their insights and comments.

ORCID

Aamir Akbar http://orcid.org/0000-0002-9421-7379

REFERENCES

1. Balan R, Flinn J, Satyanarayanan M, Sinnamohideen S, Yang H-I. The case for cyber foraging. In: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop; 2002; Saint-Emilion, France.
2. Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. *Commun ACM*. 2010;53(4):50-58.
3. Huang D. Mobile cloud computing. *IEEE COMSOC Multimed Commun Tech Comm (MMTC) E-Lett*. 2011;6(10):27-31.
4. Flores H, Srirama SN, Paniagua C. Towards mobile cloud applications: offloading resource-intensive tasks to hybrid clouds. *Int J Pervasive Comput Commun*. 2012;8(4):344-367.
5. Ahmed E, Gani A, Sookhak M, Hamid SHA, Xia F. Application optimization in mobile cloud computing. *J Netw Comput Appl*. 2015;52(C):52-68.
6. Deb K. Multi-objective optimisation using evolutionary algorithms: an introduction. In: *Multi-Objective Evolutionary Optimisation for Product Design and Manufacturing*. London, UK: Springer-Verlag London Limited ; 2011:3-34.
7. Akbar A, Lewis PR. Towards the optimization of power and bandwidth consumption in mobile-cloud hybrid applications. In: Proceedings of the 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC); 2017; Valencia, Spain.
8. Lin X, Wang Y, Xie Q, Pedram M. Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment. *IEEE Trans Serv Comput*. 2015;8(2):175-186.
9. Buyya R. Market-oriented cloud computing: vision, hype, and reality of delivering computing as the 5th utility. In: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID); 2009; Shanghai, China.
10. ETSI Group. *Mobile-Edge Computing*. Introductory Technical White Paper. Sophia Antipolis, France: ETSI; 2014.
11. Mach P, Becvar Z. Mobile edge computing: a survey on architecture and computation offloading. *IEEE Commun Surv Tutor*. 2017;19(3):1628-1656.
12. Bonomi F, Milito R, Zhu J, Addepalli S. Fog computing and its role in the Internet of Things. In: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing; 2012; Helsinki, Finland.
13. Stojmenovic I, Wen S. The fog computing paradigm: scenarios and security issues. Paper presented at: 2014 Federated Conference on Computer Science and Information Systems; 2014; Warsaw, Poland.
14. Gu X, Messer A, Greenberg I, Milojicic D, Nahrstedt K. Adaptive offloading for pervasive computing. *IEEE Pervasive Comput*. 2004;3(3):66-73.
15. Li Z, Wang C, Xu R. Computation offloading to save energy on handheld devices: a partition scheme. In: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems; 2001; Atlanta, GA.
16. Flores H, Srirama SN, Paniagua C. A generic middleware framework for handling process intensive hybrid cloud services from mobiles. In: Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia; 2011; Ho Chi Minh City, Vietnam.
17. Cuervo E, Balasubramanian A, Cho D-k, et al. MAUI: making smartphones last longer with code offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services; 2010; San Francisco, CA.
18. Chun B-G, Ihm S, Maniatis P, Naik M, Patti A. CloneCloud: elastic execution between mobile device and cloud. In: Proceedings of the Sixth Conference on Computer Systems; 2011; Salzburg, Austria.
19. Kosta S, Aucinas A, Hui P, Mortier R, Zhang X. Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: 2012 Proceedings IEEE INFOCOM; 2012; Orlando, FL.
20. Khan AR, Othman M, Madani SA, Khan SU. A survey of mobile cloud computing application models. *IEEE Commun Surv Tutor*. 2014;16(1):393-413.

21. Deng S, Huang L, Taheri J, Zomaya AY. Computation offloading for service workflow in mobile cloud computing. *IEEE Trans Parallel Distributed Syst*. 2015;26(12):3317-3329.
22. Guo S, Xiao B, Yang Y, Yang Y. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. Paper presented at: IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications; 2016; San Francisco, CA.
23. Naqvi NZ, Devlieghere J, Preuveneers D, Berbers Y. MAsCOT: self-adaptive opportunistic offloading for cloud-enabled smart mobile applications with probabilistic graphical models at runtime. Paper presented at: 2016 49th Hawaii International Conference on System Sciences (HICSS); 2016; Koloa, HI.
24. Nakahara FA, Beder DM. A context-aware and self-adaptive offloading decision support model for mobile cloud computing system. *J Ambient Intell Humaniz Comput*. 2018; 9(5):1561-1572.
25. Lorch J, Smith A. Software strategies for portable computer energy management. *IEEE Pers Commun*. 1998;5(3):60-73.
26. Fernando N, Loke SW, Rahayu W. Mobile cloud computing: a survey. *Futur Gener Comput Syst*. 2013;29(1):84-106. http://www.sciencedirect.com/science/article/pii/S0167739X12001318
27. Srinivas N, Deb K. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evol Comput*. 1994;2(3):221-248.
28. A java-based tool developed to generate hybrid-level configurations for an MC hybrid app. https://github.com/aamirakbar/Creating-Hybrid-Level-Configurations-for-MC-hybrid-Apps
29. Flores H, Srirama S. Adaptive code offloading for mobile cloud applications: exploiting fuzzy sets and evidence-based learning. In: Proceeding of the 4th ACM Workshop on Mobile cloud Computing and Services (MobiSys); 2013; Taipei, Taiwan.
30. Zhang L, Tiwana B, Qian Z, et al. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS); 2010; Scottsdale, AZ.
31. Barnett V, Lewis T. *Outliers in Statistical Data*. 3rd ed. Chichester, UK: John Wiley & Sons; 1994.
32. Haynes W. Wilcoxon Rank Sum Test. In: *Encyclopedia of Systems Biology*. New York, NY: Springer New York; 2013;2354-2355. https://doi.org/10.1007/978-1-4419-9863-7_1185

How to cite this article: Akbar A, Lewis PR. The importance of granularity in multiobjective optimization of mobile cloud hybrid applications. *Trans Emerging Tel Tech*. 2018;e3526. https://doi.org/10.1002/ett.3526