IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 16 June 2023, accepted 1 July 2023, date of publication 6 July 2023, date of current version 12 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3292887

RESEARCH ARTICLE

An Empirical Study of Mobile Code Offloading in Unpredictable Environments

SANABRIA PABLO[1,2], NEYEM ANDRES[1,2], SANDOVAL ALCOCER JUAN PABLO[1], AND FERNANDEZ BLANCO ALISON[1]

[1]Computer Science Department, Pontificia Universidad Catolica de Chile, Santiago 7820436, Chile
[2]Centro Nacional de Inteligencia Artificial CENIA, Santiago 7820436, Chile

Corresponding author: Neyem Andres (aneyem@uc.cl)

This work was supported in part by the National Agency for Research and Development (ANID)/Scholarship Program/DOCTORADO NACIONAL under Grant 2020-21200979; and in part by the National Center for Artificial Intelligence (CENIA), Basal ANID, under Grant FB210017. The work of Sandoval Alcocer Juan Pablo was supported by ANID FONDECYT Iniciación Folio under Grant 11220885.

ABSTRACT Mobile code offloading is a well-known technique for enhancing the capabilities of mobile platforms by transparently leveraging the resources to the cloud. Although this technique has been studied for years, little empirical evidence exists to demonstrate its alleged benefits in terms of performance in real-life situations. All studies conducted on this topic have so far been relegated to controlled environments in laboratory settings. As such, there is no evidence of how and how well this technique performs in real-life scenarios, where network unreliability is the norm. In this work, we present the first empirical study of an Android mobile application integrated with a code offloading framework being tested in the wild. We distributed an application that contains a set of benchmarks in APK format and deployed it on a wide gamut of Android devices to which we had no physical access. We carefully detail the methodology and infrastructure we used to monitor the benchmarks' performance of 18 devices. Overall, our results show that the accuracy of the decision-making engine is heavily affected by a couple of factors, mainly the network diagnosis and connection type. Therefore, determining whether or not it is more convenient to execute a given task in the cloud is a difficult task. We summarize five lessons we learned by performing our experiment that we believe should be considered for future experiments in this area.

INDEX TERMS Android, empirical studies, mobile cloud computing, mobile code offloading.

I. INTRODUCTION

Mobile code offloading refers to the computational paradigm in which expensive tasks in terms of memory and computing power are transparently migrated from resource-constrained mobile devices to powerful servers in the cloud [1], [2]. The increased computational resources of the server counterpart are then expected to complete said tasks at a fraction of the time, therefore improving the end-user perceived performance of the application and reducing the client device's battery consumption. This latest characteristic is of particular interest to current manufacturers, as modern smartphone and tablet devices struggle to keep up with the ever-increasing power requirements of more and more complex applications [3]. This is a fact recognized by industry leaders Google and Apple, as some of their innovations introduced in their latest OS updates specifically target optimizing power consumption (i.e., background task restrictions). Furthering this trend, mobile code offloading offers the opportunity to keep improving battery lifetime by harnessing the power of the cloud. Nevertheless, despite promising results published in academia, we have yet to see a code offloading solution being used in the industry. The reason for this contradiction is actually rather simple: current mobile code offloading solutions have yet to address the nonfunctional requirements necessary for powering applications in production. Therefore, although we can read about the results of mobile code offloading platforms in laboratory environments under controlled settings, literature regarding such platforms being experimented on in real-life environments

The associate editor coordinating the review of this manuscript and approving it for publication was Ding Xu.

is, as far as we know, non-existent. More importantly, the lack of results in a practical setting raises the question of whether or not these technologies even work in production in the first place. If we are to advance the development of this discipline, it is imperative to move into the field. Therefore, the newer mobile code offloading frameworks should begin by demonstrating that they offer analogous results in real-life conditions like those claimed under controlled settings. Additionally, a reference point should be established to evaluate the effectiveness of said solutions.

Seeking to address these issues, we have developed MobiCOP [4], [5], a mobile code offloading framework compatible with the Android operating system that was built from the ground-up to satisfy three non-functional requirements we consider to be essential:

- *Reliability:* it can work under unreliable network connections, which are very common in mobile scenarios.
- *Scalability:* it can automatically adjust the number of server instances deployed in the cloud to attend a variable amount of traffic.
- *Distribution:* it offers a client fully contained in a library, which can therefore be imported into any existing Android project by adding a single Gradle dependency.

The fundamental architecture and implementation details of MobiCOP have been extensively explained in some of our previous publications. However, in our original paper, we also limited our experimentation to standardized benchmarks in controlled environments. This left the question open on whether any of our previous claims were actually applicable to production environments. As a matter of fact, we noticed that the move into the field required substantial changes to our original proposal in order to deal with various unforeseen situations.

The main goal of this paper is to describe our experience when deploying an existing mobile code offloading solution in the wild, the considerations that need to be addressed by developers of such solutions, and to establish the first baseline to which other offloading frameworks may compare themselves against. Here, we describe our experimental setup, the relevant metrics we collected during our experiments, and their subsequent analysis. For completeness, we will briefly cover the implementation details of the MobiCOP framework. However, we would like to stress that this is not the focus of this work. Instead, we will only cover a general overview of the platform and the changes applied to the newest version of MobiCOP that stemmed from our field experimentation. For more detailed information concerning our platform, we would like to refer readers to our original publication [5].

This paper presents a systematic approach to evaluate a mobile code offloading solution in real-life scenarios. Specifically, we conducted an experiment in which 18 volunteers, each with a mobile device, installed a benchmarking app connected to a prepared cloud infrastructure equipped with MobiCOP. We carried out the study in different locations across four countries and used a wide variety of mobile devices types. Our results show that the accuracy of the decision-making engine of a code offloading platform is heavily affected by a couple of factors, mainly the network diagnosis and connection type. Therefore, determining whether or not it is more convenient to execute a given task in the cloud is a difficult task. We believe our results will guide the development and experimentation of the next generation of mobile code offload frameworks.

In summary, we make the following contributions:

- An analysis and comparison of current mobile code offloading frameworks.
- A practical approach for evaluating an offloading solution in real-life scenarios.
- Five lessons learned, challenges, and research areas that are worth exploring in the field.

The paper is organized as follows. In Section II, we review the related work and focus on the common pitfalls present in the most prominent mobile code offloading solutions. In Section III, we briefly describe MobiCOP's architecture, how it differs from other solutions, and the modifications introduced in its latest version. In IV, we describe our experimental setup and the results we obtained. Next, in VI, we go over the lessons we learned from our experience; and finally, VII offers the conclusions and future work.

II. RELATED WORK

MAUI is one of the first mobile code offloading frameworks meant for modern smartphone devices that appeared in the academic literature [6]. It defines the basic components a mobile offloading architecture should contain, including a profiler for network and code, a decision-making engine, and a communication layer. MAUI operates at a method level on .NET platforms. It requires developers to annotate which methods in an application may be offloaded; these are later instrumented by MAUI to incorporate the offloading logic. MAUI incorporates various profilers to decide whether to offload a method or not. They collect information on a method's CPU cycles, the data that is needed to be sent back and forth to the server, and the state of the network. This information is later fed to a server-based optimizer which makes the final offloading decision. No fine-grained information is available on the specifics of how this optimizer operates.

CloneCloud is another mobile code offloading framework built for the Android operating system that introduces automatic migration of arbitrary code without developer input [7]. CloneCloud features a static analyzer that computes all reasonable partition points in a program within certain constraints. Additionally, profilers integrated into a customized OS version collect all the data required for a local decision-making engine to decide whether to offload or not. The specific criteria by which the decision to offload is made is unavailable. Execution on the server is attained through the usage of application-level virtual machines. To offload code, CloneCloud uses a thread suspend-wait-resume mechanism,

in which threads are halted, their state is transferred to the server and finally synced back into the client once the server finishes its execution. Unfortunately, using a customized Android version makes it very difficult to distribute this solution to interested end-users.

After CloneCloud came to Comet, also a code offloading framework for Android, that introduced a new method for achieving transparent code migration through the usage of distributed shared memory (DSM)) [8]. Comet features offloading capabilities at the thread level by allowing threads running locally and on the server to share state through various VM synchronization primitives. These primitives were custom-built by the authors and introduced into a branch of the Android-based CyanogenMod. Java's robust memory model is leveraged to achieve memory consistency between both contexts. Comet however suffers from a strong dependency on fast and reliable networks, as its synchronization mechanism requires a substantial amount of data to be transferred between a client and a server, though extensions to Comet's offloading model have been suggested to partially address this issue [9].

Various additional mobile code offloading frameworks have been built on top of Comet's offloading architecture and communications layer, including EMCO [10], a framework that strengthens the accuracy of the decision-making engine by incorporating rules inferred from crowdsourcing; and Tango [11], a framework that disposes of the need for a decision-making engine by always running code simultaneously, both remotely and locally, and preserving the result of the faster agent through a process called flip-flop replication. Unfortunately, the lack of the synchronization primitives required by Comet to function on the standard Android SDK makes it impossible to reproduce any of these solutions on stock mobile devices.

ThinkAir is one of the first mobile code offloading frameworks to recognize the scalability issues of previous offloading frameworks [12]. As such, it supports on-demand cloud resource allocation by spawning either additional or more powerful machines according to current traffic and the specific needs of the client respectively. ThinkAir operates by creating virtual machines containing the entire state of the client. Therefore, the network bandwidth requirements are very high. ThinkAir collects various parameters of the client in order to decide when to offload a certain piece of code, including CPU usage, screen brightness, network interface, network quality, overall method execution time, CPU cycles, number of method calls, memory allocation, and garbage collector count. Unfortunately, the specifics of how this information is used to reach a decision are unknown.

Jade is a mobile code offloading framework for devices connected through an ad-hoc mobile network [13]. It leverages the power of clusters of idle devices to assist a host device in processing various tasks. Jade's main priority is minimizing energy consumption on the client device, therefore increasing battery life. Its decision-making engine translates the offloading problem to an integer linear programming optimization problem that contemplates estimates of power consumption using an energy model and estimates of the total execution time of the task on the client and the server. It then uses a solver to decide whether to offload a given task or not. Since Jade was built with ad-hoc networks in mind, it supports both Wi-Fi and Bluetooth communication. Unfortunately, the problem with ad-hoc offloading solutions, in general, is that there is little incentive for idle devices to enter the network.

Finally, we would like to mention COARA and RAPID, the two most recent mobile code offloading frameworks we have been able to find in the literature. COARA [14] is a technology that leverages AspectJ to implement its offloading mechanism and introduces the concept of lazy and pipelined transmission, wherein some input objects are replaced by lightweight proxies so remote execution can begin faster. Data may then be fetched lazily by the server (lazy transmission) or accessed as it is being sent by the client (pipelined transmission). On the other hand, RAPID [15] is a code offloading solution for Android and Linux devices that supports CPU and GPGPU offloading. Its CPU code offloading stack is based on ThinkAir, while its server-side GPGPU code execution mechanism is based on a split-driver model [16] that allows sharing the same GPU hardware with several VMs simultaneously. While both of these platforms offer various interesting innovations, they do not address the issue of the decision-making engine at all.

Mobile code offloading has come a long way since the days of cyber foraging [17], yet despite the various breakthroughs we have seen over the years, there are many pitfalls that have remained unattended all this time. For one, mobile code offloading frameworks seem to be overly reliant on synchronous communication through sockets that need to be alive throughout the entire execution of a program. This requirement is unrealistic, as mobile data plans are still very expensive in various parts of the world and mobile networks are known to be unreliable. Additionally, the need to scale has been barely looked into, even though scalability is an active topic of research in the field of mobile cloud computing [18], [19]. With the exception of ThinkAir and its extensions, we have yet to see another mobile code offloading framework that tackles this issue. Moreover, we also noticed that several of these solutions are extremely hard to replicate. The need to replace the entire OS by a custom version is particularly egregious as few end-users would be willing to flash their devices (assuming they even have the technical know-how to do it). With these issues in mind, it is unlikely for mobile code offloading frameworks to be adopted by the public.

Another common problem we noticed lies in the implementation of the decision engines. Although most offloading frameworks define some sort of mechanism to decide on runtime if offloading is advisable, the details of most implementations are very vague. Mostly, we see partial descriptions of such engines in the literature in which the parameters

TABLE 1. Comparison of mobile code offloading frameworks.

	Non-Functional Requirements			Decision Engine		Experimentation	
Framework	Scalability	Reliability	Distribution Difficulty	Implementation Details	Evaluation Available	Controlled Environment	Real-Life Environment
MAUI	NO	NO	Average	Partial	NO	YES	NO
Comet	NO	NO	High	Partial	NO	YES	NO
CloneCloud	NO	NO	High	YES	NO	YES	NO
ThinkAir	YES	NO	High	Partial	NO	YES	NO
EMCO	NO	NO	Low	Partial	NO	YES	NO
Jade	NO	NO	Low	YES	NO	YES	NO
Tango	NO	NO	High	Does not apply	-	YES	NO
COARA	NO	NO	Average	No details provided	-	YES	NO
RAPID	YES	NO	High	No details provided	-	YES	NO
MobiCOP	YES	YES	Low	YES	YES	YES	YES

which are collected are enumerated but no specific description is given on how these parameters are used to reach a specific decision. Most importantly, despite some very interesting proposals [20], [21], [22], [23], [24], [25], we have yet to see practical evaluations of any given decision-making engine algorithm running on the devices of random end-users. As such, there is very little evidence that any of the currently proposed algorithms for these effects is actually useful or if there is any insight on how they may be improved upon. The only notable exception to the above we found is CADA [26], and offloading decision algorithm that was integrated with the ThinkAir framework and tested by volunteers who were asked to use a benchmarking application integrated with said algorithm whenever they had the chance within a 3-day window. However, the evaluation of CADA was conducted on a single Android smartphone model that had to be previously configured by the researchers, therefore its effects on the average device held by actual end-users remain unknown. A comparison of the various frameworks presented in this section is available in Table 1.

III. THE MobiCOP PLATFORM

MobiCOP is a fully functional mobile code offloading framework that consists of a client library and a server component. The server component consists of a set of Android execution environments in charge of running the offloaded code, and a middleware responsible for routing the messages between client and server. Genymotion Cloud for Android 8.0 is currently used for running Android code on the server, while the middleware was built using the Node.js based Express.js framework. Our Genymotion Cloud instances are hooked with Amazon Web Services AutoScaling technology to enable scalability through automatic instantiation of additional machines when the overall CPU usage exceeds a certain threshold. This module can also destroy spare instances once traffic diminishes in order to reduce maintenance costs for the developer.

The client component, on the other hand, is distributed as a standard *.aar* Android library. It is fully standalone, and it allows developers to integrate MobiCOP into any Android project in a matter of minutes. One of the main benefits of this approach is that, unlike several alternatives proposed in the literature, no customization of the standard operating system is required. The library is compatible with Android API level 17 and above and also includes support for Android's newest background restrictions introduced in Oreo. The client contains four components: a decision-making engine that tells the platform whether to offload a task or not, a network profiler that continually samples the quality of the network at reasonable intervals, a communication layer in charge of communicating with the server and serializing input and output data, and an executor in charge of running the task locally if offloading is not recommended.

A. ARCHITECTURE

MobiCOP's architecture was built with simplicity, reliability, and scalability in mind. It was also built to fit nicely with Google's official Android design patterns and philosophy. MobiCOP exposes a simple API that allows developers to encapsulate tasks that are good candidates for being offloaded in isolated components we call ''CloudRunnables'', akin to Android's Services. Services are components defined by the Android ecosystem that are well-suited for running background tasks. As per the standard in Android development, developers are required to manually pass input parameters in the form of sets of primitives between Activities and our CloudRunnables. These primitives are then taken into consideration by our decision-making engine to estimate how long it would take to run the given task either locally or in the cloud. An overview of MobiCOP's architecture is available in Figure 1.

MobiCOP's communication layer was heavily inspired by the FTP protocol. It defines two different channels: one for dispatching control messages to the server to start a remote execution, and a second optimized for transferring large amounts of input and output data. Control messages are lightweight and can usually be contained in a single network package; input and output data on the other hand can be very large (i.e., in multimedia applications, whole videos in the order of dozens of MB may need to be transferred back and forth between client and server). As such, this channel was built on top of resumable transfer technologies, so in case of a network interruption, data transfer can be retried from the point where the connection was lost. If the result of an offloaded task fails to reach back the client before the remote execution time estimated by the decision-making engine has elapsed, the framework falls back to running the task locally.

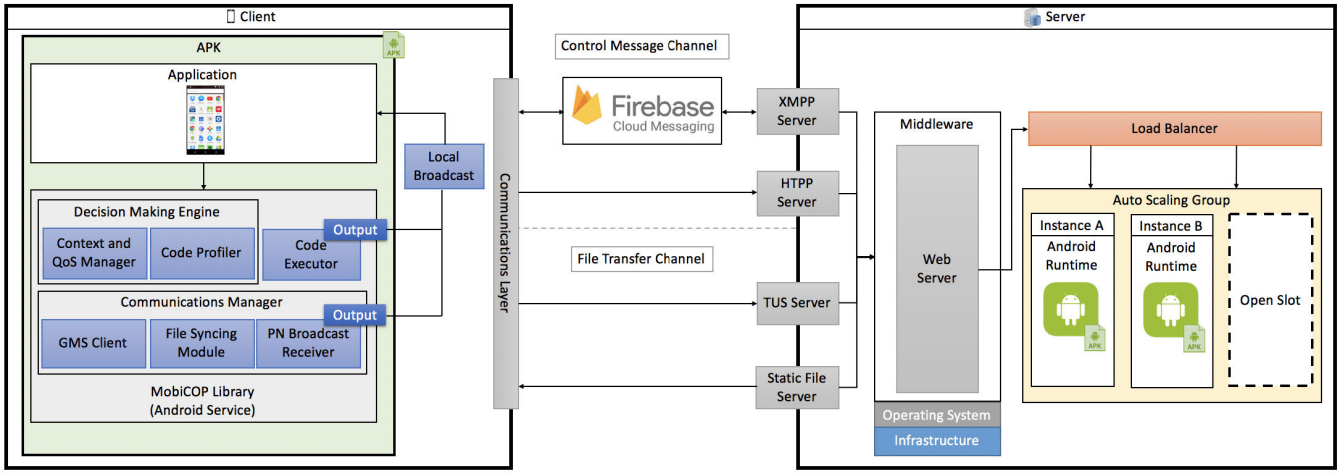

FIGURE 1. Overview of MobiCOP's architecture.

When a `CloudRunnable` task is dispatched, and our decision-making engine makes the choice to offload, the following process takes place: first, the input data is transferred, then the server executes the task, the result is later retrieved, and finally, a local broadcast is dispatched in the client to inform the task has been completed.

B. ANDROID SERVICES AND OREO BACKGROUND RESTRICTIONS

MobiCOP was designed specifically for long-running background tasks. We, therefore, wish to be compatible with tasks where the user is not required to wait in front of the screen until they are complete. Services are the de facto Android component for this type of work. Unlike activities, services are less likely to be destroyed by the operating system in case of resource shortage and may run concurrently with one another. Simply launching long-running background tasks in a separate thread on an activity when there is a strong chance for the user to stop interacting with the application and move it to the background before said task is finished is considered to be bad practice in Android. However, developers started to abuse services by having their apps make continuous usage of them for menial tasks, which would lead to multiple services running concurrently all the time, hogging the device's resources. Under such conditions, battery life became affected, and the probability of services getting killed regardless of the OS increased. In response, Google introduced significant behavior changes to background applications in Android Oreo [27]. Specifically, it introduced the concept of foreground and background app for purposes of service limitations (not to be confused with the standard foreground and background definition used in the context of memory management). In this context, an application is considered to be in the foreground if any of the following is true: it has a visible activity, it has a foreground service, or it has a component that is currently being used by another application (bound service or content provider). If none of the above is true, the application is considered to be in the background for the purposes of service limitations. Applications under these conditions are unable to start additional services, and all of their currently running services are automatically stopped by the operating system, with the exception of those placed under a temporary whitelist. This includes applications receiving a high-priority message using the Firebase Cloud Message (FCM) service or that have just recently been moved to the background in terms of memory management. This behavior applies to all applications running under Oreo or above that target API level 26 or above. Nevertheless, applications running on Oreo that target a lower API level can still have this behavior enforced through an option available in the system settings, and applications not targeting the latest API levels are continually being restricted from being published in Google's official store, the Play Store, which severely limits distribution. This is particularly problematic for frameworks such as MobiCOP that have been designed explicitly for long-running background tasks.

In order to solve this problem, MobiCOP admits two different behaviors: a light mode intended for background tasks not lasting for more than a couple of minutes, and an intrusive mode where the background tasks are launched on a foreground service instead. Foreground services are excluded from Oreo's background behavior changes but carry the disadvantage of requiring an indelible notification to be present to inform the end user a task is constantly draining the device's resources. Light mode is intended for tasks that can be completed during the small timeframe the OS grants applications to keep running background services after a user quits its UI or if it is acceptable for the background task to be aborted if the user quits the application. The usage of high-priority FCM messages will ensure the results retrieved from the server may be adequately processed by both foreground and background services. Which mode is more appropriate depends on the specific needs of the developer.

In the case of Android Pie, MobiCOP's intrusive mode also requires the developer to request the new

FOREGROUND_SERVICE normal permission in the application's manifest.

C. OPERATION MODES

It is often assumed that mobile code offloading frameworks should always choose a single context in which to run a task: either locally or in the cloud, depending often on the output of a decision-making engine. Nevertheless, since all manner of unforeseen circumstances can occur during the execution of a task, from sudden network disconnections to unexpected delays in the arrival of push notifications, it is impossible for a decision-making engine to always make the right choice. As such, it may be of interest to some developers to always run tasks concurrently on both the local device and the cloud environment, then keep the result from whichever task finishes first. Tango is one of the first mobile code offloading frameworks to suggest this approach. The main advantage of this method is that the best possible performance is always attained, albeit at the cost of higher battery consumption, as the client will always need to simultaneously make full use of its CPU (for local execution) and radio (for offloading). Nevertheless, this may be an acceptable compromise for certain use cases where performance is paramount.

MobiCOP acknowledges these potential use cases and offers two operation modes for developers that we call optimistic and concurrent. In optimistic mode, MobiCOP behaves as standard code offloading solutions, wherein a task is run in a single context depending on the output of a decision-making engine. However, in concurrent mode, the relevant task is always executed simultaneously locally and in the cloud, and only the result of whichever is finished first is kept. The developer may freely select which mode better fits the intended use case.

D. THE OFFLOADING DECISION

For years, authors have been struggling with figuring out how to best make the decision on whether to offload or not a given task. Decision-making engines should ideally adapt to changing network and device conditions in real-time and be capable of producing an output without introducing too much overhead [28]. A popular way of tackling this problem has been through linear optimization techniques: decision-making engines would model the offloading task as an integer linear programming (LP) problem and then make use of an LP solver to deliver a recommendation [13], [29]. However, LP problems can be computationally expensive to solve, as they have been proven to be NP-Hard. More advanced evidence-based methods suggest aggregating data in the cloud from traces obtained from potentially thousands of devices, in order to figure out precise rules based on fuzzy-logic, that are then pushed onto client devices so they can efficiently make the offloading decision [10], [30]. Although this is a very promising approach, the question remains on how to handle the offloading decision for new applications where no such past evidence exists. This is also known as the cold start problem in the field of machine learning.

MobiCOP solves both of these issues by using heuristics based on local historical data. MobiCOP's decision-making engine assumes that any given task that has been previously executed should take similar times to complete if a similar input is handed in. MobiCOP's decision engine also reduces its overhead by focusing solely on the performance variable. Energy consumption predictions based on software models are very inaccurate and depend on the specific device on which they are being run. Moreover, energy consumption is often related to a task's execution time, and, most of the time, longer tasks will consume more energy than shorter ones. The decision engine comprises two modules: a code profiler in charge of estimating how long a task would take to complete with and without offloading, and a network profiler responsible for estimating network availability, quality, and throughput.

The code profiler operates by registering in a local DB statistics of task executions run by the client. Our algorithm uses this information to estimate how long it would take a task to complete. If no historical data on said task is previously available, MobiCOP leverages its concurrent execution capabilities to run the task simultaneously on the client and the cloud, therefore solving the cold start problem. This serves to both collect data and not hamper client performance. Energy consumption may suffer slightly while in this mode, but MobiCOP will only run tasks concurrently until sufficient data is available to make educated decisions.

Once this condition is met, the code profiler's algorithm works as follows: for every task T_i its input set $I_i = \{i_1, i_2, \ldots, i_n\}$ is transformed into the vector V_i by applying the function F over every element of I_i. In our particular case, the input set corresponds to the hash map developers are required to pass to the framework. Different values of the input set are distinguished by the key-value pairs in the hash map. We then define F as follows:

$$F(i_x) = \begin{cases} i_x & \text{if } i_x \text{ is numeric} \\ length(i_x) & \text{if } i_x \text{ is string, array or file} \\ 1 \text{ or } 0 & \text{if} i_x \text{ is boolean} \end{cases} \quad (1)$$

Let $\hat{V}$ equal the normalized vector of V. Let t represent a task's execution time in seconds and l represent the size of a data set in bytes. Whenever a task is completed, the tuple $R_T = (t_{local}, t_{cloud}, l_{output})$ is stored in a KD-Tree by the client, with $\hat{V}$ as its search key. Let T_N be a new task to be executed. The decision-making engine executes a nearest neighbor algorithm to find those records R_{T_i} whose vector $\hat{V}_i$ most resembles $\hat{V}_N$. Let W_i be the inverse of the distance between $\hat{V}_i$ and $\hat{V}_N$; we can then make an estimate for R_{T_N} as:

$$R_{T_N} = \begin{cases} average(R_{T_i}) \forall R_{T_i} \mid \hat{V}_i = \hat{V}_n & \text{if } \exists i \mid \hat{V}_i = \hat{V}_N \\ \dfrac{\sum W_i \cdot R_{T_i}}{\sum W_i} & \text{otherwise} \end{cases} \quad (2)$$

On the other hand, the network profiler operates by periodically sampling the network to estimate availability, latency (L), and throughput speed (R). We perform a sample whenever we connect to a Wi-Fi network, when a network configuration change is detected (i.e., when switching from Wi-Fi to a mobile network or vice versa), and when our decision-making engine recommends running a task locally due to poor network connectivity. The latter is to confirm that the network is still down. If this is not the case, the following query to the decision-making engine will consider this updated information. Once connected to a network, the sampling is repeated periodically every 15 minutes on average. The network sampling task is encapsulated in an Android Job and later dispatched through the Job Scheduler API to be as friendly as possible with the battery. On the other hand, the profiling of mobile networks is considered a special case, as constant probing of a metered network over long periods of time could result in significant unwanted charges to the end user. For that reason, mobile network throughput speed is instead estimated using documented average values based on the mobile network connection's subtype (e.g., LGE, GPRS, EDGE, etc.).

Finally, the decision-making engine takes the output of these two modules and decides to offload the task only if the two following conditions are met:

- The server is reachable
- The following inequality is true:

$$t_{local} > \alpha \cdot \left(t_{cloud} + \frac{l_{input}}{R} + \frac{l_{output}}{R} \right) \tag{3}$$

The value *alfa* is an arbitrary multiplier currently set to 1.5 by default, as per the model recommended in [31]. Originally, we also considered the signal strength as reported by the Android API for making an offloading decision. However, subsequent testing revealed that this value was too unreliable to be of use: offloading was possible in many situations where the API reported low signal strength, whereas there were also several cases where offloading was impossible even if the device reported a strong signal. In the end, simply pinging the server was much more reliable.

While heuristics-based approaches are much less accurate than more advanced machine learning techniques, they are significantly more lightweight. Overall, the overhead introduced by our decision-making engine is negligible. Most importantly, our empirical testing revealed that accuracy when estimating the length of a task is far from paramount. Even in cases where substantial differences were detected between prediction and reality, our engine managed to behave reasonably well. Interestingly, the network profiler played a much more important role in making good decisions than the code profiler.

E. COMUNICATION CHANNELS

MobiCOP supports two ways to communicate with our server: standard HTTP or Google's FCM upstream messaging technology. The latter defines a communications channel in which XMPP messages are uploaded to Google's servers through the same socket used for push notifications and subsequently relayed to ours. FCM upstream messaging is advertised as offering several advantages over other standard HTTP communication, including increased energy efficiency and better usage of resources thanks to socket reuse and the asynchronous nature of XMPP. As such, MobiCOP's first implementation was exclusively built using this technology.

Unfortunately, long-term testing revealed that FCM upstream messaging has a strong tendency to introduce significant delays in dispatching messages queued for transfer in background processes. The closed-source nature of Google Play Services prevents us from pinpointing the exact reason for this behavior, but we assume this is due to Google's energy-saving policies. Because of this, standard HTTP was later introduced as well.

Developers may select whichever channel they prefer. Although HTTP has proven to be much more reliable in our testing, the increased energy efficiency offered by upstream messaging cannot be overlooked. Therefore, we recommend using the former when reliability is pivotal and diverting to the latter if the developer is more interested in saving power.

IV. EXPERIMENTAL SETUP

In order to better understand the behavior and potential benefits of mobile code offloading in real-life environments, we conducted an experiment in which volunteers installed a benchmarking app fitted with MobiCOP. When given a certain predetermined task, this application would intermittently run a job which executes a set of predefined tasks. First, the job uses the MobiCOP decision-making engine to analyze each task, and then it would simultaneously attempt to execute the task on the local device and offload it to the server.

A. RESEARCH QUESTIONS

The overall goal of this experiment is to evaluate the performance of MobiCOP in real-life environments, in particular, to answer the following research questions:

- *RQ.1. What is the impact of real-life environment on MobiCOP's* ***execution time*** *measurements?*
- *RQ.2. What is the impact of real-life environment on MobiCOP's* ***decision-engine accuracy****?*
- *RQ.3. What is the impact of real-life environment on the overall* ***code offloading performance****?*

These questions are important to understand and quantify the actual benefits of MobiCOP in terms of execution time in a real-life environment. More precisely, we are interested in evaluating the accuracy of MobiCOP's decision engine, in particular, how often its outputs correspond to the environment in which the task is actually run faster.

B. METHODOLOGY OVERVIEW

To answer our research questions, we define a linear five-step methodology:

1) *Benchmarks selection*
2) *Benchmark suite APK implementation*
3) *APK distribution and devices under study*
4) *Metrics & data model*
5) *Benchmark execution and data collection*

The rest of the subsections carefully detail each one of these steps.

C. BENCHMARKS SELECTION

We use three types of benchmarks, one in which the benchmarks were configured to execute a short, but CPU intensive task with small amounts of input and output data (small shared state); another one in which the task involved a long CPU intensive task that requires the transfer of large amounts of data between client and server (large shared state); and one which requires the use of GPU and/or CPU to solve a machine-learning related task. We believe these three environments are the most representative of the actual use cases where offloading may be involved and it is comparable to similar experiments made in the state of the art (See Table 2).

TABLE 2. Experiments used in the state of the art of mobile code offloading frameworks.

Framework	Experiments	Type
MobiCOP [5]	N-Queens, Video transcoding	CPU-Intensive, Large-file
Clonecloud [7]	Virus scanning, image searching	CPU-Intensive, Large-file
Jade [13]	Face detection, Text search	Machine learning, CPU-Intensive
Cuckoo [32]	Object recognition	Machine learning
ThinkAir [12]	N-Queens, Face detection	CPU-Intensive, Machine learning
LAMCO [24]	Montecarlo simulation, N-Queens	CPU-Intensive

Below, we describe each one of these benchmarks:

- *N-Queens (CPU-Intensive)* – The task in question was the N-Queens problem, a common benchmarking task in code offloading literature. This task consists of computing all valid N queens' placements on a chess board so that no queen is in range of one another. A value of $N = 14$ was chosen for this experiment.
- *Video Transcoding (Large-File)* – The task consisted of transcoding a .webm video into an .mp4 equivalent using the FFMEPG library. The input video had a size of 3 MB, while the output video weighed about 4.5 MB. Because of the large amounts of shared state in this second experiment, the benchmarking application imposed a much bigger burden on our users' data plans and batteries: we estimate it needed to transfer about 1 GB of data per day in the background.
- *Image Recognition (Machine Learning)* – The task consisted in detecting and listing objects present in a picture using a deep learning model called Mobilenet [33] configured to use the GPU if it is available.

D. BENCHMARKS SUITE APK IMPLEMENTATION

We developed a mobile APK application that contains all the considered benchmarks. We took special care to design it in such a way that the benchmarking applications would interfere the least possible with our participants' usage of their phones and their battery. For that reason, we encapsulated the benchmarking task in Android JobService and programmed it to be run irregularly every 15 minutes using Android's Job Scheduler API. Job Scheduler is an API introduced in Android Lollipop that allows the operating system to coordinate background tasks among multiple applications so all of them may run consecutively during time windows determined by the OS. The purpose of this API is to bundle as many background jobs as possible in order to minimize intermittent wakeups and therefore reduce battery consumption. As such, with our given parameters, it does its best to ensure jobs are executed once every 15 minutes, but two consecutive jobs will not necessarily be scheduled exactly 15 minutes apart. `JobServices` run by the Job Scheduler API are also not subjected to Oreo's background service limitations. From here onwards, we will use the term job instance to refer to a single run of the aforementioned job.

We also took into consideration the Doze feature introduced in Android Marshmallow. Doze may defer the triggering of scheduled jobs when the device has remained stationary after a given amount of time if it has the screen off and is not currently plugged into a power source. From Android Nougat onwards, an extended Doze mechanism may cause job deferrals even when the device is not stationary. We wanted to minimize the impact of Doze in our experiments. Therefore we asked our participants to add our application to their devices' Doze whitelist to prevent this behavior.

E. PARTICIPANTS, DEVICES AND APK DISTRIBUTION

Our experiment involved a total of 18 volunteers, each with a mobile device. Participants device specifications are available in Table 3. We contacted each participant via email to distribute the corresponding mobile APK application containing our benchmarks through the Play Store Beta functionality, as well as instructions on how to use it. The research team had no physical access to our participants' devices or face-to-face contact with most of them. In particular, no additional special configuration was applied to their devices.

We then asked our users to always carry their smartphones with them as we wanted to evaluate MobiCOP's behavior under a wide variety of real-world situations. We wanted to avoid the case in which users would simply leave their devices lying on a flat surface for the entire duration of the experiment. Each volunteer was required to collect at least seven days of data, although some kept the benchmarking application running for longer. Since we experienced a high degree of unreliability using FCM upstream messaging in the

long run, we configured MobiCOP's control message channel to use HTTP.

Amongst our participants, we also had owners of devices with Android Oreo or greater. Since we did not want Oreo's background limitations to interfere with our results, our benchmark app was configured to launch MobiCOP in intrusive mode on devices with Android Oreo, and on light mode on devices with Nougat and below. While running our application, each of our volunteers generated between 100 and 1000 job instance records. Additionally, the architecture used for our application is the same base architecture used for any MobiCOP application (Figure 1). We distributed the load to three AWS instances with the same configuration to conduct this experiment. Our server's middleware was deployed on an *AWS t2.micro* machine, while our Android execution environment was deployed on a *t2.small instance*. Finally, we would like to mention that although more powerful machines are available on AWS, we had to exclude them from this experiment due to financial constraints. Therefore, better results are expected if this experiment is reproduced on the most powerful AWS machines.

F. METRICS & DATA MODEL

Let t_x be a job instance that performs a benchmark x either in the cloud or on the mobile device, where $x \in (n-queens, transcoding, machine-learning)$.

1) PROFILING METRICS

For each time a job t_x is executed, we collected the following metrics:

- $location(t_x)$: it is a tuple $\{longitude, latitude\}$ that represent where the t_x was performed.
- $network\_latency(t_x)$: it is the estimated network latency at the moment that t_x was performed.
- $network\_speed(t_x)$: it is the estimated network throughput speed at the moment that t_x was performed.
- $network\_king(t_x)$: it is the type of network connectivity at the moment that t_x was performed. $network\_king(t_x) \in \{WIFI, MOBILE\}$
- $power\_source(t_x) \in \{YES, NO\}$: it indicates if the device was connected or not to a power source (for this, we used an API only available on Android Marshmallow or above). $power\_source(t_x) \in \{YES, NO\}$
- $decision(t_x)$: it represents the decision made by the MobiCOP engine. The decision could be LOCAL, CLOUD or CONCURRENT; the latter only happens when no data is available. In this particular case, two concurrent jobs are excluded from the analysis.
- $P_{local}(t_x)$: it is the predicted local execution time for task t_x. This prediction is done by the MobiCOP's cost model.
- $P_{cloud}(t_x)$: it is the predicted cloud execution time for task t_x. It includes the time that MobiCOP takes to transfer the input and output to the cloud.
- $A_{local}(t_x)$: actual completion time that task t_x takes when running locally.
- $A_{cloud}(t_x)$: actual completion time that task t_x takes when the task is offloaded. It may be equal to infinity if t_x fails to offload, or if the result is never retrieved due to network problems.

2) ACCURACY

We also computed the accuracy of the decision engine. We refer to accuracy regarding the number of times that MobiCOP's decision was correct. We consider that a decision was correct in two cases:

- *Local Accuracy:* when MobiCOP decides to compute the benchmark locally $decision(t_x) = LOCAL$, and the time for executing the benchmark locally is shorter than the time for executing the same benchmark in the cloud $A_{local}(t_x) < A_{cloud}(t_x)$.
- *Cloud Accuracy:* when MobiCOP decides to compute the benchmark in the cloud $decision(t_x) = CLOUD$, and the execution time is shorter than executing it locally $A_{cloud}(t_x) < A_{local}(t_x)$.

Note that regardless of whether MobiCOP decides to run the benchmark locally or in the cloud, we run the benchmark both locally and in the cloud, in order to measure the execution time and determine if the decision is correct. Energy measurements are beyond the scope of this study as precise energy readings require physical access to the device and connecting it to a power monitor.

3) TOTAL COMPLETION TIME(TCT)

To understand the benefits of MobiCOP's decision-engine, we contrast the time that MobiCOP requires to resolve a given task versus the time of executing all the tasks locally or in the cloud. We define the total completion time as the amount of time a user would have been required to wait from the moment he or she dispatches a task to the moment its result is available for consumption. Therefore, the completion time TCT for a task t_x is defined as follows:

$$TCT(t_x) = \begin{cases} A_{local}(t_x) & \text{if } DE(T_x) = LOCAL \\ min(A_{cloud}(t_x), P_{cloud}(T_x) + A_{local}(T_x)) & \text{if } DE(T_x) = CLOUD \end{cases} \quad (4)$$

Note that for local executions, this is simply the time it takes the client device to complete its computation. When offloading, on the other hand, the completion time depends on the output of the decision engine and the fallback policies when erroneous decisions to offload are made. Consider that MobiCOP falls back to a local execution whenever the result of an offloaded task T_x fails to be retrieved before $P_{cloud}(T_x)$ has transpired. Since P_{cloud} is slightly overestimated in our model, instances where a cloud result is retrieved after a local fallback has been triggered but before it has completed should be rare. When that does happen, though, the cloud result is reported back to the client, and an event is triggered to abort the local execution.

TABLE 3. Description of devices used during experiments.

User	Brand	CPU	RAM	OS
A	Xiaomi M2004J19C	Octa-core (2x2.0 GHz Cortex-A75 & 6x1.8 GHz Cortex-A55)	6GB	Android 10
B	Xiaomi M2006C3LG	Octa-core (4x2.0 GHz Cortex-A53 & 4x1.5 GHz Cortex-A53)	6GB	Android 10
C	HUAWEI ELE-L29	Octa-core (2x2.6 GHz Cortex-A76 & 2x1.92 GHz Cortex-A76 & 4x1.8 GHz Cortex-A55)	8GB	Android 9
D	samsung SM-A205G	Octa-core (2x1.6 GHz Cortex-A73 & 6x1.35 GHz Cortex-A53)	3GB	Android 9
E	samsung SM-A530F	Octa-core (2x2.2 GHz Cortex-A73 & 6x1.6 GHz Cortex-A53)	4GB	Android 8
F	HUAWEI JKM-LX3	Octa-core (4x2.2 GHz Cortex-A73 & 4x1.7 GHz Cortex-A53)	3GB	Android 8.1
G	Xiaomi M2102J20SG	Octa-core (1x2.96 GHz Kryo 485 Gold & 3x2.42 GHz Kryo 485 Gold & 4x1.78 GHz Kryo 485 Silver)	6GB	Android 11
H	samsung SM-A315G	Octa-core (2x2.0 GHz Cortex-A75 & 6x1.7 GHz Cortex-A55)	4GB	Android 10
I	motorola moto g(6)	Octa-core 1.8 GHz Cortex-A53	3GB	Android 8
J	samsung SM-A022M	Quad-core 1.5 GHz Cortex-A53	3GB	Android 10
K	Sony H8314	Octa-core (4x2.7 GHz Kryo 385 Gold & 4x1.7 GHz Kryo 385 Silver)	4GB	Android 8
L	Xiaomi Redmi 6A	Quad-core 2.0 GHz Cortex-A53	2GB	Android 8.1
M	HUAWEI STK-LX3	Octa-core (4x2.2 GHz Cortex-A73 & 4x1.7 GHz Cortex-A53)	4GB	Android 9
N	samsung SM-N985F	Octa-core (2x2.73 GHz Mongoose M5 & 2x2.50 GHz Cortex-A76 & 4x2.0 GHz Cortex-A55)	8GB	Android 10
O	Xiaomi Mi A1	Octa-core 2.0 GHz Cortex-A53	4GB	Android 7.1
P	samsung SM-G770F	Octa-core (1x2.84 GHz Kryo 485 & 3x2.42 GHz Kryo 485 & 4x1.78 GHz Kryo 485)	6GB	Android 10
Q	samsung SM-G973U1	Octa-core (2x2.73 GHz Mongoose M4 & 2x2.31 GHz Cortex-A75 & 4x1.95 GHz Cortex-A55)	6GB	Android 9
R	samsung SM-N986U	Octa-core (2x2.73 GHz Mongoose M5 & 2x2.50 GHz Cortex-A76 & 4x2.0 GHz Cortex-A55)	12GB	Android 11

4) SERVICE AVAILABILITY

Service availability, on the other hand, is simply the percentage of instances where the user would have been able to actually run the task in the cloud. For cloud execution, this is analogous to the number of instances where a network connection was available to the client. It makes the most sense to compare average task completion between an offloading framework and full local execution and to compare service availability between an offloading framework and full cloud execution. The corresponding results for MobiCOP were that we always had 100% of service availability because when the connection failed, we used the local environment to complete the task.

G. BENCHMARK EXECUTION AND DATA COLLECTION

Each of the 18 participants, once the benchmark APK was installed, started to execute the benchmarks every 15 minutes. We asked each participant to keep the application running for at least seven days. As a consequence, on average, each benchmark was executed around 670 times. The number of executions varies between devices due to different situations during the experiment. Among the most common situations are that the mobile device turns off or experiences problems with the network connectivity. For each job execution, we collect all previously mentioned metrics and save them in the cloud for further data analysis.

To ensure our users have used their devices in different conditions and places, we recorded the geographic coordinates of the place where each job instance started. Since each job instance took place in 15-minute intervals, we do not see a continuous trail of executions across a map, but rather clusters centered around our users' frequently visited locations. Figure 2 shows the locations where the job instances were triggered. Note that we have participants from four different countries that transported their devices around different parts of the cities with different connection conditions.

TABLE 4. CLOUD: means and standard deviations of predicted and actual offloading executions (in milliseconds).

Device		Estimated Time		Actual Time	
		Average	Standard Deviation	Average	Standard Deviation
N-Queens Experiment					
A		11908.81	597.13	9463.48	6141.45
B		12314.96	5585.65	12385.88	14633.00
C		11362.50	275.68	9280.71	494.25
E		11237.81	847.52	21443.51	27270.82
G		11931.06	674.89	20436.00	22919.48
H		11864.84	696.17	10200.76	3188.25
I		13428.60	76.74	11938.60	3976.11
J		12122.80	173.56	8741.80	715.98
K		11785.85	483.45	14033.79	17213.01
L		12025.35	1021.02	15144.35	18517.77
M		14216.64	2592.55	11161.34	8068.94
N		12238.54	2266.30	14152.80	15483.24
O		13047.14	5388.71	13470.41	10147.60
P		11409.07	595.73	9543.93	5312.96
Q		11707.62	1471.01	12962.62	16888.70
R		12299.15	2145.16	26527.48	30502.90
Transcoding					
G		26989.94	907.64	99507.24	104172.30
H		26275.25	326.94	60821.56	103415.33
K		27328.51	969.80	148506.81	127295.72
N		26669.6	2804.09	101837.50	136400.62
O		26923.74	3612.69	88214.86	104028.49
P		27034.04	1019.09	57257.84	90232.89
Q		27445.76	1172.83	70647.84	104654.90
R		27661.68	3652.01	148365.21	133582.76
Machine Learning					
No GPU	B	1329.84	419.39	6768.53	3142.68
GPU	A	1101.74	782.51	6526.89	3647.42
	E	749.90	279.13	7102.62	11551.69
	K	1310.75	1008.24	26843.50	3290.58
	N	1357.98	1655.10	8260.45	8221.27
	O.00	1938.78	3206.64	22709.55	12238.02

V. RESULTS

A. EXECUTION TIME (RQ.1.)

In order to answer our first research question, each time a benchmark is executed, we compute the estimated execution time and actual execution time for executing the benchmark in the cloud and locally. Table 4 and Table 5 summarize the estimated and the actual execution times for performing the benchmarks in the cloud and locally, respectively.

1) BENCHMARKS EXECUTION TIME

Our results show that 13 of 16 devices execute the N-queens benchmark faster in the cloud than locally, on average. Our

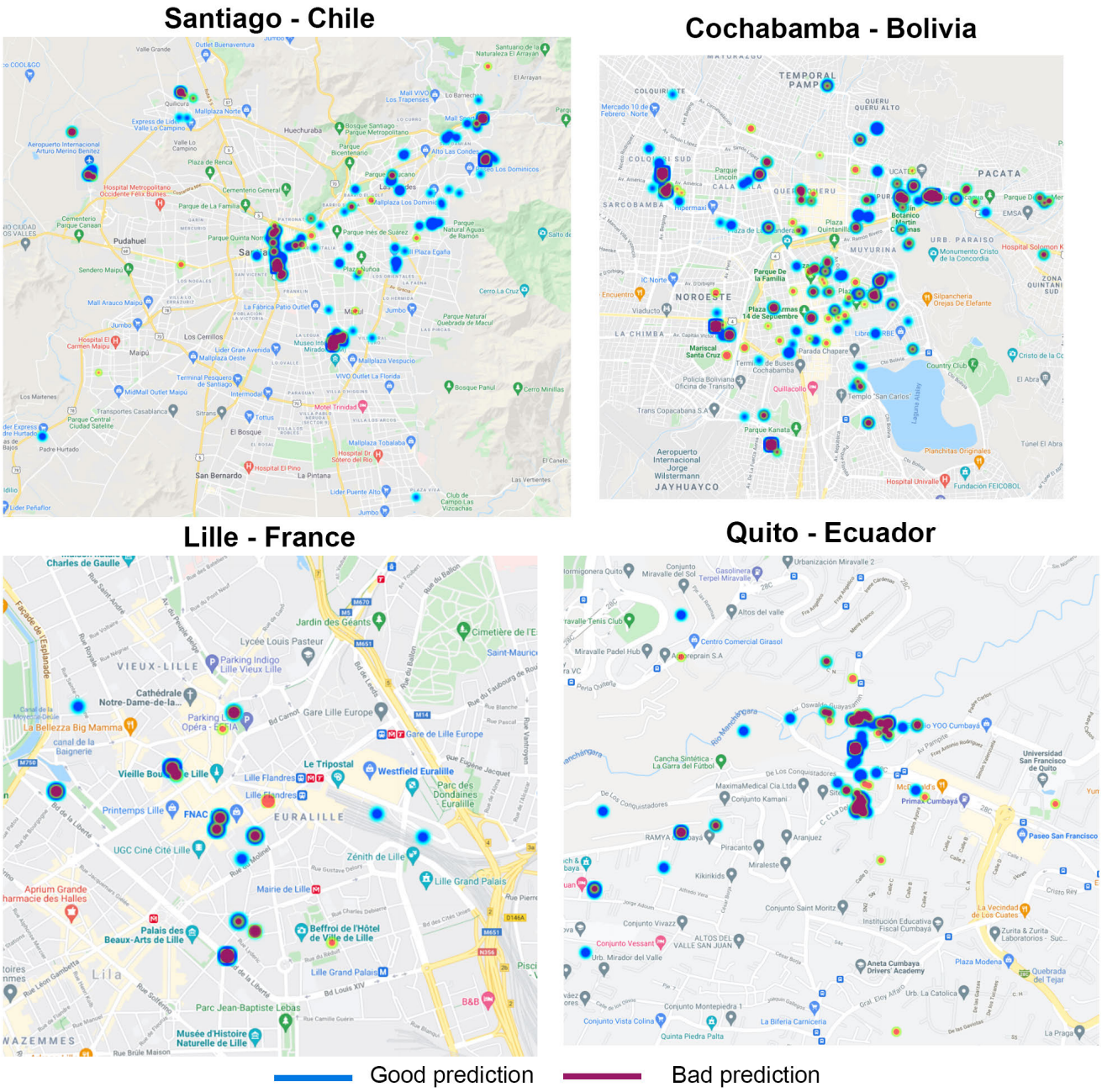

FIGURE 2. Geographic distribution of job instances for different users in different city contexts. Clusters around the users' frequently visited places are clearly noticeable.

AWS machine managed to complete the N-Queens benchmark in between 9 and 25 seconds, while our mobile devices would require up to 80 seconds. For the transcoding benchmark, it took about 100 seconds on average to finish the task in the cloud, while mobile devices could take from 160 to 513 seconds. The only exceptions worth mentioning are devices E, P, and R. All are high-end devices with Octa-core processors. Although the server performs N-Queens benchmark faster than most devices, device P managed to consistently outperform our server, while devices E and R were only slightly slower (not enough to make offloading recommendable). This happened because our deployed server has lower computing capabilities than a number of devices, in addition, there is a considerable time spent on sending and receiving the information from the server.

Regarding the machine learning benchmark, our results indicate that the cloud performs this benchmark slower than all devices. As we mentioned before, this is due mainly to the lower capacity of the server we used for the experiment. In this particular benchmark, participants' devices had

TABLE 5. LOCAL: means and standard deviations of predicted and actual local executions (in milliseconds).

Device		Estimated Time		Actual Time	
		Average	**Standard Deviation**	**Average**	**Standard Deviation**
N-Queens Experiment					
A		12153.31	145.79	12486.39	2828.24
B		23291.86	126.12	24969.56	7485.00
C		213782.2857	45509.26	21955.36	10631.56
E		18874.13	198.32	20516.31	7186.47
G		6644.54	69.26	6709.46	163.73
H		13940.32	468.18	14785.84	2040.70
I		276.00	57.55	28108.40	269.69
J		31075.60	513.06	45667.40	18588.51
K		19325.88	1502.08	14931.28	7438.93
L		26192.10	446.41	26723.16	3072.26
M		549394.19	105022.58	41761.21	106207.15
N		37580.32	63085.72	30453.50	92141.74
O		23285.91	294.98	25644.57	4189.82
P		7731.86	4.98	6923.06	39.73
Q		22548.68	92.18	22613.68	2677.68
R		18951.59	317.31	22466.07	477.31
Transcoding					
G		58473.84	1682.37	48587.68	17419.56
H		52662.81	16245.28	58258.69	43397.17
K		53838.06	2658.95	53490.14	13990.78
N		106506.45	84351.68	147868.48	547665.10
O		55335.83	1677.18	57464.32	9742.57
P		58848.44	5780.31	76090.77	15342.19
Q		67752.82	1949.50	59511.81	10312.50
R		77281.65	1509.30	72999.42	4572.05
Machine Learning					
No GPU	B	786.65	19.05	2033.83	3058.85
GPU	A	1792.28	215.55	1987.39	706.72
	E	2519.52	88.67	2662.33	378.12
	K	2568.55	307.29	2413.70	1173.56
	N	2497.3	145.06	3032.85	1782.83
	O	5273.43	712.86	9845.64	5749.55

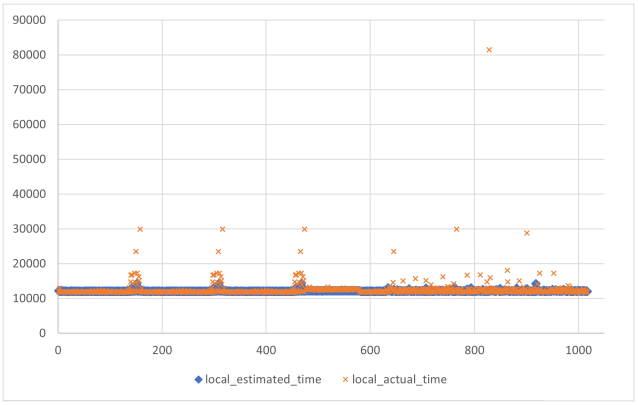

FIGURE 3. Example of estimated and actual execution times for local tasks (user A).

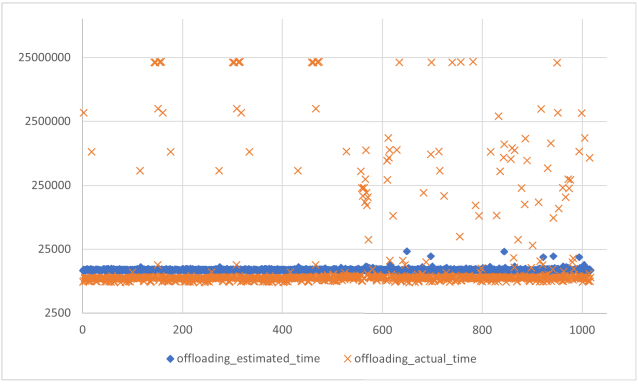

FIGURE 4. Example of estimated and actual execution times for offloaded tasks (user A). Due to the massive dispersion introduced by failed tasks that arrived too late, results are presented on a logarithmic scale.

better performance than the server in the cloud. Nevertheless, we decided to include these results in this study as they allow us to analyze our chosen engine's behavior (situations that should always be contemplated in code offloading frameworks).

2) MEASUREMENTS VARIABILITY

Another matter of interest for all benchmarks is that we noticed a great amount of variability in the amount of time needed to execute the same tasks throughout the duration of the experiment. As it has been shown in the past, diverse factors may affect execution time measurement, among them, the other tasks executed in parallel by the mobile users, the restrictions put in place by the OS on the processor's speed for background tasks, and the network connection.

For the cloud execution time, measurement variability is much more prevalent, and it presented itself on all users. One reason is that MobiCOP receives offloaded tasks' results through push notifications. And occasionally, devices would fail to receive these notifications immediately and instead would suffer a delay of several minutes. Because of this, there were various cases where an offloaded task's result would be retrieved up to 15 minutes late. This is mostly due to intermittent disconnections and the fact that the Android OS has to wait for a 'heartbeat' duration to attempt to reconnect the push notification service. As such, mean and standard deviation values seem particularly inflated. Therefore, in this case, we assumed that the offloading task failed when it surpassed a threshold of time. In Figure 3 and Figure 4 it is illustrated how the dispersion is present in both cases in the case of user A.

For the transcoding and machine-learning benchmarks, much more dispersion is noticeable for offloaded tasks. This is explained by the fact that the execution time for an offloaded task is strongly dependent on the quality of the network connection. On slow networks, the transferal of the necessary input and output files can delay overall task completion time by several dozen seconds. Because of that, the standard deviation remains above 10 seconds for some of our users.

RQ.1. What is the impact of real-life environment on MobiCOP's execution time measurements? As expected, the network and device's real-life conditions introduce a greater degree of variation in the time measurement across our three benchmarks, in both, local and cloud computation. One reason is that the cloud execution time measurements are highly dependent on the network connection to receive and send the input and output, and the local execution time depends on the

other additional tasks/applications executed on the same device. Despite these variations, our results show that N-queens and video transcoding benchmarks execute faster in the cloud than locally on the mobile device (on average), with the exception of high-end devices which perform faster than our modest AWS server in the cloud. However, the machine learning benchmark is a heavy task even for our server in the cloud. Due to this, most mobile devices perform this benchmark faster.

B. ACCURACY (RQ.2.)

This section describes the factors that affect MobiCOP's decision-engine, and whether said decision is correct or not. MobiCOP's decision-engine, first, estimated the potential benchmark execution time of performing the benchmark locally and in the cloud, then it decided whether or not it is more convenient to execute the benchmark locally or in the cloud. Table 6 summarizes the number of correct predictions that MobiCOP realized during our experiment, as well as the accuracy of MobiCOP when it decided to execute a given task locally or in the cloud.

1) PREDICTED EXECUTION TIME

MobiCOP uses an execution time estimation (predicted) of executing a given task locally and in the cloud, then uses these predictions to decide whether or not to offload the task. By contrasting the average time predictions and the real execution time, we found that predictions for local execution time are much closer $P_{local}(t_x)$ to the actual execution time $A_{local}(t_X)$ than the cloud execution time predictions. As we mentioned before, this is reasonable considering the multiple factors that can alter actual execution time when connecting to the cloud. Additionally, we have to consider that in our particular model, estimates for remote execution are inflated by a factor of 0.5. This is to ensure that only those tasks where benefits will be substantial are offloaded. Our results are concordant with this heuristic as the mean error rate between predicted and actual execution times, after excluding failed tasks, varies between 10% and 50%. Predictions for local tasks were much more reliable, with error rates below the 10% threshold for most cases. Unfortunately, for those same users where we observed a large dispersion in local task execution times, predictions were rather poor. This is especially the case for users K and L during the N-queens benchmark. Regarding users O and N during the transcoding benchmark, while predictions showed slightly worse results than when compared with the rest of the users, they remained overall solid. For user K, the reason for this behavior was a large number of failed tasks throughout the experiment which caused the decision-making engine to overestimate local execution times.

In the case of the Machine Learning benchmark, cloud execution time predictions $P_{cloud}(t_x)$ were unrealistic compared to the actual execution time $A_{cloud(t_x)}$, except for the case of user B. In this case, our engine always underestimated the offloading case, making an incorrect prediction preferring offloading over local, when the real situation shows that it is preferable to keep the task on the local side. This can be explained probably because the engine assumes that both environments have the same hardware optimizations, when in reality, the cloud server, due to the environmental restrictions, does not have GPU capabilities to use in machine learning. Because of the restrictions of Genymotion's Android version vs. those on the local side, the device has enabled GPU capabilities for machine learning tasks. In the case of user B for this experiment, the accuracy is better due to the absence of specialized hardware to do this kind of task (this is the only device that did not report a compatible GPU for machine learning tasks), so even with the sub estimation of the offloading task, the real times for local environment were almost always worse, making that, in this case, better predictions. In the other two experiments, however, in some cases, due to the changing conditions, the decision-making engine failed to adapt to new settings properly, provoking our engine's failure to predict the correct environment.

Figures 5, 6, and 7 show the similitude and distribution of the real-time and predicted time in both environments for N-Queens, Transcoding, and Machine Learning experiments respectively. These graphics also show that the engine, even

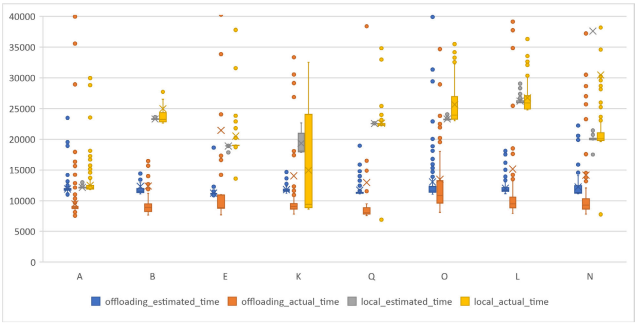

FIGURE 5. Distribution of predicted vs real-time obtained in the local and offloading case for the N-queens experiment.

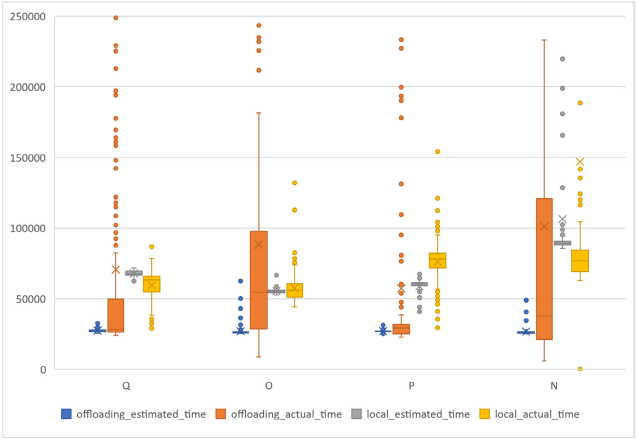

FIGURE 6. Distribution of predicted vs real-time obtained in the local and offloading case for the Transcoding experiment.

TABLE 6. Accuracy of prediction for experiments.

Device	Total experiments	Correct predictions	Total Cloud	Correct cloud predictions	Accuracy	Accuracy Cloud	Accuracy Local
N-Queens							
A	977	777	911	768	79.53%	84.30%	13.64%
B	198	159	159	157	80.30%	98.74%	5.13%
C	32	30	28	28	93.75%	100.00%	50.00%
E	320	136	136	134	42.50%	98.53%	1.09%
G	61	61	0	0	100.00%	-	100.00%
H	75	63	66	63	84.00%	95.45%	0.00%
I	10	10	10	10	100.00%	100.00%	-
J	10	10	10	10	100.00%	100.00%	-
K	366	186	186	186	50.82%	100.00%	0.00%
L	214	136	136	136	63.55%	100.00%	0.00%
M	61	54	53	53	88.52%	100.00%	12.50%
N	166	115	108	103	69.28%	95.37%	20.69%
O	249	205	211	205	82.33%	97.16%	0.00%
P	18	18	0	0	100.00%	-	100.00%
Q	114	67	67	67	58.77%	100.00%	0.00%
R	38	20	20	18	52.63%	90.00%	11.11%
Transcoding							
G	66	21	21	21	31.82%	100.00%	0.00%
H	59	45	45	45	76.27%	100.00%	0.00%
K	384	24	24	24	6.25%	100.00%	0.00%
N	145	109	109	109	75.17%	100.00%	0.00%
O	250	134	133	133	53.60%	100.00%	0.85%
P	228	203	203	203	89.04%	100.00%	0.00%
Q	340	256	256	256	75.29%	100.00%	0.00%
R	47	22	22	22	46.81%	100.00%	0.00%
Machine Learning							
A	261	9	0	0	3.45%	-	3.45%
B	49	43	0	0	87.76%	-	87.76%
E	42	4	4	4	9.52%	100.00%	0.00%
K	60	3	0	0	5.00%	-	5.00%
N	40	5	0	0	12.50%	-	12.50%
O	51	8	4	4	15.69%	100.00%	8.51%

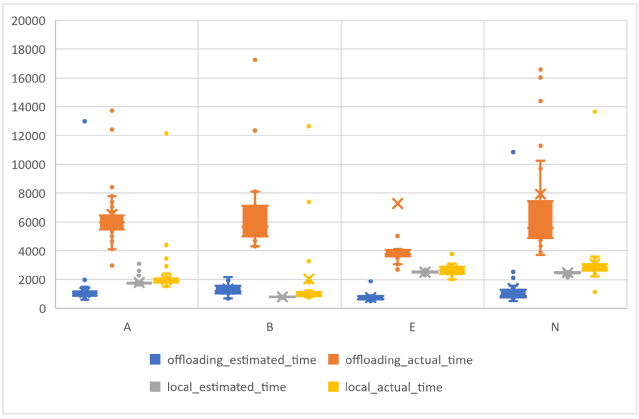

FIGURE 7. Distribution of predicted vs real-time obtained in the local and offloading case for the Machine Learning experiment.

when the environmental conditions are unstable, maintains consistency in the predictions.

2) LOCAL AND CLOUD ACCURACY

Table 6 shows the overall accuracy of the predictions of our engine and the number of executions where our decision-making engine made the correct decision (that is, the engine's output is equal to the environment where the execution would be optimal). Overall, we can see an accuracy of about 70% or more amongst most of our users (users G, I, J, and P even experienced an accuracy of 100% in the N-Queens experiment). Additionally, we can see that our decision-making engine works rather well when recommending offloading to the cloud, as comparatively speaking, fewer mistakes are made when making this decision. However, even when the predicted times for local environments are relatively precise, it does show some weaknesses when recommending a local environment (with an error rate of up to 60% when recommending a local environment). A common cause for this phenomenon is an incorrect diagnosis of network availability. We have noticed this trend among users that often move through areas with intermittent connectivity, such as when making long journeys between cities.

RQ.2. What is the impact of real-life environment on MobiCOP's decision-engine accuracy?

Overall, the decision accuracy of offloading a given task varies from 84.3% to 100%. However, the decision of performing the task locally has a very low accuracy (below 20% in almost all the cases). This last result is related to the execution time predictions. Although, with N-queens and transcoding benchmarks, local time predictions are close to really having error rates below 10% in most of the cases, cloud time predictions have a

TABLE 7. Performance gains comparing the obtained local time vs. offloading time.

Device	Average cloud time	Average Network Latency	Average local time	Performance Gain
N-Queens				
A	9463.48	701.30	12486.33	32%
B	12385.88	618.01	24969.56	102%
C	9280.71	305.73	21955.36	137%
E	21443.51	1475.36	20516.31	-4%
G	20436.00	985.88	6709.46	-67%
H	10200.76	867.48	14785.84	45%
I	11938.60	659.23	28108.40	135%
J	8741.80	567.46	45667.40	422%
K	14033.79	580.80	14931.28	6%
L	15144.35	1426.00	26723.16	76%
M	11161.34	570.80	41761.21	274%
N	14152.80	409.09	30453.50	115%
O	13470.42	333.93	25644.58	90%
P	9543.93	405.57	6923.07	-27%
Q	12962.62	964.15	22613.69	74%
R	26527.48	517.05	22466.07	-15%
Transcoding				
G	148365.21	385.24	72999.43	-51%
H	70647.84	554.87	59511.82	-16%
K	57257.84	498.00	76090.77	33%
N	88214.86	884.74	57464.33	-35%
O	101837.50	1055.85	147868.48	45%
P	148506.81	431.37	53490.14	-64%
Q	60821.56	362.39	58258.69	-4%
R	99507.24	1335.23	48587.68	-51%
Machine Learning				
A	6526.89	536.41	1987.39	-70%
B	6768.53	629.59	2033.84	-70%
E	7102.62	303.86	2662.33	-63%
K	26843.50	681.60	2413.70	-91%
N	8260.45	708.15	3032.85	-63%
O	22709.55	1097.06	9845.65	-57%

greater error rate (between 10% and 50%). In particular, machine-learning benchmark cloud execution time predictions are weighted far differently for the actual execution time. This error rate is related to a number of factors including the low GPU capabilities of our server in the cloud, the high-end technology of a number of devices, and the high measurement variability due to the network connection.

C. OVERALL MobiCOP PERFORMANCE (RQ.3.)

To analyze the benefits of MobiCOP, we contrast the total completion time $TCT(t_x)$ that MobiCOP requires to perform a task (either locally or in the cloud) against the execution time of executing a task only locally $A_{local}(t_x)$. Table 7 illustrates a detailed performance comparison between using offloading and running in a local environment. Note that network latency averages vary, which affects response time. These situations occur because the experiment was conducted in real-life conditions and not in a laboratory where the variables are controlled. Therefore, the latency varies due to diverse situations, such as the fact that the participants and their devices moved around their cities and the type of connection is too heterogeneous (e.g., 3G, 4G). Figures 8, 9 and 10 show in a graphical view, MobiCOP's average total execution time (in blue) and the average local execution time of executing the three benchmarks.

Figure 8 reveals that our code offloading framework does a relatively good job of determining if it is more convenient, in terms of performance, to offload the n-queens benchmark task. On average, in all but two devices (E and R) MobiCOP's completion time is shorter than always running the task locally. We can see that average task completion times show a significant improvement when using MobiCOP. Even though our decision-making engine is far from perfect, overall, our end-users would have experienced between 30% to 420% increased wait times had they relied on a completely local implementation.

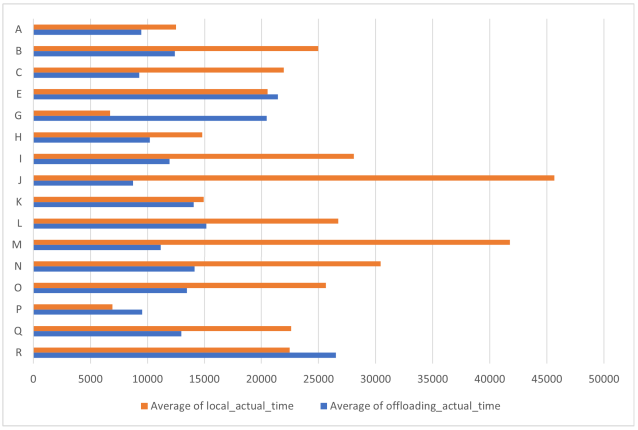

FIGURE 8. Performance gains comparing the obtained local time vs. offloading time for N-queens experiment.

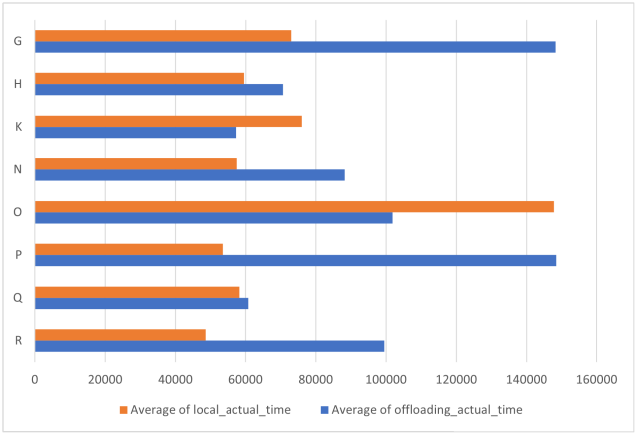

FIGURE 9. Performance gains comparing the obtained local time vs offloading time for the Transcoding experiment.

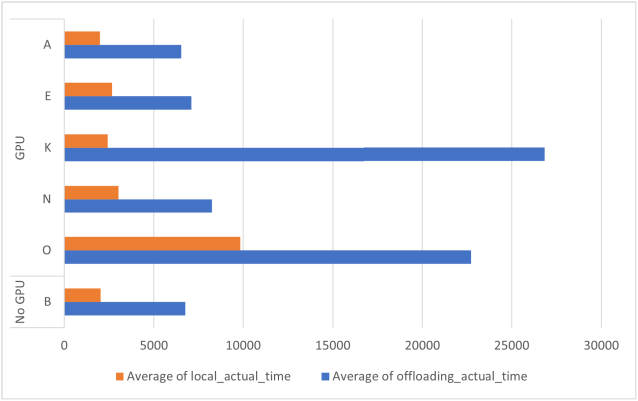

FIGURE 10. Performance gains comparing the obtained local time vs offloading time for Machine Learning experiment.

Figure 9 and Figure 10 show the result for the benchmark transcoding and machine learning, respectively. Both figures show that MobiCOP spends more time resolving the task than simply executing benchmarks locally. This is due to the restricted capabilities of our server and the mobile internet service availability. Service availability always depends on a user's location and access to a mobile network. While some users were unaffected by this (they always had network access), the rest of our users would have been unable to use the application up to one-third of the time had they relied on a local-only implementation.

RQ.3. What is the impact of a real-life environment on the overall code offloading performance? The low accuracy of the decision-making engine for benchmarks transcoding and machine-learning has a negative impact on total execution time that MobiCOP requires to perform these benchmarks, being more convenient in these cases to perform all the tasks locally. In the case of N-queens benchmark, we can observe a performance improvement between 30% and 420%, depending on the device.

VI. DISCUSSION AND LESSONS LEARNED

There were many insights we gained throughout the course of this experiment.

A. NETWORK AVAILABILITY

Much of the related work we have observed in the literature puts a great amount of emphasis on the description of their code profiler's algorithm and the many parameters collected in order to make a good prediction. The most advanced implementations include very low-level characteristics, such as CPU instruction cycles, per-thread instruction cycles, allocated RAM, and garbage collector calls. These parameters are usually not available through the standard SDKs and require hacking the OS to get access. While, indeed, this information may yield much more accurate predictions than the algorithm we have presented so far, we noticed that in many cases, the differences between execution in the cloud and on a mobile device are so great that minor inaccuracies on the exact amount of time a task takes to complete do not affect the decision-making engine's output in a significant way. Our estimates when offloading a code are usually off by about 30% to 50%; our estimates for local code execution are usually much better, yet we still noticed some cases where most of them were off by more than 100%. However, even in these extreme situations, the decision-making engine can correctly guess the fastest context for running the task on most occasions. Ironically, we have noticed in previous works that very little focus, if any, is given to the network profilers. We believe this is a mistake and that this area deserves more attention.

A closer look at the data reveals that for those cases where the decision-making engine makes a mistake, the most common cause of error is a network interruption that prevented the push notification containing the result to arrive on time. Keeping a socket open with the server throughout the entire remote execution may aid in solving this problem, but its cost on battery life can be excessive. Ideally, more work should be done to improve network profilers to predict when these network outages are most likely to occur. This is not to say, however, that code profilers are irrelevant. Boosting their accuracy may play an integral part for the decision-making engine for tasks where the differences between local and remote execution are smaller (however, in this case, the benefits of code offloading will be far less substantial) and for reducing the task completion time when a task is wrongly offloaded to the cloud.

Furthermore, researchers should consider that under real-life conditions, users and their devices commonly move between different places around their cities and therefore have different connection conditions (see Figure 2). As a result, the bandwidth is heterogeneous and the network latency varies, impacting the performance of the code offloading models.

Lesson 1. The network profiler accuracy has a high impact on the estimated execution time and, on the accuracy of the decision-engine. Therefore, we recommend considering evaluating code offloading approaches keeping in mind real-life network conditions.

B. STRESS TESTING

Originally, MobiCOP has always performed exceptionally well in laboratory experiments. Nevertheless, when moving into the field, we noticed that our Android execution environment was prone to various errors that would only surface after various days of constant operation. This would mostly occur when a single server instance received far too many offloading requests to handle. Even though our server is able to automatically scale horizontally, spawning new machines takes some time, and sudden spikes in the number of offloading requests can produce this situation. It is, therefore strongly recommended for future code offloading solutions to ensure a mechanism is put in place to constantly verify the status of the running server instances and repair them when necessary. Luckily, AWS Auto Scaling took care of most of this work for us.

Lesson 2. Android server execution environments can be very brittle, therefore, we recommend performing stress testing to evaluate the code offloading implementation in the server to avoid potential issues with multiple concurrent users.

C. NETWORK SIGNAL DIAGNOSIS

It is a well-known fact that all mobile devices offer an API that returns an estimate of the network's signal strength. Originally, we tried to take advantage of this metric to improve the predictions of our decision-making engine. Unfortunately, we found this data to be extremely unreliable to be of use. In our original work, our decision engine defined a threshold below which tasks would never be offloaded due to the risk of disconnections. However, practical testing revealed that this constraint introduced an intolerable number of false negatives (the decision-making engine would refuse to offload a task, even though there would have been no issues in attempting to do so). Ignoring this parameter altogether significantly improved the accuracy of our decision-making engine as the cases when the offloaded tasks would fail to complete in time when under a poor network connection ended up being rather low. If a future mobile code offloading framework seeks to incorporate this metric, at the very least, it should be combined with something else if an improvement is to be made. Otherwise, simply testing for server reachability is enough.

Lesson 3. Network signal quality indicator on Android is extremely unreliable. Therefore, we believe that we need to further investigate and understand the factors that influence the network signal indicator in a separate experiment.

D. EXECUTION TIME MEASUREMENTS

Against our expectations, we found out that in some cases, running the same task on the same mobile hardware with exactly the same input data can take very different amounts of time to complete. Although some variation was expected, data recorded for several users showed differences in local execution time between the fastest and slowest run of the same task of up to a factor of nine. This behavior seems more prevalent on older models with less computing power at their disposal. Because of this, more advanced code profilers should take into consideration the dynamics of the system environment of the mobile device.

Lesson 4. During our experiment, we computed the cloud execution time as the sum of the time the server needs to execute the benchmark plus the time spent to send and receive the input/output. In the future, we recommend measuring both execution times separately to better quantify the impact of the network on the offloading performance.

E. HIGH DEVICE CAPABILITIES

It was interesting to find out that in the case of the experimentation of machine learning when all the environment and the machine learning model are optimized to run on mobile devices, we must prefer to run in the local environment rather than doing it in the cloud with offloading. This is mainly due to the networking part, where we must send the input to a remote server, causing some bottleneck in that workflow. In addition to this, the cloud Android device used to run the offloaded code, does not have GPU capabilities due to the restrictions of Genymotion accessing a GPU-capable device. In these cases, the older devices with reduced performance are better candidates to do offloading, when even with the optimizations, the devices cannot get better results due to hardware constraints. Even with this situation, offloading becomes attractive when the app must deal with a heterogeneous user base that older devices will probably have. This will be true even when the hardware improves over time, mainly because the machine learning models are also growing in size over time, adapting to the new hardware but deprecating older ones. Also, we believe that by using code offloading, even when the performance indication tells us to do it on mobile, we can gain energy savings for the user in some cases, making offloading still a valid and attractive option to use.

Lesson 5. In the majority of situations, using offloading can help to obtain better results than a local environment, but in some cases, local execution is better than offloading when the task is optimized for running on mobile devices. Anyway, using offloading when it is necessary can help to run some difficult applications on older devices, and even if local execution can perform better in terms of execution time, offloading can help to save devices' energy.

VII. CONCLUSION

Mobile code offloading has been around in academia for over a decade, yet practical implementation difficulties have prevented this technology from actually being tested in the wild until now. In this work, we presented an extensive testing framework with various new metrics for evaluating the performance of such solutions when making the move to the field.

In these situations, the decision-making engine is critical for the success of the mobile code offloading framework, but details on how this component may be evaluated have not been published in the literature so far. Although one may think its accuracy when estimating the execution time when running locally or remotely is the most important factor, we have shown our code offloading framework can actually be fairly tolerant to prediction inaccuracies and still deliver good recommendations on where it would be more convenient to run a particular piece of code. As such, code offloading framework quality metrics should be focused on the actual benefits perceived by the end-user.

Determining the percentage of cases where the decision-making engine guesses the fastest execution environment is

a good first approach but comparing against an exclusively local implementation through average task completion time can also not be ignored to properly weight the impact of wrong predictions. Additionally, special care should be taken by researchers and engineers to ensure their code offloading solutions operate well in areas with no Internet access (even if the device is connected to a network), and on a wide variety of device models, especially those that might be faster than their server environment. The latter is even more important in the Android ecosystem, where fragmentation is particularly more significant than, for example, iOS.

MobiCOP's decision-making engine is not without its limitations though, and a substantial engineering effort is needed to improve its performance. For instance, it currently assumes the server environment is homogenous, which might not necessarily be the case. Also, it fails to properly filter out outliers when predicting task execution time, which might have a strong incidence on averages due to the massive dispersion detected in our results. Future work will involve figuring out a way to overcome these limitations.

We hope this work will serve as a guideline for the development of future offloading platforms that might be used in practice, and that developers and researchers can use this work's results as a baseline to improve upon.

ACKNOWLEDGMENT

The authors thank Christian Eilers for his support and contribution during the experimentation along with the different volunteers.

REFERENCES

[1] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generat. Comput. Syst.*, vol. 29, no. 1, pp. 84–106, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X12001318

[2] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya, "Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 337–368, 1st Quart., 2014.

[3] K. Sekar, "Power and thermal challenges in mobile devices," in *Proc. 19th Annu. Int. Conf. Mobile Comput. Netw. (MobiCom)*, 2013, pp. 363–368, doi: 10.1145/2500423.2505320.

[4] J. I. Benedetto, A. Neyem, J. Navon, and G. Valenzuela, "Rethinking the mobile code offloading paradigm: From concept to practice," in *Proc. IEEE/ACM 4th Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*, May 2017, pp. 63–67.

[5] J. I. Benedetto, G. Valenzuela, P. Sanabria, A. Neyem, J. Navón, and C. Poellabauer, "MobiCOP: A scalable and reliable mobile code offloading solution," *Wireless Commun. Mobile Comput.*, vol. 2018, pp. 1–18, Jan. 2018, doi: 10.1155/2018/8715294.

[6] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst., Appl., Services*, Jun. 2010, pp. 49–62, doi: 10.1145/1814433.1814441.

[7] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, Apr. 2011, pp. 301–314, doi: 10.1145/1966445.1966473.

[8] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "$*COMET*$: Code offload by migrating execution transparently," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement. (OSDI)*. Hollywood, CA, USA: USENIX Association, Oct. 2012, pp. 93–106. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gordon

[9] Y. Li and W. Gao, "Minimizing context migration in mobile code offload," *IEEE Trans. Mobile Comput.*, vol. 16, no. 4, pp. 1005–1018, Apr. 2017.

[10] H. Flores and S. Srirama, "Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning," in *Proc. 4th ACM Workshop Mobile Cloud Comput. Services*, Jun. 2013, pp. 9–16, doi: 10.1145/2497306.2482984.

[11] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, "Accelerating mobile applications through flip-flop replication," in *Proc. 13th Annu. Int. Conf. Mobile Syst., Appl., Services*, May 2015, pp. 137–150, doi: 10.1145/2742647.2742649.

[12] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 945–953.

[13] H. Qian and D. Andresen, "Jade: An efficient energy-aware computation offloading system with heterogeneous network interface bonding for ad-hoc networked mobile devices," in *Proc. 15th IEEE/ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput. (SNPD)*, Jun. 2014, pp. 1–8.

[14] R. Friedman and N. Hauser, "COARA: Code offloading on Android with AspectJ," 2016, *arXiv:1604.00641*.

[15] R. Montella, S. Kosta, D. Oro, J. Vera, C. Fernández, C. Palmieri, D. Di Luccio, G. Giunta, M. Lapegna, and G. Laccetti, "Accelerating Linux and Android applications on low-power devices through remote GPGPU offloading," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 24, Dec. 2017, Art. no. e4286, doi: 10.1002/cpe.4286.

[16] F. Armand, M. Gien, G. Maigné, and G. Mardinian, "Shared device driver model for virtualized mobile handsets," in *Proc. 1st Workshop Virtualization Mobile Comput.*, Jun. 2008, pp. 12–16, doi: 10.1145/1622103.1622104.

[17] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *Proc. 10th Workshop ACM SIGOPS Eur. Workshop, Beyond PC (EW)*, 2002, pp. 87–92, doi: 10.1145/1133373.1133390.

[18] M. R. Rahimi, N. Venkatasubramanian, and A. V. Vasilakos, "MuSIC: Mobility-aware optimal service allocation in mobile cloud computing," in *Proc. IEEE 6th Int. Conf. Cloud Comput.*, Jun. 2013, pp. 75–82.

[19] M. R. Rahimi, N. Venkatasubramanian, S. Mehrotra, and A. V. Vasilakos, "On optimal and fair service allocation in mobile cloud computing," *IEEE Trans. Cloud Comput.*, vol. 6, no. 3, pp. 815–828, Jul. 2018.

[20] B. Zhou, A. V. Dastjerdi, R. N. Calheiros, S. N. Srirama, and R. Buyya, "A context sensitive offloading scheme for mobile cloud computing service," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 869–876.

[21] K. Lee and I. Shin, "User mobility-aware decision making for mobile computation offloading," in *Proc. IEEE 1st Int. Conf. Cyber-Physical Syst., Netw., Appl. (CPSNA)*, Aug. 2013, pp. 116–119.

[22] J. L. D. Neto, D. F. Macedo, and J. M. S. Nogueira, "Location aware decision engine to offload mobile computation to the cloud," in *Proc. NOMS - IEEE/IFIP Netw. Oper. Manage. Symp.*, Apr. 2016, pp. 543–549.

[23] A. Ravi and S. K. Peddoju, "Mobile computation bursting: An application partitioning and offloading decision engine," in *Proc. 19th Int. Conf. Distrib. Comput. Netw.*, Jan. 2018, pp. 1–10, doi: 10.1145/3154273.3154299.

[24] Y. Ballan, A. Ahmed, and N. Baghaei, "LAMCO: A layered approach to mobile application computation offloading," in *Proc. Int. Wireless Commun. Mobile Comput. (IWCMC)*, Jun. 2020, pp. 1336–1341.

[25] P. A. L. Rego, F. A. M. Trinta, M. Z. Hasan, and J. N. de Souza, "Enhancing offloading systems with smart decisions, adaptive monitoring, and mobility support," *Wireless Commun. Mobile Comput.*, vol. 2019, pp. 1–18, Apr. 2019, doi: 10.1155/2019/1975312.

[26] T.-Y. Lin, T.-A. Lin, C.-H. Hsu, and C.-T. King, "Context-aware decision engine for mobile cloud offloading," in *Proc. IEEE Wireless Commun. Netw. Conf. Workshops (WCNCW)*, Apr. 2013, pp. 111–116.

[27] Google. *Background Execution Limits: Android Developers*. Accessed: May 26, 2023. [Online]. Available: https://bit.ly/33RVJcD

[28] Y. Zhang, H. Liu, L. Jiao, and X. Fu, "To offload or not to offload: An efficient code partition algorithm for mobile cloud computing," in *Proc. IEEE 1st Int. Conf. Cloud Netw. (CLOUDNET)*, Nov. 2012, pp. 80–86.

[29] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: Profile-based partitioning for sensornet applications," in *Proc. NSDI*, vol. 9, 2009, pp. 395–408.

[30] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile code offloading: From concept to practice and beyond," *IEEE Commun. Mag.*, vol. 53, no. 3, pp. 80–88, Mar. 2015.

[31] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring Android Java code for on-demand computation offloading," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 233–248, Oct. 2012, doi: 10.1145/2398857.2384634.

[32] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: A computation offloading framework for smartphones," in *Mobile Computing, Applications, and Services: Second International ICST Conference, MobiCASE 2010, Santa Clara, CA, USA, October 25–28, 2010, Revised Selected Papers 2*. Berlin, Germany: Springer, 2012, pp. 59–79.

[33] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

SANABRIA PABLO is currently pursuing the Ph.D. degree in computer science with Pontificia Universidad Católica de Chile. He has worked in several projects related to mobile applications and web applications in private software development companies. He is a part of the Software Engineering and Intelligent Systems Laboratory (SEIS Lab). His research interests include mobile applications, edge computing, virtual reality, and machine learning.

NEYEM ANDRES received the Ph.D. degree in computer science from the Universidad de Chile. He is a Professor with the Computer Science Department, Pontificia Universidad Católica de Chile. His research interests include software engineering, mobile and cloud computing, machine learning for intelligent systems, engineering and medical education, and extended reality. In these research areas, on the one hand, he has authored or coauthored a wide range of papers in conferences proceedings and journals, and, on the other hand, he has developed several software products of these types of cloud-based mobile systems.

SANDOVAL ALCOCER JUAN PABLO received the Ph.D. degree in computer science from the University of Chile, Chile, in 2016. He is an Assistant Professor at the Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile. He is a part of the Software Engineering and Intelligent Systems Laboratory (SEIS Lab). His research interests include software engineering, more specifically in the fields of software maintenance, mining software repositories, software performance, software visualization, and search-based software testing. He participated as a reviewer expert in various prestigious conferences and journals in the field including: EMSE, IEEE VIS, VISSOFT, JSS, and IST. He is also a member of the Pharo Community.

FERNANDEZ BLANCO ALISON is currently pursuing the Ph.D. degree in computer science with the University of Chile, Chile. She is a part-time Lecturer of software engineering at the Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile. She is a part of the Software Engineering and Intelligent Systems Laboratory (SEIS Lab). Her research interests include software visualization, software performance, software maintenance, data mining, and search-based software testing.