Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 8715294, 18 pages
https://doi.org/10.1155/2018/8715294

Research Article

MobiCOP: A Scalable and Reliable Mobile Code Offloading Solution

José I. Benedetto,[1] Guillermo Valenzuela,[1] Pablo Sanabria,[1] Andrés Neyem,[1] Jaime Navón,[1] and Christian Poellabauer[2]

[1] *Computer Science Department, Pontificia Universidad Católica de Chile, Santiago, Chile*
[2] *Computer Science and Engineering Department, University of Notre Dame, Notre Dame, IN, USA*

Correspondence should be addressed to José I. Benedetto; jibenede@uc.cl

Received 26 September 2017; Accepted 12 December 2017; Published 9 January 2018

Academic Editor: Konstantinos E. Psannis

Code offloading is a popular technique for extending the natural capabilities of mobile devices by migrating processor-intensive tasks to resource-rich surrogates. Despite multiple platforms for offloading being available in academia, these frameworks have yet to permeate the industry. One of the primary reasons for this is limited experimentation in practical settings and lack of reliability, scalability, and options for distribution. This paper introduces MobiCOP, a new code offloading framework designed from the ground up with these requirements in mind. It features a novel design fully self-contained in a library and offers compatibility with most stock Android devices available today. Compared to local task executions, MobiCOP offers performance improvements of up to 17x and increased battery efficiency of up to 25x, shows minimum performance degradation in environments with unstable networks, and features an autoscaling module that allows its server counterpart to scale to an arbitrary number of offloading requests. It is compatible with the most relevant Android technologies optimized for heavy computation (NDK and Renderscript) and has so far been well received by fellow mobile developers. We hope MobiCOP will help bring mobile code offloading closer to the industry realm.

1. Introduction

Over the last few years, we have witnessed an unprecedented growth in the popularity of mobile devices. This is mostly due to an increased demand of premium smartphones and the high availability of low-cost devices in emerging markets. The fact is that this trend will continue, as predictions made by Cisco indicate that, by 2021, there will be 11.6 billion mobile-connected devices (around 1.5 mobile devices per capita) [1]. With such promising prospects, hardware and software manufacturers have been pushing the boundaries of mobile devices ever further in an attempt to capture this growing mass of consumers. As such, multicore processors, high-resolution displays, powerful graphics processing units, and multiple sensors have become the norm. However, despite the massive advancements in the overall capabilities of mobile devices, battery life has been struggling to keep up. According to Sekar, "power remains and will remain a first-class design constraint to the continued advancement of mobile technology" [2]. Thus, power optimization has remained an important topic of research amongst mobile platform engineers. In response, industry leaders Google and Apple have recently introduced new operating system features meant to restrict the usage of power-hungry applications and thus prolong battery life. For example, iOS 9 introduced a Low Power Mode that can be manually enabled in order to reduce the amount of battery consumed by iPhone and iPad devices; Android Marshmallow on the other hand introduced Doze, a set of heuristics meant to disable certain features, such as networking or alarms, in case no activity is detected on the device. However, a common trend in these recent innovations is that all efforts to tackle the problem have been solely focused on local optimizations, leaving the potential of the cloud untapped.

Cloud computing has been widely recognized as the next generation of computing infrastructure, enabling on demand powerful servers and other computing resources [3].

Mobile cloud computing (MCC) in particular has been acknowledged as an efficient means for improving the capabilities of mobile devices. Within MCC, code offloading has been proposed as a technique for increasing the apparent capabilities of mobile platforms, as well as their battery life [4], by transparently migrating computation-intensive tasks to a resource-rich offsite surrogate not constrained by power limitations. Good candidates for offloading are tasks that perform extensive CPU or GPU computation, although, more recently, migrating network-intensive operations have also been proven to yield benefits due to the more reliable and higher throughput characteristics of the wired connectivity of a server [5]. However, even though performance benefits of one order of magnitude and battery benefits of up to two orders of magnitude have been reported, no code offloading solution has yet been seen in widespread use in the community.

Despite promising results in controlled laboratory environments, we argue that a lack of evaluation in real-life scenarios shows that there are several hurdles yet to be addressed by code offloading that would impede practical applications. An analysis of these solutions allows us to group these obstacles into three major categories: reliability, scalability, and distribution. Concerning reliability, the reliance on synchronous full-duplex communication, often across the entire duration of an offloaded task, is unrealistic for real-life environments, where mobile networks have a strong tendency to intermittently fail. For multimedia applications, which rely on very large file sizes, this constraint is severely restrictive [6]. Concerning scalability, most currently published solutions have failed to provide proof of support for multiple clients simultaneously. Indeed, for some architectures, it is strongly suggested that a one-to-one ratio between servers and clients is needed for proper operation. Beyond that, we know of very little work that includes an autonomic computing module for supporting arbitrary loads. Finally, for distribution purposes, many solutions are based on customized OS versions that are extremely difficult to configure for the average user and require the potential audience to voluntarily skip eventual security patches and feature updates.

In this paper, we present a new computation offloading platform named MobiCOP (mobile computation offloading platform) [7]. Contrary to most other solutions, MobiCOP avoids dealing with minute optimizations that may arise from migrating specific methods in arbitrary locations and instead focuses on the optimization of whole long-running computation-intensive tasks present in modern applications. This solution has been specifically designed to address the three constraints mentioned above. In particular, one major advantage of MobiCOP is that, unlike all other known alternatives, MobiCOP's client is fully self-contained in a library and is compatible with most stock Android devices. As such, adding it to an existing Android project is as easy as adding a single Gradle dependency. The server component on the other hand is available through a public AWS AMI, which allows interested parties to have a fully configured MobiCOP server instance in a matter of minutes. It is, to our knowledge, the easiest mobile code offloading solution to reproduce.

This paper presents the following major contributions.

(1) It presents MobiCOP, a new fully functional code offloading framework that simultaneously satisfies the requirements of reliability, scalability, and distribution present in the mobile industry.

(2) It presents a new paradigm for implementing code offloading that requires neither OS customization nor integration with extraneous third-party tools.

(3) It introduces a new evidence-based decision-making engine that enables accurate predictions for the execution time of arbitrary code in most common scenarios.

The rest of the paper is organized as follows: Section 2 presents the related work; Section 3 presents a detailed description of MobiCOP, our proposed solution; Section 4 presents multiple evaluations used to validate the solution and offers a brief discussion of the results; lastly, in Section 5, we highlight MobiCOP's contributions and research implications and present our conclusions.

2. Related Work

Several code offloading frameworks that show promising results have been proposed in academia. In this section, we list some of the most prominent fully developed code offloading frameworks featuring a complete end-to-end solution.

Satyanarayanan et al. propose the cloudlet, an extension of cyber foraging, that seeks to harness the capabilities of proximate public clouds [8]. Instead of executing CPU-intensive tasks on distant servers via the web, they propose the usage of nearby resource-rich public computers that become accessible through short-range wireless technologies. Offloading can then be achieved through virtualization. Cloudlets are preconfigured with several base VMs that feature all common functionalities needed by the potential clients. Those in need of additional computer resources and in range of a cloudlet can then transmit a lightweight VM overlay that, in conjunction with its associated base, can achieve a quick replication of the client's environment and enable synergy between them. While theoretically sound, this solution requires a massive cloudlet infrastructure that is not yet available on a large scale today. Gai et al. on the other hand perceive cloudlets as an intermediate layer that plays a mediator role between mobile devices and cloud servers. They propose a "dynamic energy-aware cloudlet-based mobile cloud computing model" (DECM) [9], which uses dynamic programming to determine which servers, in a wide and heterogeneous cloud, should be accessed by a client in order to minimize net energy consumption and therefore achieve green computing. However, DECM focuses only on the architectural model and offers no specific implementation.

More recently, an evolution of the cloudlet paradigm has been suggested in the form of mobile-edge cloud computing [10]. In this model, servers located at the edge of pervasive radio access networks in close proximity to mobile users complement their capabilities by harnessing the resources of the distant cloud. Edge servers can then offload (partially or totally) computation to the cloud in a speedy manner, thanks to the use of high throughput wired connectivity, and then relay the result back to the client through short-range communication technologies. Unfortunately, while multiple mobile-edge computing architecture models currently exist [11], again, no specific implementation is currently available for widespread use.

In line with the idea of harnessing nearby computing resources, another proposed use of offloading is migrating parallelizable resource-intensive tasks to proximate idle mobile devices connected through an ad hoc network. One example of this technology is the Hyrax framework, an Android-compatible MapReduce implementation based on Apache Hadoop that allows running a parallelizable task across multiple nearby nodes, that is, other similar devices in the near vicinity accessible through a short-range wireless channel [12]. Nevertheless, performance in practical scenarios is still an issue due to the unreliability of public mobile ad hoc networks, lack of incentives for entering the network, and the fact that MapReduce has not been optimized for such situations.

Qian and Andresen propose Jade [13], a computation offloading framework targeted at mobile devices running the Android operating system. This framework minimizes the energy consumption of power-hungry tasks by offloading them to a server with a Java VM installed, through the usage of Remote Procedure Calls (RPC). It provides a simple programming model that enables Android developers to create arbitrary remotable objects by implementing a Java interface. Jade provides a runtime, which defines three main components: a profiler, an optimizer, and a communication handler. The profiler consists of two subcomponents: a program profiler (estimating the energy cost of a given remotable object) and a device profiler (keeping track of the network's availability and throughput). Secondly, the optimizer is in charge of determining whether to offload a remotable object or not depending on the profiler's output. Finally, the communications layer performs the offloading operation by implementing a suspend-wait-resume scheme on the client and contacting a server deployed on another mobile device through either Wi-Fi or Bluetooth.

Moving onto distant server-based solutions, Cuckoo enables code offloading by extending the Android programming paradigm that is already familiar to mobile developers [14]. In Cuckoo, tasks that may be offloaded are encapsulated in services. Later, service invocations are captured by the framework and executed either locally or remotely according to the current conditions.

MAUI is a .NET code offloading framework for maximizing energy savings [15]. It achieves code portability by leveraging the properties of the .NET Common Language Runtime. Developers are asked to manually annotate methods that may be offloaded. Afterwards, these methods are extracted via reflection and a server-side solver uses input from a client-side profiler to decide whether to offload or not.

CloneCloud is another code offloading framework built around a customized Cupcake Android branch [16]. These modifications are necessary for the framework to support dynamic profiling and migration. Unlike MAUI, CloneCloud is able to automatically select portions of code to be offloaded without programmer intervention. This is achieved by having a static analyzer that determines all legal partition points within a certain set of constraints and estimating the timing between the entry and exit points of said partitions with and without offloading. Server-side execution is achieved via application-level virtual machines and communication is handled by a migrator thread.

ThinkAir is another code offloading framework that seeks to address MAUI's scalability issues by supporting on demand cloud resource allocation [17]. It also solves CloneCloud's application, input, and environmental conditions restrictions by adopting an online method-level offloading mechanism. ThinkAir handles scalability by creating a complete VM of a smartphone system in the cloud and providing an interface that allows clients to ask for either more powerful or a greater number of VMs to handle the most resource-intensive tasks. Methods susceptible to being offloaded are annotated by the developer. This allows a code generator that runs at compile time to build the necessary communications layer between client and server.

COMET explores a new way of performing code offloading by experimenting with distributed shared memory (DSM) [18]. For this to work, the authors extended the DalvikVM by adding several VM synchronization primitives to the Android CyanogenMod Gingerbread release so it could support DSM. COMET operates at the thread level, allowing the same threads to run on two different environments. A full-duplex byte stream is then established to allow both threads to migrate from one environment to the other and to share their state.

Flores et al. propose EMCO, a code offloading framework with a heavy emphasis on strengthening offloading decisions by exploiting crowdsourcing and evidence-based learning methods in the cloud [19, 20]. A server-based neural network algorithm analyzes offloading traces of clients and uses them to deduce if-then-else rules for offloading which are later pushed to mobile clients as fuzzy rules via GCM. Input from multiple clients can therefore be used to develop highly accurate decision models.

Finally, Gordon et al. take one step back and propose an alternative offloading architecture in their latest framework, Tango. Instead of relying on an inherently imperfect decision-making engine, the full app is replicated in a cloud environment and run simultaneously at all times [5]. A modified deterministic thread-scheduling mechanism allows both environments to produce the exact same output under all possible circumstances. Out of both replicas, the fastest is considered the leader and the one driving the program forward; should a context switch result in the slower replica becoming faster, the leadership roles are exchanged through what the authors refer to as flip-flop replication.

Table 1 shows a summary of the different approaches mentioned above. The main problem common to all of the previous platforms is that very little to no evidence is supplied regarding how well they would operate under real-life scenarios, where network instability is the norm and server traffic spikes are to be expected. In general, we have noticed that many of these frameworks incorporate features found on desktop-level code offloading solutions, and, as

Table 1: Comparison of current mobile code offloading frameworks.

Frameworks	Purpose	Offloading mechanism	Partitioning	Preparation	Offloading Decision	Granularity level
Hyrax	Energy and performance	Code	MapReduce algorithm	Replication on server with Java VM	Not available	Class
Jade	Energy	Code	Static analysis	Replication on server with Java VM	Dynamic	Class
Cuckoo	Energy or performance	Code	API	Replication on server with Java VM	Not available	Method
MAUI	Energy	Code	Annotations	Replication on server with .NET CLR	Dynamic, on server	Method
CloneCloud	Performance	VM	Dynamic profiling	App/device cloning	Dynamic	Thread
ThinkAir	Scalability	VM	Annotations	Code generator	Not available	Method
COMET	Performance	Code	Dynamic profiling	Distributed shared memory	Dynamic	Thread
EMCO	Performance	Code	Annotations	Code generator	Dynamic	Method
Tango	Energy and Performance	Code	N/A	Flip-flop replication	Always concurrent	Entire app

such, they disregard some of the most prominent issues arising in mobile environments. MobiCOP seeks to address most of these concerns.

3. The MobiCOP Platform

3.1. Motivation. There exists a substantial amount of research in the area of OS-level requirements necessary for the implementation of optimal mobile offloading solutions. Many proof of concepts have been designed by forking the latest Android Open-Source Project (AOSP) branch and adding several features with this objective in mind, such as the ability to collect low-level metrics on the execution of arbitrary functions and DSM. However, until now, very little work has been done in exploring the feasibility of constructing a mobile code offloading solution with unaltered currently available mobile operating systems. MobiCOP was born from the desire to explore this possibility. Our main research objectives are twofold: first, we seek to evaluate the robustness of a code offloading solution built with the standard Android API; second, we seek to address the engineering shortcomings present in currently available mobile code offloading solutions that impede usage in practice.

This research shows that it is possible to build a fairly robust fully featured code offloading framework for the Android operating system without modifying the OS. Furthermore, it offers several design considerations that should be taken into account in the construction of future code offloading solutions to improve overall quality, and it lists several challenges that need to be overcome if we seek to implement these solutions in practical settings.

3.2. Design Overview. MobiCOP is a fully functional code offloading framework that offers implementation of all modules expected of such a system, including a remote execution environment, a decision-making engine, and a communication layer. It is designed to minimize execution time and power consumption on long-running background tasks.

In the Android operating system, extended background tasks should ideally be handled by an Android service component in order to decouple the task from the user interface in an activity and to allow the OS to eventually destroy activities to recycle memory without hampering the background operation. Additionally, services are less likely to be destroyed in case of resource shortages than activities; therefore, operations encapsulated by a service are more likely to complete. Once a service has been defined by the developer, invoking it is fairly straightforward. The only caveat of this process is that activities and services act as isolated components; therefore, in order to pass parameters from one to the other, developers are required to manually pass input arguments one by one to enable marshaling by the Android OS. Alternative means of passing data, for example, through static variables, are not recommended and considered bad practice, as they do not support preemptive resource destruction from the OS.

MobiCOP fully embraces this philosophy and offers both service-encapsulated core implementation to optimize resource consumption and a similar interface for offloading tasks. This offers us two major advantages: first, it allows us to provide developers with an API evoking a certain sense of familiarity with the standard Android SDK; second, it allows us to overcome one of the biggest challenges in offloading frameworks: state transfer. Ever since MAUI, state transfer has been recognized as the biggest challenge in the development of code offloading frameworks. A common technique for addressing this concern involves automatic reachable heap calculation [16]: the entire heap that is reachable from the offloaded task is calculated and sent to the cloud. Unfortunately, this requires a significant amount of data to be sent across the web, a characteristic that is advised against in the context of unreliable and expensive mobile networks. Some researchers have suggested different means for limiting the state to be transferred, for example, through static analysis of the relevant executable code [21] or via the implementation of distributed shared memory (DSM) [18]. However, these alternatives are far from optimal as they require significant amounts of either processing or network traffic.

Our implementation takes a step back when compared to traditional offloading frameworks and requires developers to manually pass relevant input to the tasks that would be offloaded. While disadvantageous at first glance, this approach is the same that Android developers are accustomed to when implementing data sharing between Android components, thus cohering very naturally. With this, the data to be transferred between local and server environments can be both immediately determined and also kept at a minimum.

The platform's interface is also deeply integrated with our decision-making engine. By passing data to the framework as a hash map, all the input data is tagged by default. This information can then be used to make accurate predictions of a task's execution time based on past evidence, assuming the task is deterministic. This, coupled with a context and network profiler in charge of evaluating the network's status and throughput, allows us to make educated decisions about the best context for executing a given task for optimizing both performance and battery usage.

One final feature supported by our platform is the ability to support concurrent execution and offloading of tasks. Although our context and network profiler is capable of properly evaluating the status of the network at any given time, it is impossible to ensure that the quality of the network will remain the same across the entire lifecycle of an offloaded operation. Once an operation is successfully transferred to the server, it may be that its eventual output will be unable to reach the client due to a sudden drop in network availability. By default, our framework recognizes these scenarios via a timeout and falls back to local implementation if a result fails to reach the host application in time; nevertheless, under such circumstances, delays will be inevitable. It can therefore be of interest to developers to simultaneously run a task locally and on a server, in order to ensure the best possible performance will be attained. That way, the client will retain the result from the most efficient context, despite network quality issues, although battery consumption is expected to slightly increase. Tango is one of the most prominent code offloading frameworks to feature concurrent task executions [5]. One more use case where this is relevant is when handling the cold start of new app installs. Our decision-making engine is based on past evidence in order to make decisions on the most appropriate context to run a task. If said evidence does not exist (either due to a fresh install or the end-user deleting the application's local data), it becomes impossible to return an educated guess. Under these circumstances, our framework will always default to a concurrent execution, both to guarantee a reasonable performance and to start acquiring the data needed for future decisions. From here onwards, we will refer to a workflow in which a task is run in a single context based on the output of the decision-making engine as *optimistic* and to one in which it is run simultaneously on both of them as *concurrent*.

3.3. Architecture Overview. The essential part of the client's library revolves around a core Android service that oversees the handling of incoming tasks and interconnects the platform's various components. Whenever the user dispatches a task to this service, it queries the decision-making engine to decide on the proper execution context and whether the task should be executed concurrently or not. An overview of MobiCOP's workflow can be seen in Figure 1, and a full rundown of the architecture is available in Figure 2.

MobiCOP's communication layer was built with mobility in mind and designed to minimize traffic and power consumption under unreliable network conditions. As such, it makes use of asynchronous communication and resumable data transfers to minimize problems stemming from payload transfers interrupted midway and premature socket closure. MobiCOP defines two communication channels, one dedicated exclusively to the transfer of control messages for coordinating the state of an offloaded task and a second one optimized for the transfer of large payloads. In order to minimize resource consumption, it was decided to base the transfer of control messages on the Google Firebase upstream messaging technology. Upstream messaging is an XMPP communication channel that allows clients to reuse the same sockets Android uses for the handling of push notifications and relay small payloads to arbitrary recipients through Google's infrastructure. Its primary benefits include socket reuse for improved battery life and resource optimization thanks to the asynchronous nature of XMPP.

On the other hand, to ensure the best possible performance and reliability of the transferring of large payloads, it was decided to build the second channel on top of HTTP-based resumable data transfer technologies. The main reason for this choice was to allow applications to resume transfers from the point they were disconnected in the case of network interruptions. We decided against the construction of a custom protocol in order to ensure MobiCOP's availability in public environments with networks configured to restrict certain ports (e.g., schools, airports). For uploads, our solution makes use of the TUS protocol for resumable uploads. TUS is an open-source HTTP-based communication protocol meant to provide a unified solution for handling the problem of midway connection failures. It provides a robust framework for dividing large payloads into smaller packages, assembling them in the server once all packages have been sent, and error handling. For downloads, we made use of the Range metadata defined in HTTP, which allows static file servers to resume downloads starting from an arbitrary offset defined by the client. For both of these solutions to work, all large payloads must first be recorded

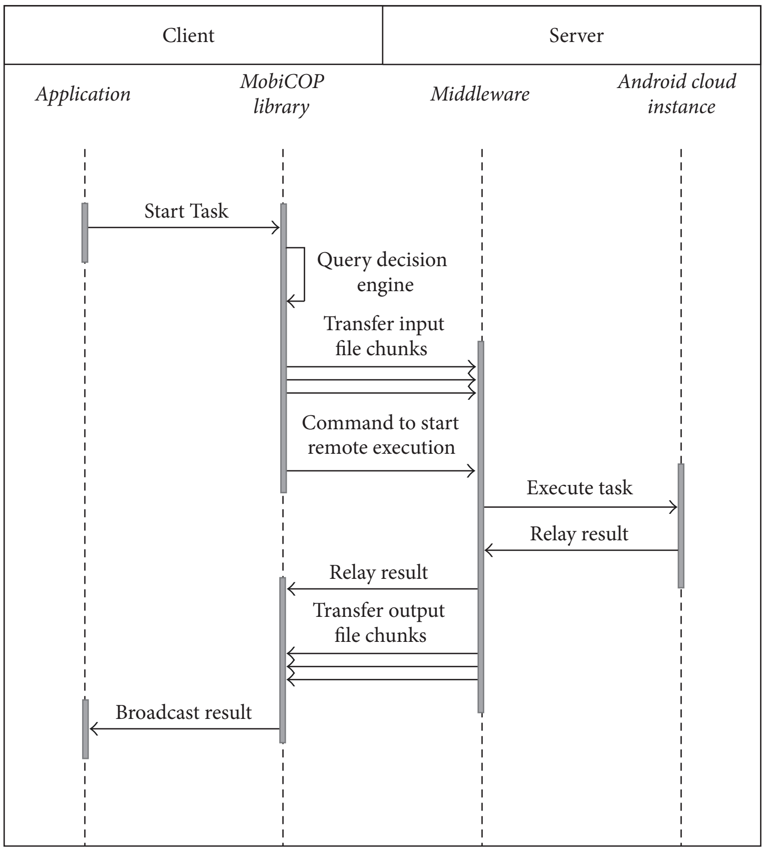


Figure 1: MobiCOP offloading sequence diagram.

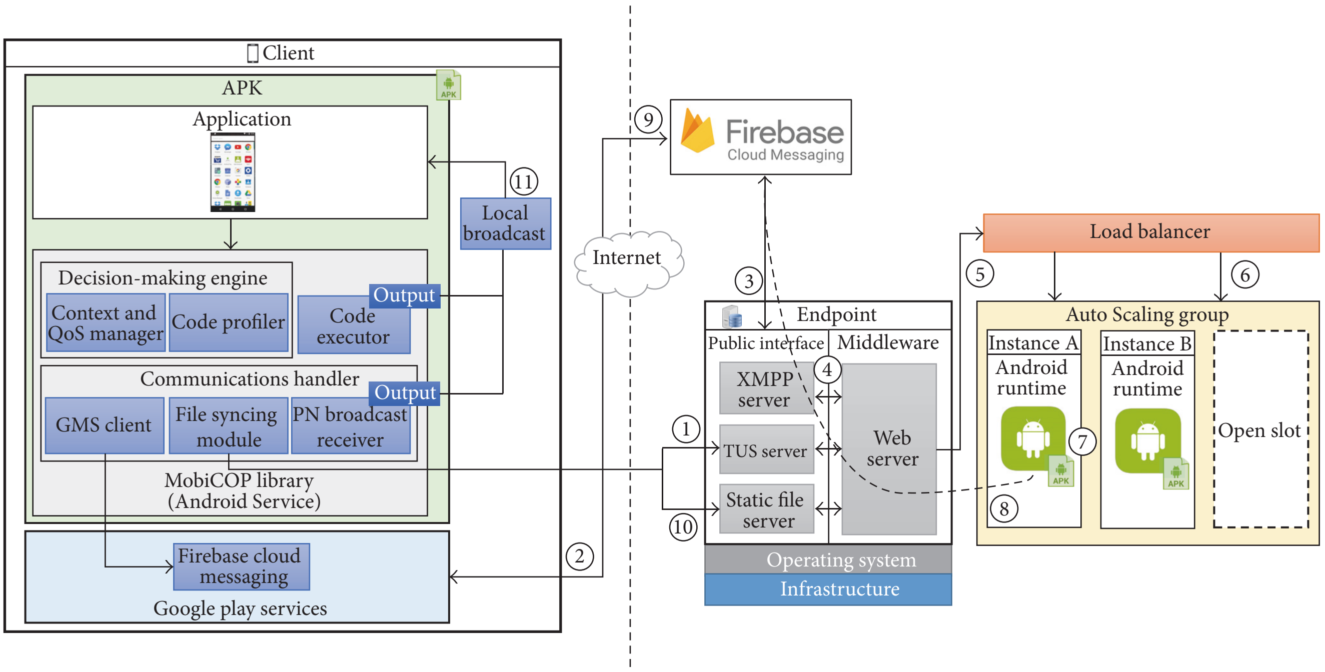
onto disk as a file. This has two primary benefits: first, it allows streaming from disk instead of memory, therefore reducing an application's memory footprint and risk of "out-of-memory" errors; second, it allows deferring the processing of results, a useful feature to have for noncritical tasks running when the end-user is in the middle of a different activity.

When it comes to differentiating between when to use which channel, we take into consideration the 4 KB limit imposed by Google for messages transferred via upstream messaging. If the combined size of a task's input set plus its associated metadata defined by MobiCOP does not exceed 4 KB, all data is sent through the first channel alone; otherwise, the server/client is instructed to wait for the associated input/output data to be sent through the second channel before proceeding.

MobiCOP's server component defines four distinct modules: a public interface, middleware, an elastic cloud component, and an x86-compatible Android server environment. The public interface exposes the services that clients may interact with: an XMPP endpoint hooked up to Google's infrastructure to receive upstream messages, a TUS server for managing the resumable upload of large payloads, and a static file server that conforms to the Range HTTP header specification for handling resumable downloads of large payloads. Offloading requests are passed to the middleware in charge of managing the status of pending tasks and sanity checking the Android execution environments. The latter are currently based on "Genymotion on Demand," a provider of virtual x86 compatible Android environments built on top of Amazon Web Services' infrastructure.

Several alternatives were considered for running Android in the cloud, including running our own custom server with Android x86 and Ravello Systems' Android emulation technology [22], yet Genymotion managed to significantly outperform the others in our various performance tests. Genymotion on Demand also gives us the added benefit of enabling integration with other AWS services, such as AWS Auto Scaling. The latter is an integral part of our architecture as it allows us to scale the number of Android virtual machines to adapt to the number of offloading requests being received at any given time. Moreover, spare instances are removed once traffic slows down, which allows us to save on monetary costs. Additional benefits of Auto Scaling include improved fault tolerance, as Auto Scaling automatically checks the status of currently running instances and reboots them in case they become unresponsive, and improved availability, since Auto Scaling creates additional instances in geographical regions where network traffic is busier.

Developers are required to preinstall their application's APK into the Genymotion on Demand Android virtual machines for the platform to be operational. This allows us to replicate the exact same logic as that of the clients without the need of transferring executable code or its dependencies.



(1) Large payload transfer through TUS-based resumable upload file syncing
(2) Offloading request via Firebase upstream messaging
(3) Firebase relays the message back to our server
(4) XMPP interface relays message to middleware
(5) Middleware sends message to active Android runtime or may request a new server instance
(6) Load balancer decides, based on current workload, to either relay the offloading request to an active instance or start a new one
(7) Task is executed server-wise
(8) Cleanup is performed and response is sent back to the client via Firebase downstream messaging
(9) Firebase relays the message back to the client
(10) Large payload transfer via resumable download based file syncing
(11) Response is sent back for processing via a local broadcast

Figure 2: MobiCOP architecture overview.

Once a task is completed in the server, its result is sent back to the client via Firebase push notification. If the task's output exceeds 4 KB, the data is stored in a file and relayed through the static file server. The moment the client receives the data back, the task's results are forwarded to the application via an Android broadcast and later captured by an Android broadcast receiver that defines how to handle the results. One major consequence of our framework's design is that it forces developers to code in a thread-safe manner. Accordingly, MobiCOP does not support midtask client-server communication (only lightweight status notifications from the server to the client in order to support simple features such as progress bars) or intercontext synchronization (e.g., lock sharing). This is intentional, as such features are prone to concurrency errors and modern mobile APIs have been abstracting them for years.

In summary, a standard MobiCOP offloading workflow operates as follows: whenever MobiCOP's decision-making engine decides to offload a task, the size of its input data is measured. If it is above 4 KB in size, the file-syncing module is activated and the input parameters are transferred to the server via TUS (Figure 2, step (1)). Otherwise, this step is skipped and we proceed to contacting the server via FCM (Figure 2, steps (2) and (3)). Once the server receives a request to offload code (Figure 2, step (4)), the middleware application logs the entry (Figure 2, step (5)) and forwards it to the AWS Auto Scaling module (Figure 2, step (6)). This module will then relay the message to an operational Android runtime built on top of Genymotion on Demand (Figure 2, step (7)) and will also spawn additional instances if the server load becomes too high or terminate some of them in the opposite case. Once the task is completed, a message is sent back via push notifications using Firebase downstream messaging (Figure 2, step (8)). Here, it is possible for the message to be temporarily withheld by Google if the client cannot be reached. This can happen if the client's connection is lost or if the user shuts down the device. In that case, FCM will wait until the device is reachable again before forwarding the results (Figure 2, step (9)). When the client receives the message, the file-syncing module may again require retrieving additional data from the static file server if the output data did not fit inside the FCM package (Figure 2, step (10)). Again, if this is not the case, this step is skipped. Finally, once all results are available on the client, the framework emits a broadcast to notify the application that the operation is completed (Figure 2, step (11)).

3.4. Application Programming Interface. Tasks to be offloaded are encapsulated in a runnable derived from a superclass in the API (Figure 3). This class provides utility methods for accessing the Android context and posting partial results. It defines a single abstract method that needs to be

```
/**
 * Encapsulates any execution to be run either locally or
 * in the cloud.
 */
public class OffloadingCodeRunnable
        extends CloudRunnable {
    /**
     * Code to be executed.
     *
     *@param params A set of parameters supplied by
     *                  the developer.
     *@param lastState The last recorded state managed
     *                  to be sent by the cloud service.
     *                  This is used in case the code
     *                  execution is resumed locally after
     *                  being offloaded. This may happen
     *                  if the connection is interrupted
     *                  before obtaining the final
     *                  result. May be null.
     *@return A result encapsulated in a Params object.
     */
    @Override
    public Params execute(Params params,
                          Params lastState) {
        // Do stuff...

        Params result = new Params();
        // Populate result object
        // (e.g: result.putString("key", "value");
        return result;
    }
}
```

Figure 3: MobiCOP API sample: how to define code that may be offloaded.

```
private void offloadCode() {
    Params bundle = new Params();
    bundle.putString("key", "value");

    CloudOperation operation = new CloudOperation(
            this, OffloadingCodeRunnable.class);
    operation.setParams(bundle);
    String operationId =
          CloudManager.executeCloudOperation(
                this, operation);

    // Operation id may be used to identify the offloaded
    // operation when the results are received back in a
    // broadcast receiver
}
```

Figure 4: MobiCOP API sample: executing an offloaded operation.

implemented by the developers. Here, they may include any custom logic they have in mind. Input and output parameters are encapsulated in a generic object facilitated by the API that includes binary serialization logic to marshal the parameters from the mobile client to the cloud transparently. This object may be populated by a dictionary of arbitrary primitives and their corresponding arrays, adopting a pattern very similar to Android's intents. In order to transfer data between client and server, all parameters are serialized using Google's protocol buffers, a lightweight platform-neutral binary serialization technology.

Once this class is defined, users simply need to declare an input set and pass the pair (input, runnable) to the framework (Figure 4). This will in turn immediately yield an operation ID to the developer which may be used to associate future results to their respective tasks. If in optimistic mode, the library will then decide whether to run the code locally or to offload it, with the decision being transparent to the developer; if in concurrent mode, the task will be simultaneously run on both contexts unless network conditions impede offloading. In the latter case, a synchronization mechanism ensures only the first received result for a specific operation is relayed back to the application. In case the client finishes before the result from the server is received, the latter will be omitted. Otherwise, the client's result will be omitted. In this last case, it is important to highlight that running operations will not be forcefully aborted in order to prevent a potential failure to clean up resources. Instead, a thread interrupted flag will be triggered which developers are encouraged to check in order to prematurely terminate an operation.

Finally, the results are received by a local broadcast receiver. The decision to use this Android feature to collect results was based on the fact that users may exit the application before the task is finished. Should they do so, a broadcast receiver allows the results to be processed even if the UI is not visible or if the application has already been terminated by the operating system. If declared in the Android manifest, it even allows receiving and processing results after a reboot.

3.5. Decision-Making Engine. MobiCOP's decision-making engine is made up of two modules: a quality of service (QoS) monitor and a code profiler. The QoS component is in charge of profiling the network, that is, keeping track of its availability, latency L, and transfer speed R. For Wi-Fi connections, throughput speeds are sampled the moment the device connects to a new network, then periodically at regular intervals. Mobile networks on the other hand are slightly more involved. We decided on seeking an alternative approach in this case, in order to avoid generating unnecessary costs to the end-user. As such, the network speed is estimated based on documented average transmission rates of the subtype of the connection (LTE, GPRS, EDGE, etc.).

The code profiler is responsible for keeping track of past task executions and predicting the running time of future tasks. This module's operation is based on the heuristic that two tasks sharing the same logic and input parameters will take the same time to be completed. Its implementation was rendered possible thanks to the particularities of our programming interface. 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:

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

Let $\widehat{V}$ 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

$R_T = (t_{\text{local}}, t_{\text{cloud}}, l_{\text{output}})$ is stored in a KD-Tree by the client, with $\widehat{V}$ as its search key. Once enough data has been recorded, the decision-making engine can make accurate predictions for newly arriving tasks. Let T_N be a newly arriving task. The decision-making engine executes a nearest neighbor algorithm to find those records R_{T_i} whose vector $\widehat{V}_i$ most resemble $\widehat{V}_N$. Let W_i be the inverse of the distance between $\widehat{V}_i$ and $\widehat{V}_N$; we can then make an estimate for R_{T_N} as

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

Our algorithm has so far shown good results in our tests, but it does have limitations when dealing with nondeterministic algorithms and vectors with high dimensionality. Also, we have had to cap the size of the KD-Trees to prevent too much overhead from being accumulated. Using both the quality of service manager and the code profiler subcomponents, the decision to offload is made if both of the following are true:

(i) A good network connection is available (Wi-Fi: RSSI ≥ -82 dBm; mobile networks: ASU ≥ 5).

(ii) The following inequality is true:

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

where the value α is an arbitrary multiplier currently set to 1.5 by default, as per the model recommended in [23].

4. Results and Discussion

With the goal of proving production-readiness, MobiCOP has undergone testing in a wide variety of scenarios. These include microbenchmarks for comparison purposes with other solutions on both stable and unstable network connections, server stress testing to evaluate the robustness of our backend, and macrobenchmarks in the form of integration with actual commercial applications. Testing was done on both low-end and high-end stock Android mobile devices. For low-end, we used a Lenovo A319 with a dual-core 1.3 GHz Cortex-A7 CPU and 512 MB of RAM; for high-end, we used a Samsung Galaxy S5 with a quad-core 2.5 GHz Krait 400 CPU and 2 GB of RAM. The server component was deployed on an AWS m4.xlarge instance running Genymotion on Demand with Android 6.

In all of our experiments we measured two metrics: performance and energy. For measuring battery consumption, we used a Monsoon Power Monitor device. For all benchmarks, we compared both metrics when performing a local execution and when offloading the task. The potential of mobile code offloading technologies depends largely on the mobile network technology they are using, as it affects both throughput and energy [24]. Therefore, we considered offloading under both a Wi-Fi connection and a 3G mobile network.

Finally, we evaluated ease of use and adoption by handing a MobiCOP-based mobile cloud computing assignment to an advanced computer science engineering university course and asking the students for their opinion on the platform.

Overall, for all of our experiments, offloading via MobiCOP resulted in significant gains in both performance and energy when compared to a local execution. Scalability testing and ease of use evaluation were also positive.

4.1. Microbenchmarks

4.1. Microbenchmarks. The microbenchmarks used for evaluating MobiCOP were carefully selected to represent the most common likely use cases for MobiCOP. The most common scenario is computing a standard Java algorithm. However, as MobiCOP mainly focuses on CPU or GPU-intensive long-running tasks, we gave special attention to Android's technologies designed specifically for computation-intensive applications. With this in mind, we have included microbenchmarks for algorithms exploiting Android's NDK (which enables interacting with optimized C/C++ code) and Renderscript (framework for concurrent CPU, GPU, and DSP computation). It is important to highlight how pivotal it is for offloading frameworks to support these technologies as modern commercial applications making use of intensive computation are highly likely to take advantage of at least one of these. We also consider a microbenchmark for an application requiring large amounts of state transfer between client and server to evaluate behavior under high network traffic conditions. For each of these four categories, we consider the following specific benchmarks.

(i) Pure Java Computation: N-Queen Problem ($N = 14$). The task is to enumerate the placements of all N valid queens on an $N \times N$ chess board such that no queen is in range of another.

(ii) NDK Computation: Chess Engine (8 Plies). The task is to calculate the optimal move in a game of chess with the computer taking into consideration up to 8 possible moves in advance. The chess engine selected for this microbenchmark was implemented in C++ and interfaced through the Android NDK.

(iii) Renderscript Computation: Mandelbrot Fractal. The task consists of generating a Mandelbrot fractal through an algorithm implemented in Renderscript and visualizing it in a 3000 × 3000 image.

(iv) Computation Involving Heavy Traffic: Video Transcoding. The task consists of transcoding a WebM video into an MP4 equivalent using FFMPEG. The video used in this experiment is 48 seconds long and 4.5 MB in size, and it would need to be transferred in its entirety both at the beginning and at the end of the offloading task.

All of these benchmarks were based on open-source implementation available online. Figure 5 shows the results under stable network conditions.

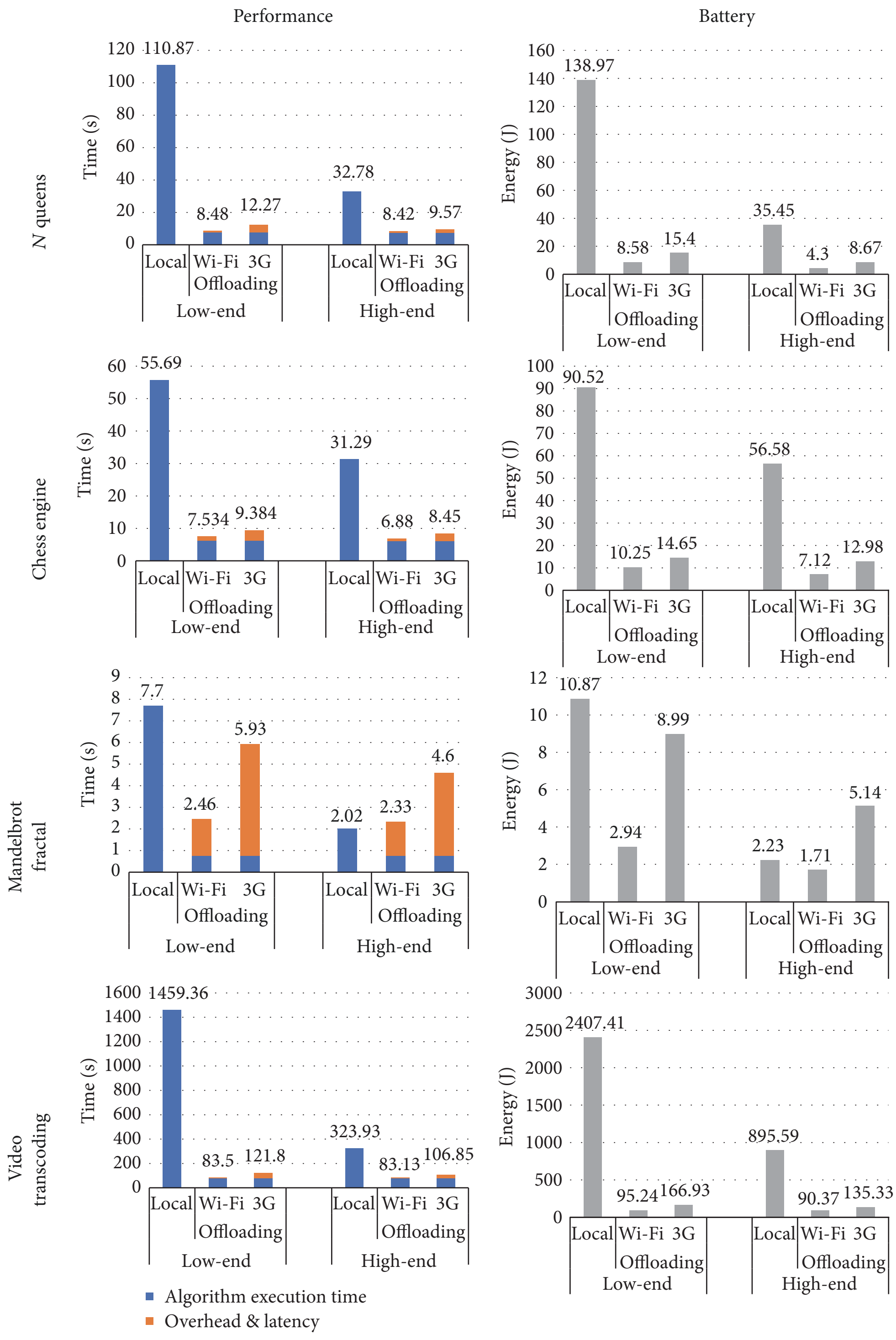


FIGURE 5: Microbenchmark results for stable network environments.

Overall, all metrics show substantial improvements when offloading under either Wi-Fi or 3G mobile networks. The only exception is the overall execution time and power requirements for the Renderscript benchmark in high-end devices. The primary reason for this is that the task itself is too short when compared to the time it takes to transfer the output file back to the client. In general, the longer the offloaded task takes to be completed, the more pronounced the MobiCOP's benefits become.

It is also important to demonstrate that MobiCOP performs adequately under unreliable connections as well. It is difficult to set up a reproducible experiment for unreliable networks in real-life environments; therefore, we decided to emulate an unstable connection by routing our mobile

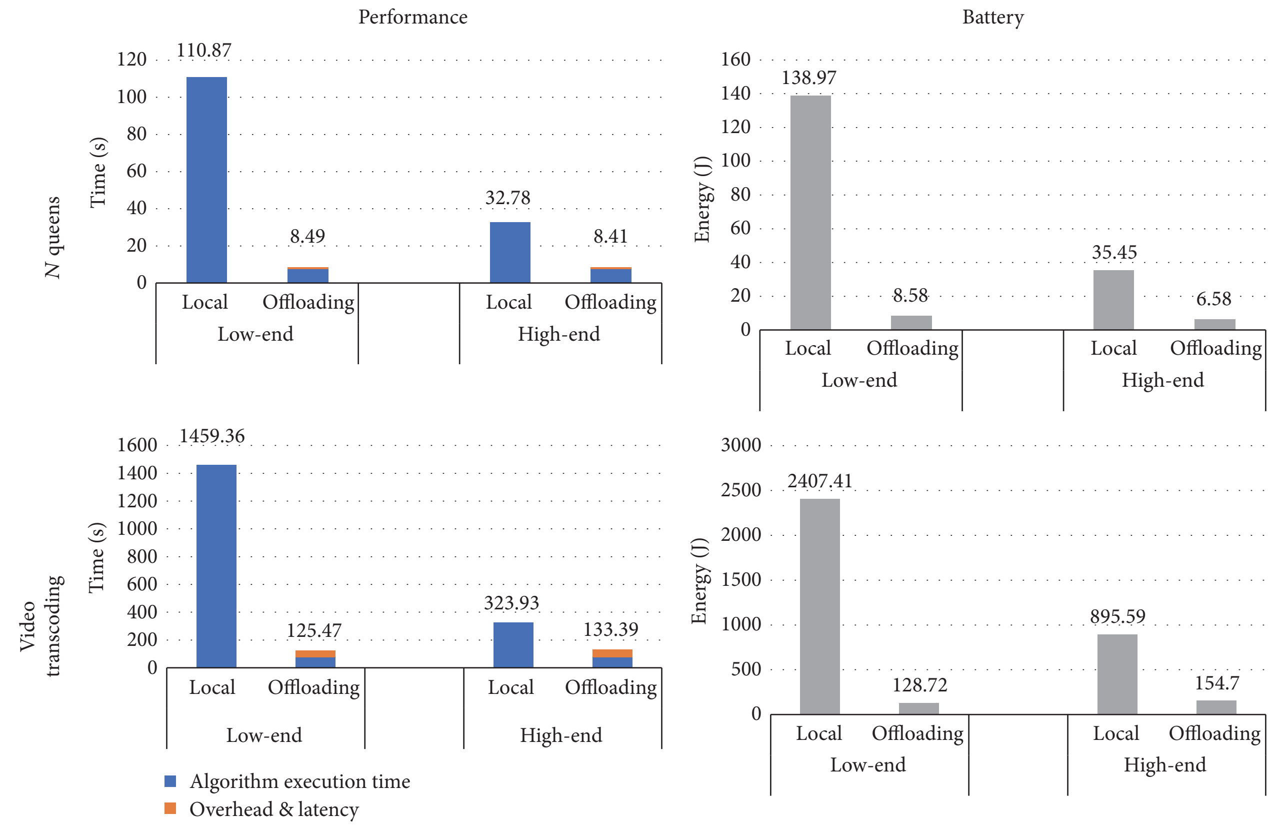


FIGURE 6: Microbenchmark results for unstable network environments.

device's network connection through a proxy equipped with network throttling capabilities. For this experiment, we used the Charles Proxy software and configured it to emulate a slow 3G connection with a package transfer failure rate of 1% per each 10 KB transmitted. When transferring files several megabytes large, this actually results in rather frequent disconnections. For this reason, this experiment is particularly relevant to computation requiring large amounts of data to be transferred between client and server. As such, we only show the results for one benchmark involving little data exchange (N queens) and one involving large amounts of data exchange (video transcoding). The results are shown in Figure 6.

There is little change in all metrics in the situation where the required state to transfer is low. Differences become much more noticeable when the payloads are large in size, as in the video transcoding benchmark. While battery and performance gains do indeed become slightly more modest because of the increased latency and transfer times, MobiCOP still manages to outperform a local execution by a fair margin in both low-end and high-end devices.

4.2. Scalability Testing. As previously mentioned, MobiCOP was designed to address the scalability issues commonly found in current code offloading solutions. In order to prove scalability, we performed three stress test experiments in which we sent varying amounts of offloading requests for an N-queen problem ($N = 15$ for this experiment) across a 30-minute time frame. To emulate the irregular rate at which requests are received in a practical setting, we used a Poisson process with rates of 3, 6, and 12 requests per minute. Auto Scaling instance replication policies may be fine-tuned according to the specific needs of the developer; however, in our case, we set up our environment to double the number of server instances whenever the overall CPU work load across all instances would exceed 50% and to terminate one-quarter of the instances whenever the CPU workload would go below 25%.

Figure 7 shows the results obtained from these tests. The charts on the left show the evolution of the number of Android instances throughout the tests. The charts on the right show the evolution of the average response time for intervals where the number of Android instances was constant. As a point of reference, the response time for an isolated request is around 53 seconds.

From the evaluation, we can identify three scenarios. The first (3 requests per minute) is a case in which the demand is adjusted to the initial conditions of the cloud platform. In fact, no additional server instances running Android are needed, and the average response time is kept stable throughout the execution of the test. The second scenario (6 requests per second) represents a case in which the demand is slightly too high for the initial conditions of the cloud platform. Therefore, the response times are initially subpar. As the platform starts to increase the number of Android instances, the response times start to converge toward the expected values from the first test. Finally, the

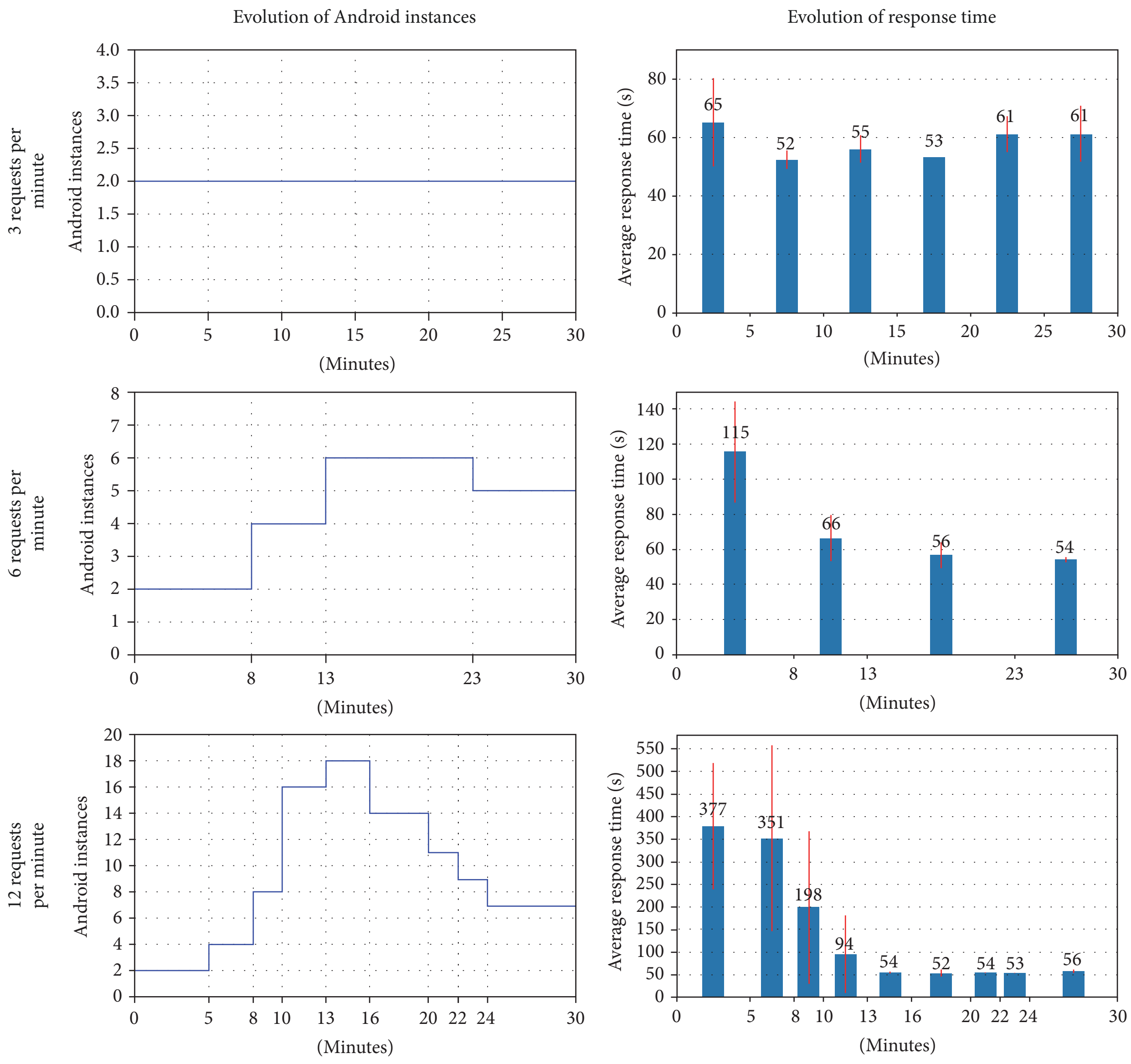


Figure 7: MobiCOP stress test results.

third scenario (12 requests per minute) is a case in which the demand totally collapses the cloud platform given its initial conditions. As such, for a limited amount of time, server performance becomes very lackluster. In order to be able to satisfy this traffic, the platform aggressively scales the number of Android instances until the servers recover their capacity to process them efficiently. After that, the platform starts to decrease the number of Android instances until the point where the demand is in equilibrium with the capacity of the platform and the response times stabilize at their expected values. In this last scenario, it takes our server about 13 minutes to adapt itself to provide an optimal response time for the offloading request. Nevertheless, we must highlight that it is possible to further decrease this interval by configuring Auto Scaling to increase the number of server instances when under high workload conditions by a factor greater than two.

In addition to the evolution of response time, we used a second metric, the throughput, to evaluate the scalability of our solution. The throughput can be defined as the rate at which requests are processed by the system. According to Little's law, we can calculate the throughput of the system as follows:

$$X = \frac{N}{R}, \tag{4}$$

where N is the average number of transactions being processed, R is the average response time, and X is the throughput of the system [25]. Table 2 shows the results obtained in our experiments.

From the table, we can see that, for a relatively low quantity of concurrent requests, the throughput increases linearly. However, given the nature of the processes (high CPU usage for almost a minute for an isolated request),

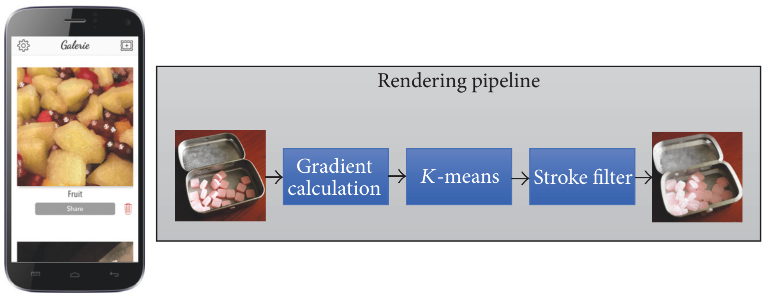


Figure 8: Galerie's rendering pipeline.

Table 2: Throughput of the system for the N-queen experiment.

Concurrent requests	Android instances no.	Average response time (s)	Throughput (requests per minute)
3	2	55	3.27
6	2	65.8	5.47
12	2	152.3	4.73
24	2	283.2	5.08

we can see that the system quickly reaches a cap on its throughput at 6 concurrent requests. Additional concurrent requests rapidly degrade the throughput of the system if the number of Android instances remains constant. Nevertheless, the ability of our system to autoscale on demand ensures a high throughput is maintained by automatically instantiating additional Android instances on the cloud when needed.

We can observe from the results that MobiCOP can adapt itself to different levels of demand, increasing or decreasing the number of Android instances according to its needs. It is important to note that the performance of the scalability policies for a particular application could be improved by making a more detailed analysis of MobiCOP's intended use cases.

4.3. Practical Use Cases. MobiCOP was designed with the intent of integrating it with production-ready applications. It is therefore important to demonstrate that it can be successfully integrated into industrial level software with complex workflows and use cases. With this in mind, we will now show two macrobenchmark examples of professional applications that have been able to reap benefits from MobiCOP.

4.3.1. Galerie. Galerie is an image processing application that makes use of machine learning techniques to learn the patterns of a specific painter's styles and apply them to an arbitrary image in order to redraw them and obtain a painting-like equivalent.

Galerie's algorithm is based on three consecutive steps (Figure 8). First, a color gradient is calculated for each pixel in the input image. Then, gradients are grouped into clusters with the K-means algorithm. Finally, a filter is applied to imitate a painter's brush strokes. Galerie was designed to be operational in offline settings; therefore all of the processing is done locally. For performance reasons, Galerie takes full advantage of OpenGL to harness the device's GPU, in addition to the CPU, in order to increase the algorithm's processing speed. As such, the image transformation is very CPU- and GPU-intensive, so it takes a couple of minutes to be completed (even more on low-end devices), during which a significant amount of battery is drained.

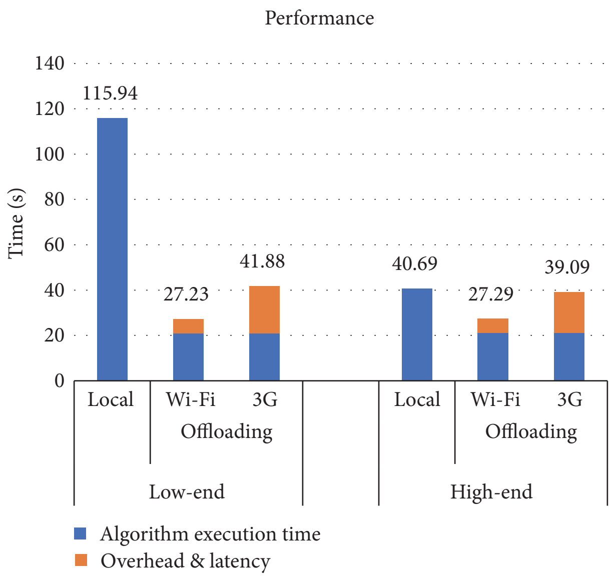
MobiCOP presented itself as an excellent opportunity to grant Galerie the ability to increase performance and battery life in settings where a network is available, without having to sacrifice offline support (although in this case MobiCOP would not grant any additional benefit). Moreover, MobiCOP allows this to be achieved without having to rewrite the algorithm's complex logic in a server environment. Results when running Galerie for transforming a 748 × 748 image 1.3 MB in size, with and without MobiCOP, are shown in Figure 9.

MobiCOP gave Galerie excellent performance and battery gains for both low-end and high-end devices. Although the performance gains are less noticeable for higher-end devices, particularly those with a 3G connection, the reduction in battery consumption remains significant due to the device remaining idle during the operation.

Considering that Galerie makes intensive use of GPU and the fact that Android instances on AWS do not have physical support of GPUs (they are emulated by CPU), we were also interested in how this fact could impact our platform. We also took separate measurements of the execution time of the K-means (CPU-intensive) and the stroke filter (GPU-intensive) processes when executing the algorithm locally (both devices have a physical GPU) and when offloading the execution. The results are shown in Figure 10.

The results show that, while the relative gains for the GPU process are less than the gains for the CPU process, the GPU process is still faster when it is offloaded. From these results, we conclude that even though access to a physical GPU is restricted for Android instances in AWS, tasks are still executed faster on the server. Nevertheless, should AWS eventually add GPU support for Android, we expect results to become even more promising.

4.3.2. Contect. Contect is an early warning medical application that seeks to detect possible neurological complications (e.g., concussions, traumatic brain injuries) by analyzing a patient's speech [26, 27]. Specifically, it asks the patient to



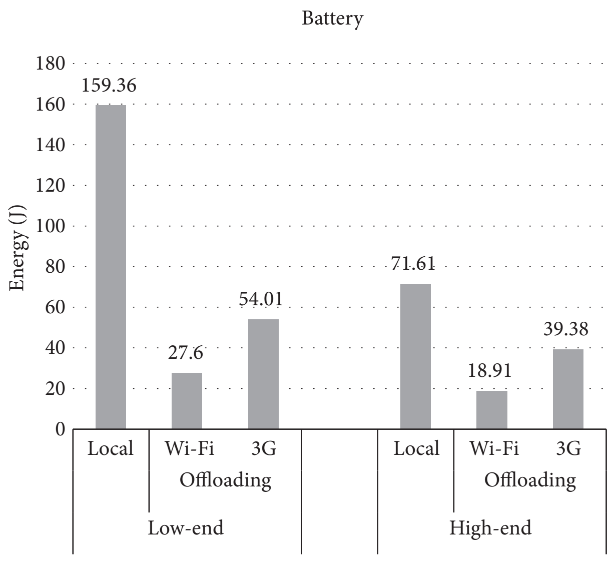


FIGURE 9: Galerie performance and battery results.

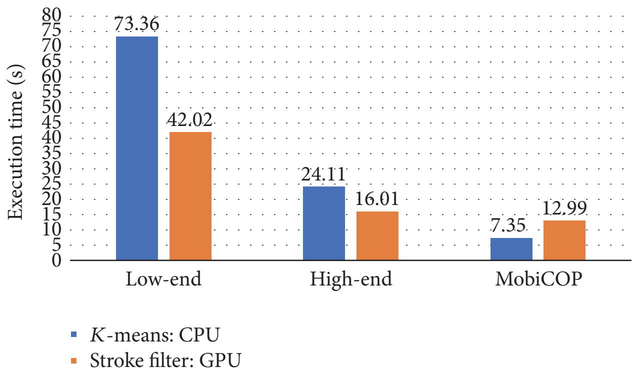


FIGURE 10: Comparison of relative gains for CPU and GPU processes.

take a sustained vowel test (to say "AHHH" for as long as possible), records the audio, and then feeds the speech sample to a deep learning neural network that processes the sample and delivers a concussion probability. Sustained vowel tests are a common examination requested by speech-language pathologists. Example screenshots of the application are available in Figure 11.

As deep learning algorithms are most suited for image recognition applications, the speech samples are first transformed into spectrograms before being fed to the network. Contect's neural network is incredibly complex as it tries to detect patterns that are very hard for human beings to notice. For this to work, a complex architecture consisting of ten parallel 50-layer deep residual neural networks was built. The outputs are then agglomerated into a series of consecutive fully connected layers before delivering a final result. When offloading with MobiCOP, the spectrogram is sent as input across the network to the server, and the concussion probability is sent back to the client as the operation result.

Due to the complexity of the model, the resulting neural network ended up being around 250 MB in size, requiring over 1 GB of memory to be loaded and executed on a mobile device. This makes it unsuitable for low-end devices that fail to meet these minimum specs, as running this model would result in an out-of-memory error. Even in high-end devices, a single speech sample processing would take over two minutes to be completed, again draining a significant amount of battery.

MobiCOP had two major benefits for Contect. First, it allowed high-end devices to gain both performance and battery efficiency when a network was available. Second, it enabled Contect to become compatible with lower-end devices that fail to satisfy its minimum required specifications, although only in settings with a network connection. Still, restricted operation in specific environments is preferable to none at all, and all of the above was achieved with minimum programming effort and without the need of porting the model to a server-compatible neural network framework. The results are shown in Figure 12.

Thanks to MobiCOP, Contect's performance was boosted by 4 times, its energy consumption was reduced by 6 times, and its target audience was significantly widened by allowing owners of lower-end devices to make use of the application.

4.4. Quasi-Experiments with Mobile Developers.

An aspect that is often disregarded in code offloading frameworks is how difficult it is for third parties to adopt the platform. This is a very important aspect to consider when seeking to confirm any reported results. In order to evaluate the ease of use of MobiCOP, we sought the assistance of the students at the IIC3380 Mobile Platforms Workshop class at Pontificia Universidad Católica de Chile. Students at this class are taught fundamentals and advanced concepts of Android application development and mobile cloud integration. It is

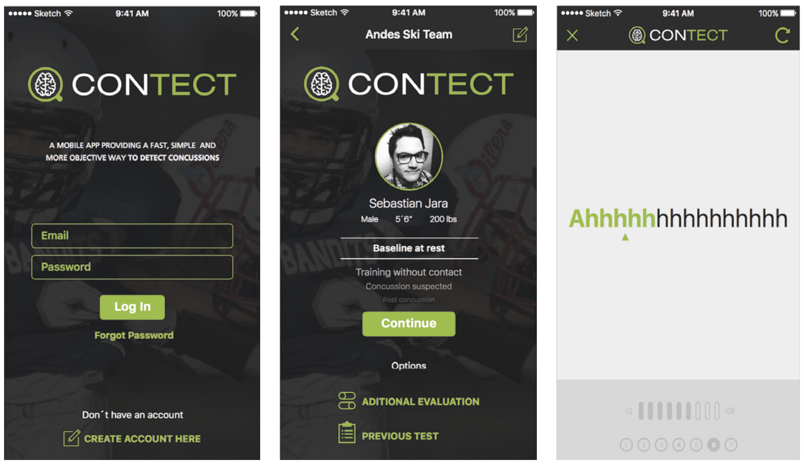


Figure 11: Contect screenshots.

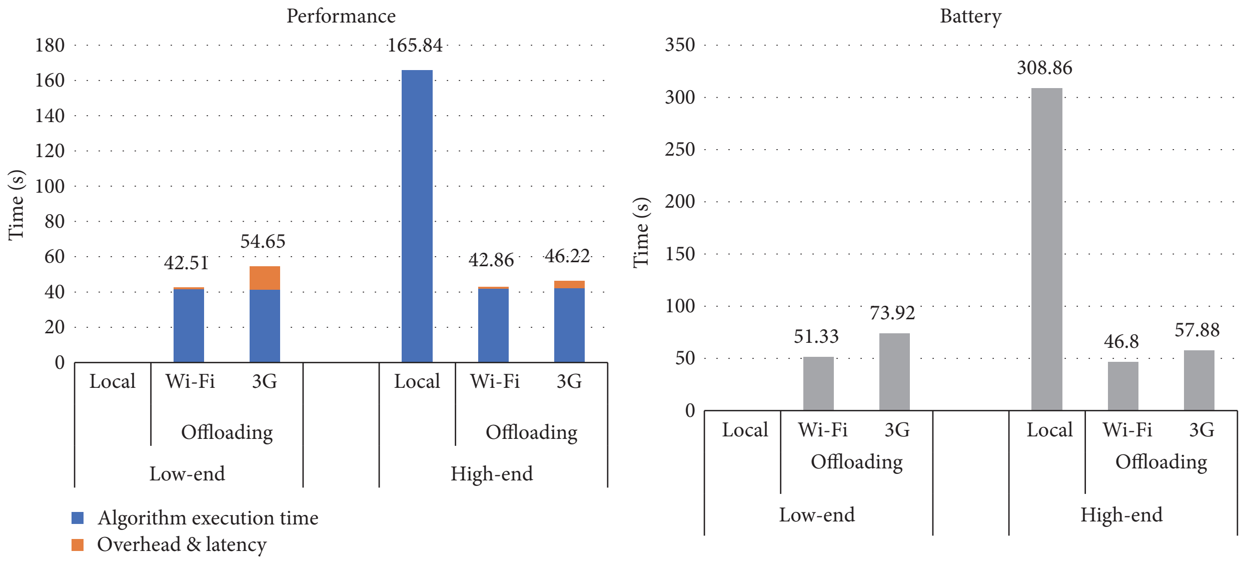


Figure 12: Contect performance and battery results.

an advanced course made up of both graduate students and senior undergrads.

At the 2016 instance of this class, the 14 attending students were split up in four different groups and tasked with replicating both the N-queen problem benchmark and the video transcoding benchmark. Additionally, each group was assigned one of the most representative frameworks of the offloading environment and asked to do the same tasks with them as well. The selected frameworks for this assignment were MAUI, COMET, Tango, and EMCO. Students were expected to compare them with MobiCOP. Due to a lack of documentation regarding these frameworks, students were also encouraged to directly contact the respective authors for assistance. In regard to MobiCOP, students were granted access to a repository holding MobiCOP's code and the N-queen benchmark example. No further help nor documentation was given.

The assignment was handed out at the beginning of the semester and due at the end. Three experienced teaching assistants were available at all times during the workshops to guide the students. At the end of the assignment, all groups were successfully able to reproduce both benchmarks with MobiCOP on their environments with very little difficulty. A survey was then conducted to collect the students' opinion on the respective frameworks, which allowed us to do a qualitative assessment of MobiCOP's developer experience. One of the groups reported the following:

> *MobiCOP's installation was straightforward. [...] The framework's implementation closely resembles how Android intents work, and thus it was very easy to get used to it.*

A single group found that both their assigned framework and MobiCOP shared a similarly low implementation difficulty:

> *MobiCOP abstracts the offloading and creates an interface similar to any asynchronous task that*

depends on function callbacks. [. . .] [Both EMCO and MobiCOP] are easy to include into any app.

Overall, however, the benefits offered by MobiCOP were found to be superior:

Regarding the experiments and benchmarks carried out, EMCO effectively presents several advantages in comparison to local code execution, but is dwarfed in performance when compared to the proposed MobiCOP framework.

Nevertheless, MobiCOP was not devoid of criticism. Lack of documentation and severe hardships with configuring the server were objected to. However, these comments were taken to heart and both issues were solved by developing a series of brief step-by-step tutorials and distributing an AMI on AWS of a fully configured MobiCOP server that allows reproduction in a matter of minutes (this was not available at the time of the experiment and all server infrastructure configurations had to be done by hand).

Beyond EMCO, however, our students found very little success in replicating the other frameworks. Installation instructions were either nonexistent, unclear, or extremely restrictive (i.e., only taking into consideration specific Android models), and not every author answered their requests for assistance. At the end of the semester and despite several attempts, our students were unsuccessful in replicating MAUI, COMET, and Tango. While other authors have managed to use them in their experiments, this indicates that the average code offloading framework requires highly specialized knowledge of the mobile ecosystem to be successfully reproduced.

5. Conclusions

MobiCOP was designed from the ground up to specifically tackle three common deficiencies often found in other mobile code offloading frameworks: reliability, scalability, and limited distribution or replicability. Instead of providing a general method-level offloading framework that ignores the various constraints of mobility, MobiCOP offers a programming model based on standard Android practices and concepts giving developers full control of the task migration process. From this point of view, our approach is similar to that of Cuckoo. Additionally, MobiCOP was designed to be fully compatible with public IaaS providers, allowing total replication without the need for physical access to a server.

Many of the other alternatives available in academia show promise in theory when tested against a single device and for a single set of operations, but little information is offered as to how well they behave when handling multiple offloading requests at once across an extended period of time. Frameworks such as ThinkAir recognize the need to scale rapidly, but few others incorporate this requirement in their design.

The matter of distribution is also commonly omitted. Many of the previous frameworks only work on customized Android versions that impede adoption by the general public. Even if one were to succeed in replicating them on a personal device, they would impose some severe constraints that would put into question their overall benefits. First, by being based on a custom OS version, there is little guarantee regarding security; additionally, security patches, OS updates, and new APIs published for standard Android versions would become unavailable. MobiCOP, on the other hand, offers innovative implementation that is fully contained in a library compatible with all Android versions, API level 17 and above. Integration into any project is as simple as adding a single Gradle dependency. Server replication is equally simple, thanks to the AWS AMI repository.

Overall, MobiCOP's results were very positive across the entire spectrum of tests it was subjected to. Generally speaking, the benefits of code offloading frameworks become more apparent when targeting long-running tasks under stable and high throughput network connectivity. This is still the case for MobiCOP: for a task taking several minutes to complete, it managed to display up to 17 times better performance when compared to a low-end device and up to 25 times less battery life depletion. Although performance differences are less pronounced when comparing against high-end devices, battery gains remain significant, and even outside of ideal network conditions, results remain very promising. In our video transcoding benchmark, MobiCOP still performed 12 times better than a local execution under 3G connectivity, and emulated network interruptions only degraded the results by less than 10%. This is a significant achievement when compared to traditional code offloading solutions that rely on constant full-duplex socket communication, where a network error would immediately trigger a local execution fallback. Instead, our asynchronous messaging system allows for minimum network traffic, all without triggering a significant overhead on the client's limited resources.

Server-side Genymotion on Demand has allowed us to harness the full power of the AWS suite of software tools and further enhance MobiCOP's capabilities. As of today, only AWS EC2 and Auto Scaling have been integrated into MobiCOP. This alone has granted our platform significant robustness with server instance sanity checking, autonomic computing capabilities, and the ability to deploy servers near the geographical location with highest demand. It is expected that integration with additional AWS services, such as RDS or S3, would further increase MobiCOP's capabilities.

Beyond that, the fact that our entire solution is fully contained in a library for the client and an AMI for the server has dramatically increased the target audience for our technology. Our solution does not require any OS customization nor added third-party tool for the build system. MobiCOP just works, and most Android devices (smartphones, tablets, TV, embedded devices, etc.) are compatible with it. Students have so far given us good reviews regarding the approachability of MobiCOP, but further work is still needed to further streamline the developer experience, for example, by adding a better error-reporting system (to guide the user in case of misconfigurations) and by making a formal documentation publically available.

Future work on the MobiCOP platform currently involves three major areas. First, support needs to be added for heterogeneous cloud resources [28]. One major limitation of MobiCOP is that it assumes that all available servers for offloading possess similar hardware characteristics. As such, the decision-making engine presumes that a given task will take the same amount to be completed on the cloud regardless of the server it ends up on. We are currently working on improving the decision-making engine to take this variable into account and on building a custom load balancer that better matches incoming tasks with servers more adequate to handle them.

Second, we are working on adding state-of-the-art context-aware capabilities to the platform by including compatibility with proximate surrogates such as cloudlets and edge servers. This can greatly enhance the overall performance of MobiCOP by allowing proximate servers to deal with offloading requests under poor or nonexistent network conditions, minimize the need for triggering a local fallback after an offloading decision, and significantly reduce latency [29]. In the future, we hope MobiCOP will be able to dynamically select at runtime which nodes it should offload to given several possibilities to choose from.

Finally, we are working on implementing a more robust security protocol on top of MobiCOP to protect the platform from malicious offloading requests. Currently, no authentication mechanism has been implemented, so the server is particularly vulnerable to denial-of-service (DOS) attacks. Further work is needed to minimize this possibility.

With the advent of more and more innovative mobile applications requiring intensive processing capabilities, more and more potential use cases that may benefit from offloading are appearing. To give a few examples, we can highlight the following:

(i) image processing (e.g., object recognition, image filtering, face recognition);

(ii) multimedia processing (e.g., video transcoding, speech recognition);

(iii) text processing (e.g., machine translation, natural language processing, document classification);

(iv) artificial intelligence (e.g., video game AI);

(v) simulations (e.g., Monte Carlo simulations);

(vi) security (e.g., virus scans);

(vii) IoT (e.g., processing of sensor data).

We hope MobiCOP will help to further the offloading trend so that more developers can gain access to this technology and integrate it into their applications to offer better user experiences and battery efficiency.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was partially supported by a DCC-UC research grant, the CONICYT-PCHA/National Ph.D./2016 Grant no. 21161015, and AWS Cloud Credits for Research. The authors would also like to thank the authors of the Galerie application (Juan Antonio Karmy and Felipe Balbontín) and the Contect application (Bryan (Ning) Xia) for granting them access to their code base. Finally, the authors would like to thank all the students from the IIC3380 Mobile Platforms course at Pontificia Universidad Católica de Chile who were involved in this research project.

References

[1] "Cisco visual networking index: global mobile data traffic forecast update, 2015–2020 white paper," http://goo.gl/ylTuVxWhite Paper.

[2] K. Sekar, "Power and thermal challenges in mobile devices," in *Proceedings of the 19th Annual International Conference on Mobile Computing and Networking, MobiCom 2013*, pp. 363–368, USA, October 2013.

[3] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[4] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: can offloading computation save energy?" *The Computer Journal*, vol. 43, no. 4, Article ID 5445167, pp. 51–56, 2010.

[5] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, "Tango: accelerating mobile applications through flip-flop replication," *GetMobile: Mobile Computing and Communications*, vol. 19, no. 3, pp. 10–13, 2015.

[6] J. Shuja, A. Gani, M. H. U. Rehman et al., "Towards native code offloading based MCC frameworks for multimedia applications: a survey," *Journal of Network and Computer Applications*, vol. 75, pp. 335–354, 2016.

[7] J. I. Benedetto, A. Neyem, J. Navon, and G. Valenzuela, "Rethinking the mobile code offloading paradigm: from concept to practice," in *Proceedings of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 63–67, Buenos Aires, Argentina, May 2017.

[8] M. Satyanarayanan, P. Bahl, R. Cáceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[9] K. Gai, M. Qiu, H. Zhao, L. Tao, and Z. Zong, "Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing," *Journal of Network and Computer Applications*, vol. 59, pp. 46–54, 2016.

[10] U. Drolia, R. Martins, J. Tan et al., "The case for mobile edge-clouds," in *Proceedings of the 10th IEEE International Conference on Ubiquitous Intelligence and Computing, UIC 2013 and 10th IEEE International Conference on Autonomic and Trusted Computing, ATC 2013*, pp. 209–215, Italy, December 2013.

[11] A. Ahmed and E. Ahmed, "A survey on mobile edge computing," in *Proceedings of the 10th International Conference on Intelligent Systems and Control, ISCO 2016*, India, January 2016.

[12] E. E. Marinelli, "Hyrax: cloud computing on mobile devices using MapReduce," DTIC Document, 2009.

[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 *Proceedings of the 15th IEEE/ACIS International Conference on*

Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD 2014, USA, July 2014.

[14] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *Mobile Computing, Applications, and Services*, vol. 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 59–79, Springer, Heidelberg, Berlin, Germany, 2012.

[15] E. Cuervoy, A. Balasubramanian, D.-K. Cho et al., "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '10)*, pp. 49–62, New York, NY, USA, June 2010.

[16] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proceedings of the 6th ACM EuroSys Conference on Computer Systems (EuroSys '11)*, pp. 301–314, ACM, April 2011.

[17] 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 *Proceedings of the IEEE INFOCOM*, pp. 945–953, March 2012.

[18] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12*, pp. 93–106, 2012.

[19] H. Flores and S. N. Srirama, "Adaptive code offloading for mobile cloud applications: exploiting fuzzy sets and evidence-based learning," in *Proceedings of the 4th ACM Workshop on Mobile Cloud Computing and Services (MCS '13)*, pp. 9–16, June 2013.

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

[21] Y. Li and W. Gao, "Minimizing context migration in mobile code offload," *IEEE Transactions on Mobile Computing*, vol. 16, no. 4, pp. 1005–1018, 2017.

[22] "How to run the Android Emulator (with Hardware Acceleration) on Amazon EC2 and Google Cloud," https://goo.gl/HnQqiB.

[23] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012*, pp. 233–247, USA, October 2012.

[24] K. Akherfi, M. Gerndt, and H. Harroud, "Mobile cloud computing for computation offloading: issues and challenges," *Applied Computing and Informatics*, 2016.

[25] R. N. Chintalapti, R. Ganesan, and S. A. Wagh, "System for performance and scalability analysis and methods thereof," U.S. Patent No. 7,546,222, 2009.

[26] C. Poellabauer, N. Yadav, L. Daudet, S. L. Schneider, C. Busso, and P. J. Flynn, "Challenges in concussion detection using vocal acoustic biomarkers," *IEEE Access*, vol. 3, pp. 1143–1160, 2015.

[27] L. Daudet, N. Yadav, M. Perez, C. Poellabauer, S. Schneider, and A. Huebner, "Portable mTBI assessment using temporal and frequency analysis of speech," *IEEE Journal of Biomedical and Health Informatics*, vol. 21, no. 2, pp. 496–506, 2017.

[28] Z. Sanaei, S. Abolfazli, A. Gani, and R. Buyya, "Heterogeneity in mobile cloud computing: taxonomy and open challenges," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 369– 392, 2014.

[29] Z. Liu, X. Zeng, W. Huang, J. Lin, X. Chen, and W. Guo, "Framework for context-aware computation offloading in mobile cloud computing," in *Proceedings of the 2016 15th International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 172–177, Fuzhou, China, July 2016.