ULOOF: a User Level Online Offloading Framework for Mobile Edge Computing

José L. D. Neto, Se-Young Yu, Daniel F Macedo, José M. S. Nogueira, Rami Langar, Stefano Secci

HAL Id: hal-01547036

https://hal.sorbonne-universite.fr/hal-01547036v1

Submitted on 26 Jun 2017 (v1), last revised 14 Mar 2018 (v2)

ULOOF: a User Level Online Offloading Framework for Mobile Edge Computing

José L.D. Neto, Se-young Yu, Daniel F. Macedo, José M.S. Nogueira, Rami Langar, Stefano Secci

✦

Abstract—Mobile devices are equipped with limited processing power and battery charge. A mobile computation offloading framework is a software that provides better user experience in terms of computation time and energy consumption, also taking profit from edge computing facilities. This article presents User-Level Online Offloading Framework (ULOOF), a lightweight and efficient framework for mobile computation offloading. ULOOF is equipped with a decision engine that minimizes remote execution overhead, while not requiring any modification in the device's operating system. By means of real experiments with Android systems and simulations using large-scale data from a major cellular network provider, we show that ULOOF can offload up to 73% of computations, and improve the execution time by 50% while at the same time significantly reducing the energy consumption of mobile devices.

1 INTRODUCTION

Mobile applications are expanding beyond our day-to-day activity, and mobile computation is becoming more frequent and intense. According to Chaffey's report [1], both the number of users and the time spent using mobile devices exceeded the desktop use. Users spend at least 15% of their time playing mobile games and another 20% for entertainment that requires intense computation power and energy.

Mobile devices have limited processing power [2] and battery charge [3] by nature. To overcome these limitations, mobile computation offloading solutions were proposed [4], [5], [6] to delegate intensive computation tasks to more capable computing device(s). With the recent advances in Mobile Edge Computing (MEC) [7], computation offloading gains industrial interest after more than a decade of academic research activities.

Computation offloading supports MEC by adjusting where the computation will take place based on the network and user context. Fig. 1 presents the different characteristics of the networks that a user employs throughout the day. Each network has a different sojourn time (represented by the blue circles), and different associated processing and latency capabilities, due to the use of local operator clouds, private cloudlets (also called MEC hosts) and

- *José Leal Domingues Neto is with Google Inc, Belo Horizonte, Brazil. Email: joseleal@google.com.*
- *Se-young Yu and Stefano Secci are with Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6. 4 place Jussieu, 75005 Paris, France. Emails: {se-young.yu, stefano.secci}@upmc.fr*
- *Daniel Fernandes Macedo and José Marcos Silva Nogueira are with the Universidade Federal de Minas Gerais, Belo Horizonte, Brazil. Emails: {damacedo, jmarcos}@dcc.ufmg.br.*
- *Rami Langar is with University Paris-Est Marne la Vallée, France. Email: rami.langar@u-pem.fr*

remote clouds. Cloudlet/MEC deployments are expected to favour computation offloading that have not emerged yet with legacy cloud deployments, thanks to their ability to strongly decrease the access latency because of geographical vicinity between users and servers. In each of those levels, we assume that the owner of the network or an over-the-top service provides VMs running an offloading service.

A core element of any mobile computation offloading framework is the decision engine, since it determines when a task needs to be offloaded to an external (MEC) server. Offloading a task to an external server may incur expensive overhead, hence the offloading decision shall be based on predictions of the energy and time required to offload the processing, as well as the computation time and/or the energy consumption of the computation itself, among other possible metrics.

It is not trivial to predict the execution time and energy consumption of an application method ahead of its execution. An offloading framework addresses these challenges to make efficient offloading decisions and also to provide application developers and/or users a way to integrate their application into the offloading framework. Most offloading frameworks proposed so far predict the available bandwidth or execution time, as done in CloneCloud, MAUI and COSMOS proposals [8], [5], [9], without considering variable wireless network capacity over time. Another common assumption is that the inputs of the computation do not vary much, as in [5], [10], which can lead to imprecise execution time estimation.

This article presents an offloading framework called User-Level Online Offloading Framework (ULOOF). It is equipped with novel algorithms aimed to estimate the execution time and energy consumption of application methods, as well as a location-aware wireless network capacity estimator. This article improves the preliminary work described in [11] by a more detailed description and analysis of the framework, an enhanced decision engine logic in particular in the execution time and energy profiling parts, and by conducting a novel set of experiments and simulations. Our contributions can be summarized as follows.

- We designed and developed a comprehensive mobile offloading framework that does not require neither superuser privileges on the mobile device nor modifications to the underlying operating system.
- We conceived and developed a decision engine that provides accurate execution time and energy consumption estimations to support the offloading decision, while requiring minimum user effort for the code instrumentation.
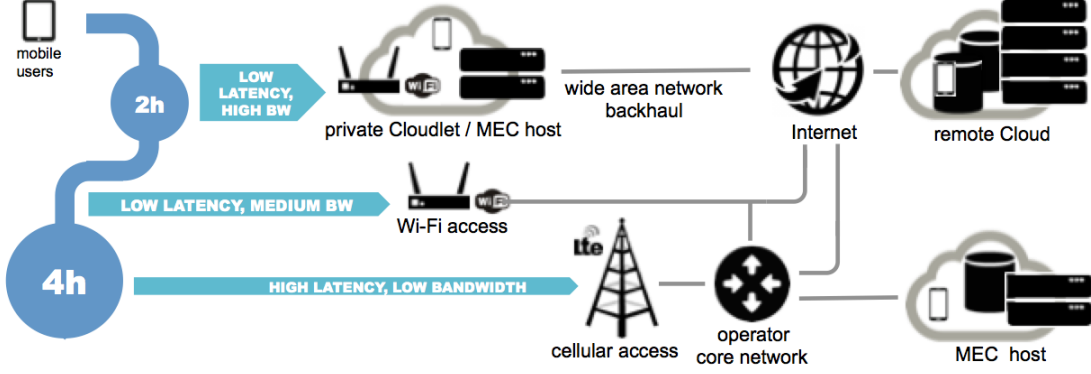
Fig. 1. Mobile computation offloading scenarios

- We evaluated our framework both by simulations using real cellular mobility data and by real-world experiments using a proof-of-concept implementation[1].

The remainder of the article is organized as follows. Section 2 discusses the related work and the state of the art. Section 3 outlies the framework with the different modules involved. Section 4 presents the details of the decision engine, describing the problem model and the prediction algorithms. Section 5 describes the experimental setup used to test the decision engine and discusses the results in depth. Section 6 reports simulation results obtained by processing real large-scale mobility data. Section 7 presents the conclusions and future work. Two appendices provide more details on the energy profiling, and the offloading decision objective trade-off and its effect in ULOOF.

2 RELATED WORK

In this section we review relevant works on mobile computation offloading at the state of the art, highlighting how our User-Level Online Offloading Framework (ULOOF) complements the previous works.

Cuervo et al. [5] propose a framework called MAUI, which focuses on energy saving. It uses a profiler that measures energy consumption and a solver that decides whether to offload or not a method based on the measurements provided by the profiler. The authors evaluate MAUI using three mobile applications, revealing that computation offloading not only saves energy, but also that it allows applications to run faster.

Chun et al. [8] propose CloneCloud, which partitions the application binary with a set of execution points. The execution points are determined so that the resulting partitions are executed in the most efficient execution environment. As a result, Clonecloud can determine the most efficient execution points and execution configuration for each partition.

Kemp et al. [4] propose Cuckoo, a simple offloading framework that always offloads a method when the remote server is available. Cuckoo implements a library to manage the communication between the mobile device and a communication middleware.

Verbelen et al. [12] propose AIOLOS, an offloading framework focusing on class-level offloading using an OSGi framework. They provide an Eclipse IDE plugin that helps developers to build an AIOLOS-enabled application bundle in Android. The bundle is

1. A proof-of-concept application, using a running server at LIP6, is made available in [27], along with a demo video.

executed to update the execution time and to return a size profile. The profile is then used to predict future executions.

Kosta et al. [10] propose ThinkAir. It generates a wrapper for methods to be offloaded so that an execution controller decides whether to offload the method based on execution time, energy and cost. They modelled the execution time and energy using historical data from previous executions. A client handler manages the connection to the remote VM and is also responsible for managing the VM configuration.

Shi et al. [9] propose the COSMOS framework; it determines the benefit of offloading based on argument size, upload bandwidth, result size and download bandwidth using a predefined threshold. The predictions are refined at the end of every execution by adjusting the predicted upload and download bandwidth.

Our proposed offloading framework employs a decision engine that can model execution time and energy consumption accurately. Machine (device) execution time and energy consumption are learned and used to predict the method execution behaviour. Our framework also demands minimal intervention to the normal application development, so neither it requires the developer to understand how the framework operates nor needs modifications/privileges on the Android OS.

Our framework is explained in detail in next section. As a preliminary insight, let us first report via Table 1 how ULOOF is positioned with respect to the above-mentioned existing mobile computation offloading frameworks. A cell is marked with "✓" when the corresponding attributes are featured in the framework, otherwise it is marked with "✗". Intrusiveness corresponds to the level of intervention to the application development to permit a given offloading framework to work; it can be either operating system runtime modifications or modifications in the code and/or development strategies such as programming methods into modules. Some works do not have a decision engine (marked with ✗), choosing always to offload when possible, while others have their own decision engine to decide the execution platform of a specific code. Other parameters are self-explanatory. According to this analysis, MAUI is the most similar proposal to ULOOF, however we were not able to access its implementation therefore we could not perform quantitative comparisons.

Furthermore, there are other types of works focusing on allocating computation resources for offloading and optimizing their allocation. Esteves et al. [13] allocate computation resources using the capital-budgeting technique, typically used in the field of financial option valuation. Such a technique allows to choose the best remote servers among many, based on an offloading

TABLE 1
Comparison between state of the art frameworks and the ULOOF framework

Name	Intrusiviness	Decision Engine	OS/Language	Energy Model	User-Level	Plug-and-play
MAUI [5]	Low	Imprecise prediction	Win/C#	Online	✓	✓
CloneCloud [8]	Runtime modification	Offline instrumentation	Android/Java	Offline	✗	✗
Cuckoo [4]	Development in AIDL	✗	Android/Java	✗	✓	✓
AIOLOS [12]	Development in OSGi	✓	Android/Java	✗	✓	✗
ThinkAir [10]	Runtime modification	Imprecise prediction	Android/Java	Online	✗	✓
COSMOS [9]	Code modification	✓	Android/Java	✗	✓	✗
ULOOF	Low	✓	Android/Java	Online	✓	✓

cost (execution time) and a transmission cost (transmission time). Kristensen et al. [14] approach the resource allocation problem by foraging computation resources from different mobile devices. Mobile devices share their available resource and schedule their work based on their available computation resources. ULOOF is inspired by these resource allocation works, employing a method-level code offloading solution that constantly improves its decisions by learning from previous outputs. Our framework provides computation resource information of both mobile devices and nearby cloud servers to the decision engine, hence allocating computation to the most suitable computing element.

One of the challenging parts of developing a framework for mobile computation offloading is to measure energy consumption of a mobile device. Cignetti et al. [15] provide efficient and accurate energy models for specific models of phones. However, it is difficult to calculate the method energy consumption from a user-level point of view [16], [17], i.e., with the limited system rights and APIs given to a standard mobile application. Zhang et al. [18] provide a general power model using device dependent parameters, and measuring each component energy separately. However, this requires a periodic and active monitoring of the components as a background service, which increases energy usage. Miettinen et al. [19] analyze energy consumption usage of mobile devices to find a comparison between radio and CPU, expressing one part as a function of the other. We complement these works with a non intrusive energy profiling methodology explained in the next sections.

3 ULOOF GENERAL FRAMEWORK

ULOOF is a computation offloading framework that offloads method calls in a user application. Each ULOOF offloading decision is made based on the energy and execution time estimations. Those estimations are updated after every local or remote executions, so that the framework adapts to changes in the execution environment.

ULOOF does not require changes in the operating system, or special user privileges (i.e., without 'rooting' the device). This allows us to modify the application without requiring additional knowledge on the depending Android libraries, moreover it decreases security risks due to rooting and allows users to keep the hardware warranty of the device.

Figure 2 presents a flow diagram of ULOOF. The key elements are:

- The *instrumentation component*, which instruments the candidate methods for offloading. Whenever such methods are called, it intercepts the call and makes execution time and energy consumption estimations for both local and remote method execution cases. The decision engine then
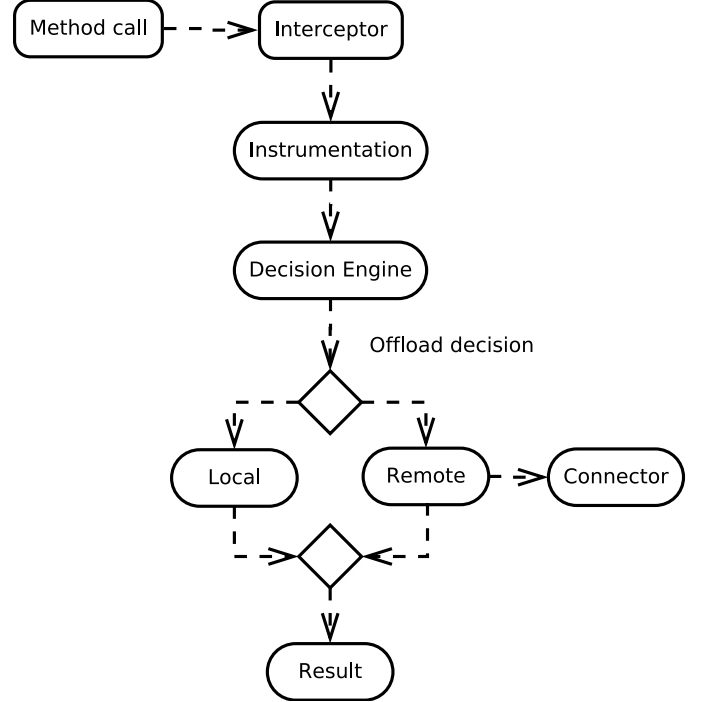

Fig. 2. Diagram of the ULOOF offloading framework modules

chooses whether to execute the method locally or remotely based on the estimation.
- The *connector module* takes care of the connectivity between the mobile device and the offloading VMs. It serializes the offloaded request/response object(s) when the method is to be run remotely.

To enable this instrumentation, an offloading-enabled Android application (or APK) needs to be prepared. Figure 3 shows the APK preparation process. First, offloadable methods are marked with an explicit annotation ("@OffloadCandidate"). Then the application is compiled with annotations, and a post-compiler creates a modified APK integrating the offloading logic in the application. The ULOOF post-compiler uses the Soot framework [20].

We also developed a remote execution environment using an Android VM to create an execution environment similar to the mobile device. The remote execution environment receives requests from an offloading-enabled application, executes the requested method and returns the result to the mobile device. To establish a communication between the offloading-enabled application and the remote code execution environment, we used HTTP to transport serialized objects in the request/response body.
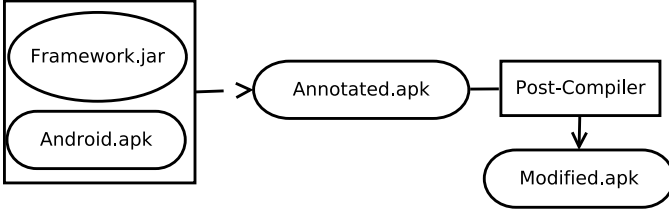
Fig. 3. Preparation of an offloading-enabled application

Because HTTP is a stateless protocol, it is suitable for a remote execution application.

3.1 On the Offloading Granularity

An offloading decision must be applied at a specific computing granularity. For example, AIOLOS [12] partitions the application on a per-class basis. Others such as CloneCloud and Cuckoo [8], [4], [5] apply the offloading decision at the method level. We choose to run our framework at the method level because class instances may contain both offloadable and non-offloadable methods (for example methods that read/write on the mobile phone's storage).

We allow application developers to explicitly mark methods as offloading candidates. This is because the developers have knowledge on which methods are the most suitable when they are offloaded and others that should not be offloaded. Developers can annotate the suitable methods with the annotation label "@OffloadCandidate", for instance, to enable offloading for the methods. As future work we plan to develop profiling tools to automatically identify the methods that are more suited to offloading.

4 ULOOF DECISION ENGINE

This section details the design of the ULOOF decision engine. We focus on the design rationale of the framework, execution time and energy consumption prediction algorithms and limitations of the framework.

4.1 Offloading Cost Functions

The ULOOF decision engine uses cost functions that estimate the energy and time required to run the methods. These cost functions make use of empirical traces to express the cost of running a method in the remote execution application.

Since the offloading problem is a multi-criteria decision problem, we employ a scaling factor, α, to prioritize either time or energy in the decision engine. In that way, α can be used to either prioritize saving execution time or battery charge. The equations for the offloading binary decision are:

$$L(m) = \alpha \cdot t_l(m) + (1 - \alpha) \cdot e_l(m) \qquad (1)$$

$$R(m) = \alpha \cdot t_r(m) + (1 - \alpha) \cdot e_r(m) \qquad (2)$$

where M is the set of methods of the mobile application, $m \in M$ is the offloading candidate method, $L : M \to \mathbb{R}$ and $R : M \to \mathbb{R}$ are the estimated cost functions in case of local and remote execution, respectively. The two components of each cost function are the execution time (t) and consumed energy (e) related to the local or remote execution of method m. Lower α

values make the decision tend towards saving energy, while higher values will improve computing responsiveness. One may develop an algorithm to adjust α based on the variable network latency, however this would have a certain impact on the computing overhead and falls outside of the scope of this work. ULOOF triggers method offloading if the following condition holds:

$$R(m) < L(m) \qquad (3)$$

That is, if the remote execution cost is lower than the local execution cost. We detail how time and energy functions are computed in the following subsections.

4.2 Execution time prediction

In ULOOF, annotated methods need to be profiled for predicting execution time when each method is called. There are two possible methodologies to profile the method execution time for computation offloading; either analyzing the target method structure to model the execution time, or predicting the execution time based on the previous execution results and dynamically adjust the prediction at each call.

In ULOOF, we follow the latter approach because it is difficult to analyze every possible method structure in various applications and compute the actual execution time based on the complexity. It is possible to compute Cyclomatic Complexity [29] of a method as a measure of its structural complexity, however this comes at a non negligible risk of big gap with the actual execution time of the method [30].

We introduce the concept of 'input assessment' to model the execution time as a function of the method arguments. It converts a list of arguments into a numerical value using an $assess(args)$ function. This function is monotonically increasing with the method time complexity; a higher value of $assess$ indicates that the method should take longer to finish. ULOOF is able to provide input assessment of primitive type arguments, however the application developer may provide its own assessment profiles for their own data structure as a $AssessmentConverter$ interface. The prediction of the execution time uses an interpolation approach. We generate an initial curve of the execution time after a few executions (five in our prototype), and each subsequent execution updates the model to improve the prediction accuracy. In order to bootstrap the decision engine, it randomly chooses between local and remote execution until it gathers five executions for each case. The decision also depends on the network state: obviously, if there is no Internet connectivity the decision is not to offload.

To avoid recalculating the curves at each new execution, we use a low-complexity 'lazy' update in our interpolation [21]: a data point from a recent execution does not trigger an update of the entire curve. Instead, only the region of the curve next to that point is updated. We chose the Akima Spline function as interpolator [22] rather than a cubic spline because we found frequent fluctuations through our initial experiment. The Akima spline can maintain value locality and avoid interference of nearby points [23].

4.2.1 Local execution

The local execution time $t_l(m)$ is defined as follows:

$$t_l(m) = \phi(assess(args_m)) \qquad (4)$$

Where ϕ is the Akima interpolation function, and $args_m$ is the set of arguments from offloading candidate method m. We

use here the already presented $assess$ function that maps method arguments to a numerical value.

4.2.2 Remote execution

Since ULOOF aims at maximizing the user experience, the remote execution time must account for the network latency due to the uploading of the method arguments and downloading results as well as the running time of the method on the remote platform.

In order to track the remote execution time, the framework gathers the execution time from the server and interpolates the execution time. The network latency depends on the network being used at that time, so latency measurements are stored on a per-network basis. When the mobile device is on Wi-Fi, measurements are bound with the access point *MAC* address. For cellular networks, the tower unique cell identifier (LAC/CID) is used as the unique identifier. ULOOF also estimates the bandwidth b of each network (as detailed in Section 4.4).

Thus, the remote execution time prediction follows equation (5).

$$t_r(m) = \phi(assess(args_m)) + d_r(m) \qquad (5)$$

$$d_r(m) = size(m, r)/b \qquad (6)$$

Where the function $size(m, r)$ returns the amount of data required to send the arguments for remote execution and to retrieve the results. Therefore, $d_r : M \rightarrow \mathbb{R}$ gives the estimated transfer delay.

4.3 Energy consumption prediction

The aim of the ULOOF energy model is to build an energy consumption profile based on the empirical data measured in the device; such a profile could be based on public device-specific records or locally generated measures.

The proposed model does not consider energy consumption from components such as the screen, GPS and file I/O nor offload methods which access these components. This is because offloading methods using these components provide a small benefit [32].

We define the energy consumption for local and remote execution as $e_l(m)$ and $e_r(m)$, respectively, as follows:

$$e_l(m) = c_{cpu}(k_m, m) + c_{radio,l}(m) \qquad (7)$$

$$e_r(m) = c_{radio,r}(m) \qquad (8)$$

$c_{cpu} : \mathbb{R} \times M \rightarrow \mathbb{R}$ is the execution time component, a function of the method m and its CPU usage k_m. The CPU usage is expressed in CPU ticks. More precisely, the estimated execution time is computed as follows:

$$c_{cpu}(m) = l_{cpu}(k_m/runtime_m) \cdot runtime_m \qquad (9)$$

where $runtime_m$ is the local execution time for method m. l_{cpu} is an estimated energy consumption function based on the tick frequency (number of ticks divided by the known execution time); it is expressed in watt per second. This quantity is then multiplied by the execution time to estimate the consumption for the whole method. l_{cpu} is independent from different methods and is device-specific.

$c_{radio,l} : \mathbb{R} \times M \rightarrow \mathbb{R}$ and $c_{radio,r} : \mathbb{R} \times M \rightarrow \mathbb{R}$ give the network latency component for a given method m, when executed locally and remotely, respectively. Radio energy consumption is also considered in local execution to account for candidate methods using the network regardless of the offloading procedure. They are computed as:

$$c_{radio,l}(m) = l_{radio}(b) \cdot d_l(m) \qquad (10)$$

$$c_{radio,r}(m) = l_{radio}(b) \cdot d_r(m) \qquad (11)$$

where l_{radio} is the energy consumption in watt per second, which is a function of the radio interface throughput. To estimate the overall consumption, l_{radio} is multiplied by the estimated time spent for transferring data. $d_l(m)$ and $d_r(m)$ are the time spent for transferring data in a local and remote execution, as explained in the next section. When the method is executed locally, it is:

$$d_l(m) = size(m, l)/b \qquad (12)$$

where $size(m, l)$ is the amount of network traffic when the method m is executed locally.

l_{cpu} and l_{radio} are therefore device-specific and method-independent functions. They characterize the device energy consumption profile. If different radios are used (e.g., Wi-Fi and 4G), different l_{radio} functions are needed. We detail in Appendix A a methodology to create such device-specific energy consumption profiles.

4.4 Network bandwidth estimation

ULOOF periodically measures available network bandwidth to improve execution time and energy consumption predictions. The transmission delay depends on the network bandwidth available to the mobile device when an offloadable method is called, therefore measuring accurate network bandwidth is an important part of the decision engine.

The bandwidth estimation algorithm performs periodic measurements. In order to account for variations in bandwidth over time, we apply an Exponentially Weighted Moving Average (EWMA) function with the previous bandwidth measured to smooth the variation using Equation 13; such a function is used because it was successfully employed to smooth noisy RTT values in TCP protocol [31].

$$b_{t+1} = b_t \times (1 - \beta) + b \times (\beta), \ 0 \leq \beta \leq 1 \qquad (13)$$

The estimation of the available bandwidth (b in the previous equation) varies from Wi-Fi and cellular. On a Wi-Fi network, the bandwidth is computed as:

$$b = \frac{d}{\triangle t} \qquad (14)$$

where d is the size of data transferred over the network and $\triangle t$ is the time to send/receive the data.

Because cellular towers serve a much larger area than Wi-Fi access points, the bandwidth in a cellular network may vary significantly based on the location. We use the reversed Shannon-Hartley's theorem to estimate the user's bandwidth. First, ULOOF derives S as follows:

$$S = \frac{b}{log_2(1 + SNR)} \qquad (15)$$

Where b is the perceived bandwidth using equation 14 and SNR is the signal-to-noise ratio. As the user moves to a different position within the coverage of the same tower, we compute the new bandwidth c' using S stored previously and SNR' at the

current position. We use the Shannon-Hartley theorem to give an estimate of the bandwidth in a new location:

$$b' = S * log_2(1 + SNR') \qquad (16)$$

Finally, ULOOF maintains historic bandwidth data for every network that it encounters. When the device is connected on a Wi-Fi network, we associate the SSID of the Wi-Fi network with the bandwidth. For cellular networks, S is stored in the location aware database along with LAC/CID.

4.5 Limitations of ULOOF predictions

Before reporting experimental and simulation results, let us point out the limitations of the proposed prediction logic.

First, the device-specific energy consumption is chipset dependent, and should be generated for the desired mobile devices (our approach is described in Appendix A). Equation 21 assumes no changing in the type of connectivity between Wi-Fi and Cellular (e.g., due to user mobility, interference phenomena, etc) within a single method execution. If interface changes occur (i.e., vertical handover), the real bandwidth and energy consumption may be different from the predicted value.

Second, the $assess$ function should provide a good estimation on the complexity of the candidate offload method. Developers should be aware of the algorithms employed and how their inputs influence the execution time. However, this may not always be an easy task. For example, most algorithms have an execution time bounded to the argument size (e.g., larger integer numbers or larger vectors). Nonetheless, many algorithms do not follow such assumption. For instance, a method that checks integers for primality: Mersenne numbers (numbers in the form of $2^n - 1$) are much easier to check than ordinary numbers. We could not find in the state of the art any method complexity measure based on the data structures or arguments. We give a detailed assessment of execution time prediction in section 5.2.

Third, an ULOOFed application is expected to work better over time, since more empirical data points generates more precise estimations. Conversely, calculations that are run occasionally may suffer from worse predictions. One way to refine the decision is to use crowd sourcing. A server would aggregate the data points for methods in an application from multiple users. This server would periodically upload to the mobile devices the most recent CPU execution curves and possibly the cellular speed estimates.

5 EXPERIMENTAL RESULTS

We developed a proof-of-concept navigation application we refer to as 'CityRoute': it finds the shortest route between two points in a graph using breath-first search. Its task complexity is $O(V + E)$, where V is the number of vertices and E is the number of edges. A city is represented by an unweighted graph, where vertices are crossroads and edges are streets. For the tests, we generated a graph with 120000 edges using the 'SNAP' dataset [25]. Each batch is composed of 36 route computations. In each computation, a destination is chosen so that the distance from the source randomly ranges from 19 to 140 edges away.

We consider three different usages of the CityRoute application: an unmodified application, an ULOOFed application with α set to 0 (i.e., energy driven), and another case with α set to 1 (i.e., time driven) to compare the execution time and energy consumption of each case. A desired α value can be computed as mentioned in Appendix B.

We used a Samsung Galaxy S5 with a Snapdragon 801 processor, a 2.5 GHz quad-core CPU and 2GB RAM as a mobile device in our experiment. We deployed a VM with a 64-bit GNU/Linux 4.4, with an Intel i7-4500U processor with four cores at 1.80GHz with 8GB of memory as remote server. An HTTP server runs on Android-x86 [24] to serve remote execution requests from the mobile device.

We set up two experiment environments to inspect the effect of different latencies on the ULOOF performance.

- Wi-Fi scenario: it is a semi-controlled environment, where the mobile device uses a Wi-Fi network to reach a server located within the same local area network. This is the case for cloudlet/MEC environments envisioned for access networks, hence for simplicity we refer to it as cloudlet use-case. There is no additional latency injected in the Wi-Fi network and it shows less than 1 ms RTT between the mobile device and the remote server.
- Cellular scenario: it is a mobile environment, where the latency with the remote server is higher than in the Wi-Fi case and can vary due to mobility.

Figure 4 reports the experienced bandwidth for the experiments of the two scenarii.

5.1 Execution time and energy consumption

We measured the execution time and the energy consumption of the mobile device while the application is run. We measured the start time and end time of each test, and the battery level during each test (using the Android BatteryManager API).

5.1.1 Wi-Fi scenario

Figure 5 reports the execution time of 20 contiguous execution batches of the CityRoute application in the horizontal axis, i.e., the last point of each line is the global execution time. During such executions, we noted the instant when the battery drained by 1% (it is the minimum measurement step available with user-space Android primitives) and report it accordingly as a vertical step in the figure. Execution time for both ULOOF versions are significantly reduced compared to the unmodified application as noticed from the shorter horizontal length of the plot. Battery usage from both versions is also reduced as shown in the shorter vertical height of plots.

The results show that ULOOF reduced the execution time as well as energy consumption, and that for both time and energy driven variants. More precisely, methods were offloaded 72.95% and 62.41% of the times for the time-driven and energy-driven modes, respectively. The execution time was reduced by about 50%. The battery gain ranges from 5 to 6% in terms of absolute battery consumption, which roughly corresponds to 56 to 224 mAh for the given phone.

The differences in time-driven and energy-driven modes are seen as a longer time to drain the battery: drops in energy occur later for the energy-driven operation. Further, the time-driven version finishes the execution a few seconds before the energy-driven version.

5.1.2 Cellular scenario

For the cellular scenario, we experimented on a moving vehicle around the city of Belo Horizonte, Brazil, along a predefined route. We first measured the performance of cell towers in the city and then used the data gathered from the measurement.
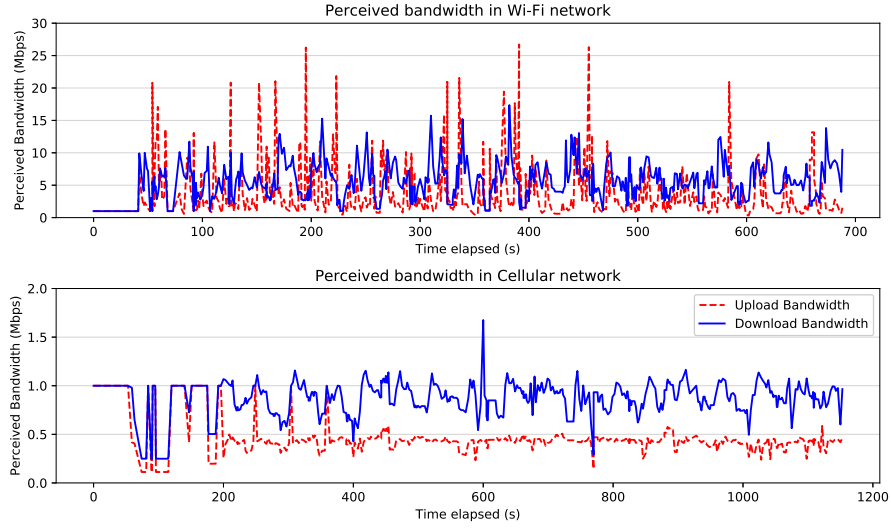
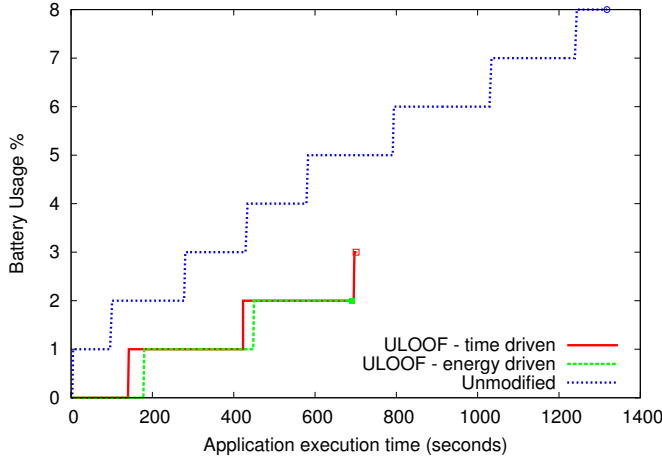Fig. 4. Change in perceived bandwidth over elapsed time.

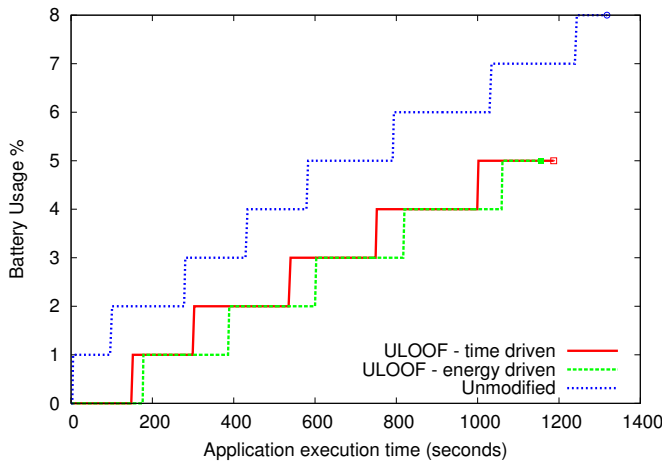Fig. 5. Battery usage and execution time in the Wi-Fi scenario.

Fig. 6. Battery usage and execution time in the cellular scenario.

Figure 6 shows the accumulated execution times and battery consumed by the CityRoute application executed 20 times in a row. Both time-driven and energy-driven ULOOF improved the execution time and energy consumption compared to the unmodified application. However the absolute gain in execution time and energy consumption has reduced compared to the Wi-Fi/lower-latency case.

More precisely, the decision engine offloaded 27.6% of the executions with energy-driven ULOOF and 29.13% with time-driven ULOOF, with only a small difference between the two modes. The average execution time of the offloaded execution was around 2.5 seconds, above the global average of 1.5 seconds.

Compared to the Wi-Fi scenario, there was 47.64% longer execution time and 80% higher energy consumption; this is due to the smaller number of offloads in the higher-latency network. Because of the longer latency, the remote execution time estimate increases and therefore the decision engine decides to offload less often. It causes mobile devices to execute more often locally and consumes more energy.

Moreover, the time-driven ULOOF in the higher-latency network had an execution time slightly longer (globally 30 seconds longer, i.e. 2.6%) than the energy-driven ULOOF: this is also due to the uncontrolled environment with network latency variations.

5.2 Prediction accuracy

In order to assess the accuracy of the prediction in ULOOF, we post-processed the execution time and energy consumption for both scenarios.

Figure 7 reports the prediction error ratio on the left vertical axis using boxplots (minimum, quantiles, maximum). The prediction error ratio is computed as the ratio between the predicted value and the actual value. The right vertical axis reports the mean value of the execution time (the red line plotted in the figure) with 95% confidence interval. The X axis indicates the distance to destination in number of edges in the graph, which roughly approximates how much computation was required to calculate the shortest path for each destination.

Each figure block shows results for both local executions (top) and remote execution (bottom). It is worth noting that we had to
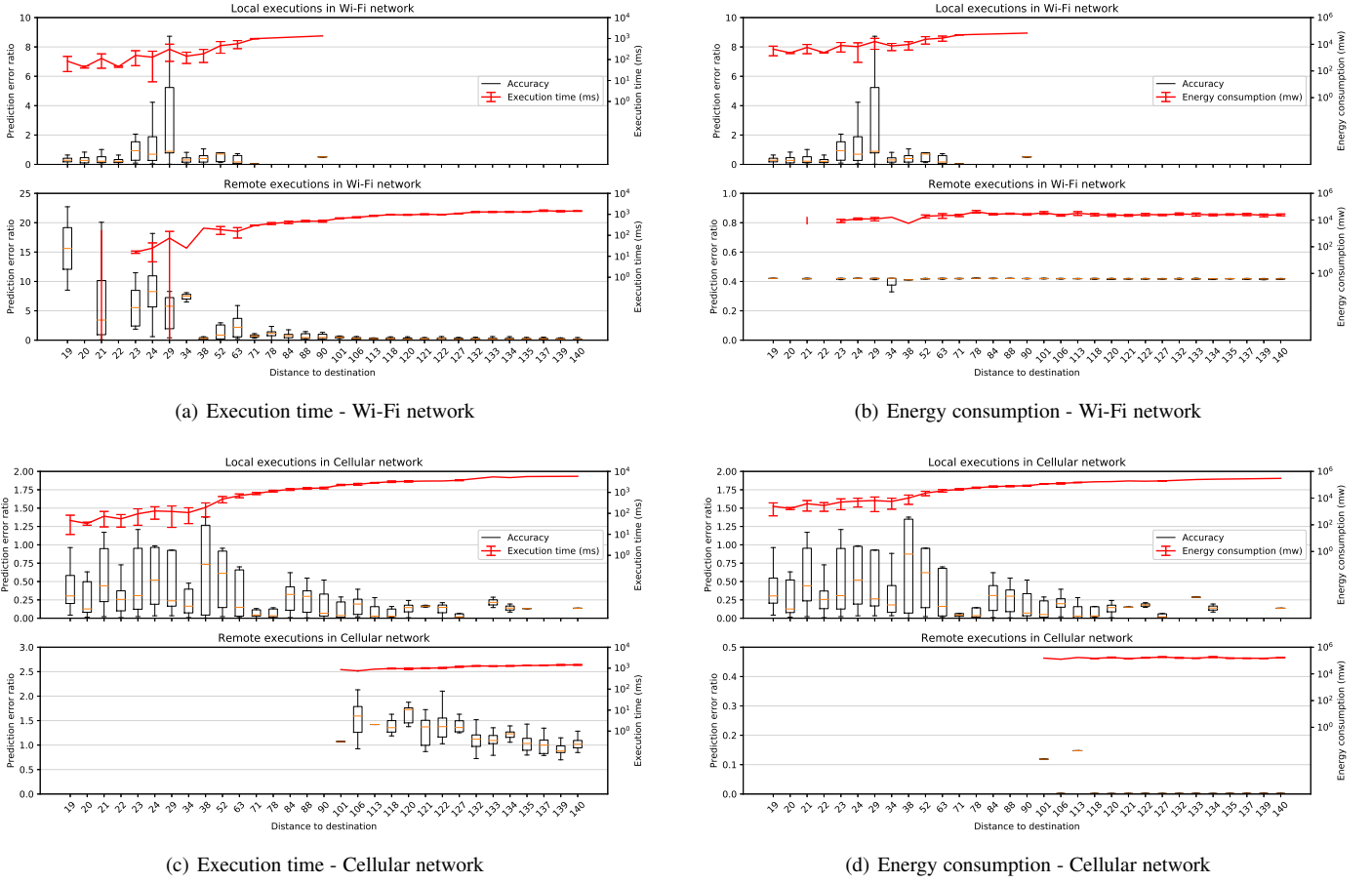
(a) Execution time - Wi-Fi network

(b) Energy consumption - Wi-Fi network

(c) Execution time - Cellular network

(d) Energy consumption - Cellular network

Fig. 7. Prediction error as a function of method complexity (expressed in ho distance from source to destination for navigation map app).

rely on our energy consumption fitting model for both prediction and the actual consumption. This is because it is not possible to record the energy consumption of a method-level granularity from the device.

We discuss the results for the lower and higher network latency cases in the following sections. It is worth stressing that the samples for the remote execution are concentrated at longer distances because offloading happens less often in executions that take less time. Conversely, the number of samples is lower for local executions at high distances.

5.2.1 Wi-Fi network

For the Wi-Fi network scenario, method calls with lower computation execution time suffer from high prediction errors, especially for remote executions. As the execution time increases, the prediction error decreases, with a median always below 50% starting by 63 hops for local executions. As the processor in the mobile device is shared among processes, the error margin is higher for methods with short execution time.

We can further notice that for the local executions, there is an increasing trend in terms of accuracy as the execution time increases. We found there is an average margin of prediction error of 93.54% when the computation takes 156 ms to complete, which decreases to 3.16% as the execution time increases to 985 ms. For the remote executions, we have a similar trend, with an average margin of prediction error of 555% when the execution

time is 15.79 ms, which decreases to 14.08% as the execution time increases to 1435 ms.

The effect of large error margins however does not impact the overall performance of ULOOF. This happens because the offloading will occur only for larger instances of the problem, in which the execution time is much longer. For shorter execution times, most executions happen locally due to the delay required to send data to the remote environment.

The prediction error ratios in energy consumption show that our framework is more accurate when predicting the energy required for remote execution. This happens because the remote execution energy consumption relies heavily on the number of bytes transferred across the execution (i.e. size of argument and result transferred), and the amount of bytes transferred do not differ much for each method call. In contrast, local executions suffer from high prediction errors when the complexity is low, because of the noise related to background computations.

5.2.2 Cellular network

Figure 7 (c) and 7 (d) show the prediction error in the cellular/higher-latency network experiments.

The prediction error ratio in execution time decreases with higher computation complexity for local executions. For remote executions, instead, the error ratio shows a median between 100% and 150%, which is likely due to bandwidth variations in the cellular network. Compared to the Wi-Fi scenario, the error ratio is smaller in Wi-Fi because there is less variation in bandwidth.

The energy consumption prediction is also more accurate for the higher-latency case. As the complexity increases in local executions, the prediction accuracy improves. Because the local execution time and the local energy consumption are closely related (i.e. longer execution consumes more energy) and they both use the Akima interpolation [22], their accuracy is similar. For the remote execution, however, the prediction of the energy consumption improves significantly.

5.3 System overhead

In terms of system performance, it is important to qualify the overhead caused by the ULOOFed applications. Figure 8 shows the time taken for making offloading decision relative to the actual method execution using CityRoute in the cellular network. We measured the overhead of our framework by measuring the time taken to predict execution time and energy consumption against the total method execution time.

The overhead incurred from making offloading decision was less than 40 ms at all times. For short execution times, the overhead tops at 32%, which is 33 ms of overhead. However, for longer execution times this overhead is lower than 10%. Although this overhead may be significant for methods that run the least, it is worth noticing that the average execution time of the methods is 513.47 ms.
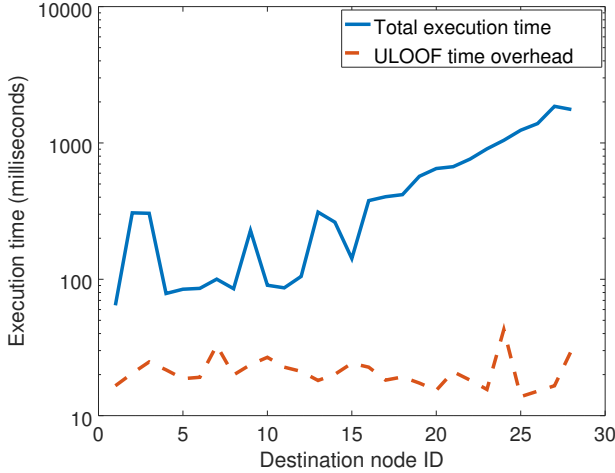

Fig. 8. Overhead caused by the offloading framework.

6 SIMULATION USING REAL MOBILE TRACES

In an attempt to have an assessment less specific to a given testbed or a given trace, we have used a cellular mobility data-set obtained in the frame of a collaboration with a major French mobile network operator.

6.1 Simulation environment

In the provided data-set, each device is located at the LAC (Local Area Code) level whose coverage ranges from few kilometers to dozens of kilometers of radius depending on the population density. We used France-wide mobile user trajectories with a time-stamped list of traversed LACs in a given day. We discarded trajectories that were covered by only one user to ensure 2-anonymity aggregation. We kept trajectories crossing more than
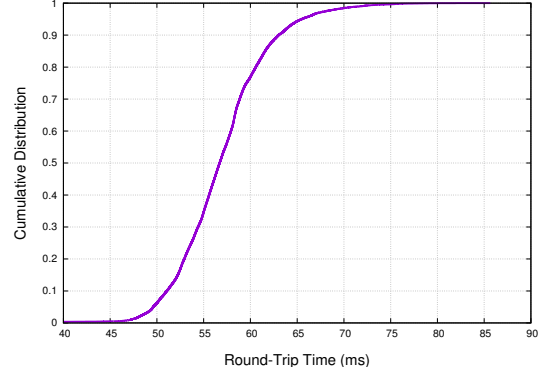

Fig. 9. Round-trip-time distribution from the simulation data-set.

3 LACs for a more realistic use-case and finally we obtained the 36,450 trajectories for our simulations.

Both a remote cloud facility and a nearby facility configurations are simulated, with the remote cloud facility fixed in Paris, France. The MEC facility is fixed in the nearest LAC (hence supposing adaptive virtual resource mobility as envisioned in MEC specifications and investigated in [28]). The location of a user changes in time with a variable LAC sojourn time. The CityRoute repeatedly recalculates the distance of the predefined 36 destinations as the user moves to another LAC. The connectivity between the mobile users and the closest VM is supposed to be automatically handled by standard cellular mobility management protocols. For each user, we computed RTTs between the remote LAC or nearest LAC and the user position to compute propagation delays in wired networks; three times the distance between each point, then adding 40 ms to each RTT to reproduce bufferbloat phenomena typical of cellular networks, following the methdology in [26]:

$$RTT = 3 \times d(LAC_u, LAC_s) + 40ms \qquad (17)$$

where $d(x,y)$ is a function for calculating euclidean distance between x and y. The resulting RTT distribution is shown in Figure 9. Based on the RTT distribution, we computed the latency of each offloading request and response. The latency is computed as the following equation 18

$$Latency = RTT + D_t + D_s \qquad (18)$$

Where D_t is the transmission delay and D_s is serialization/deserialization delay. D_t and D_s are modeled from empirical results obtained in the previous evaluation scenario. The execution time and energy consumption profiles are also derived from the experiments in the previous section. For the predictions, we used an α parameter set to 0.993, which is a trade-off found in an empirical analysis described in Appendix B.

6.2 Simulation results

We show simulation results when varying the RTT and based on the trajectory of the user. The RTTs values in the X axis of the figure 10 are limited to 85ms because of the characteristics of the dataset, which does not provide trajectories with larger RTTs. First, we use the samples of every possible combination of user LAC-server LAC RTTs. This configuration is meant to give a rough estimate of the gain one could have with the given geographical scope, independently of the cloud location policy.

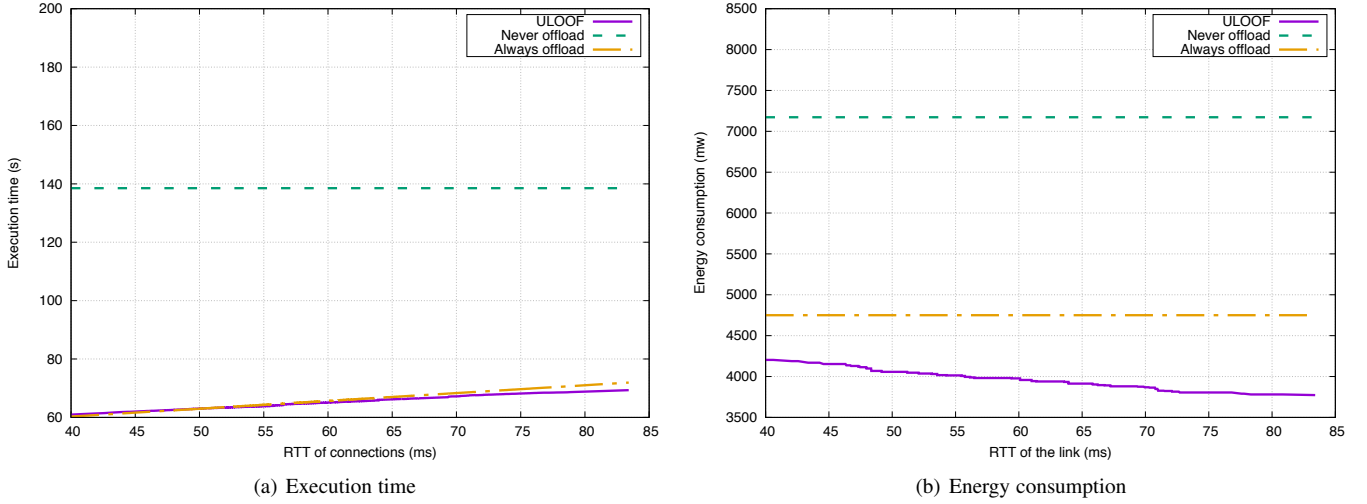(a) Execution time

(b) Energy consumption

Fig. 10. Performance as a function of the RTT (simulation).

Figure 10 shows the execution time with ULOOF comparing to the case where offloading is not executed ('never offload') and the case where offloading always happens ('always offload'). We can see that with ULOOF, the total execution time can be reduced by a factor between 49% and 55% and that, as the RTT increases, this gain decreases essentially due to the stronger impact of the RTT in the global execution time. Figure 10b shows the energy consumption results. The energy gains show a lower range, from 40% to 47%.

Figures 11 and 12 show the results obtained from a user trajectory perspective instead. All the 36,450 trajectories were used here to extract an individual offloading experience assessment. In this analysis, we assumed that when a user changes its LAC, the CityRoute application is executed and the energy and time performance is stored. The weighted average of each performance result (time or energy gain) is then computed, weighting each offloading result with the sojourn time of the user in the given LAC.

We calculate the weighted average W_a for each user using the following equation:

$$W_a = \frac{\sum_i^k g_i t_i}{\sum_i^k t_i} \tag{19}$$

where i is a trajectory of a user with the maximum k trajectories, g_i is the resource gain of a user in the trajectory i and t_i is a sojourn time of a user at the trajectory.

The results show that the median of the execution time gain is 55% for the remote cloud case (fixed server in Paris) and 58% for the nearby cloud case (server in the closer LAC). In terms of energy gain, the median gain ranges from 49.3% to 49.7%, roughly, respectively. The remote execution consumes less energy because the offloading decision in the simulation is skewed towards shortening the running time. A small percentage of the methods that are run in the local cloud actually consume more energy than in the mobile device, however they are run in the cloud in order to reduce the response time. Those methods are not run in the remote cloud because the RTT is longer, and hence there is no benefit, either in energy or in runtime, of a remote execution.

As remote cloud have longer RTT compared to the nearby cloud, they consumed less energy as shown in Fig 10 (b). They did
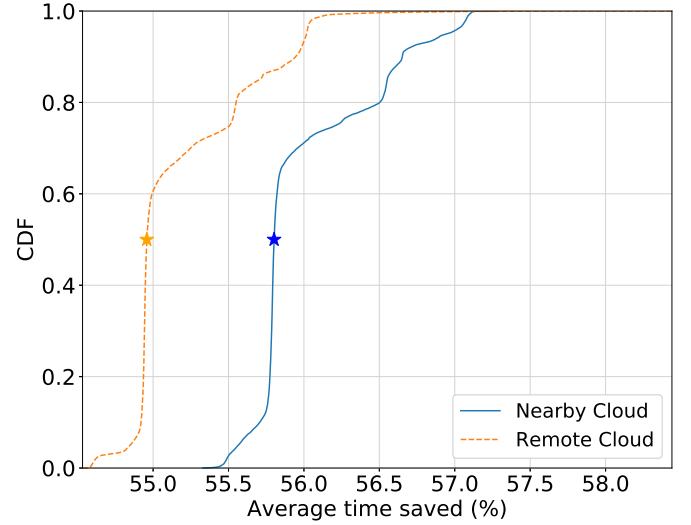

Fig. 11. Execution time gain cumulative distribution (simulation).

not offload a small percentage of the methods that consume more energy when offloaded because the RTT is longer, and hence there is no benefit, either in energy or in runtime, of a remote execution.

With respect to the trajectory-agnostic results in Figures 10, we can notice that using a nearest LAC policy for server placement does not show different gains, which differ only of a few percentage points for both execution time and energy consumption. This seems to suggest that running the service in a fixed VM in a central location of the access network does not lead to significantly lower performance than a case where the VM is moved closer to the user in a nation-wide network.

7 CONCLUSIONS

This article presented the ULOOF mobile computation framework, a user-level online computation offloading framework including an innovative decision engine to decrease energy consumption of mobile devices and the execution time of mobile applications. The ULOOF decision engine exploits empirical profiles to predict the energy consumption and execution time of Android
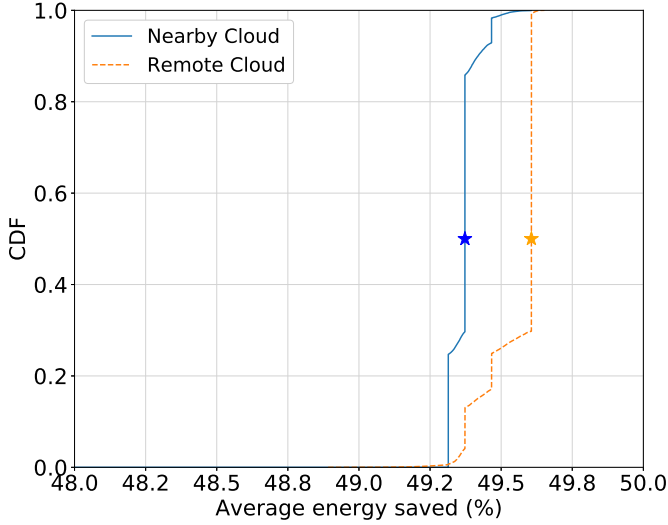
Fig. 12. Energy gain cumulative distribution (simulation).

application methods, using an assessment of the inputs, and by taking location awareness into account. It uses a low overhead energy consumption model to aid in the mobile offloading decision process. ULOOF does not require any special configuration nor modifications to the runtime of both the device and the edge computing platform, being easily plugged into any framework and application without the need to root or modify the device operating system. An example of ULOOFed application is made available in [27].

The framework was evaluated by testbed experiments and large-scale simulations using real data from a major cellular access provider. The results show that both execution time and energy usage can be significantly improved by offloading methods to an external server. We considered both a nearby server (local cloud) scenario, like in MEC environments, and a remote cloud scenario with a longer network latency. The effectiveness of the modelling was evaluated, measuring the accuracy of the interpolations as well as whether the bandwidth actually changes among cell towers. The results indicate that ULOOF can reduce the energy consumption on the mobile device of roughly 50% for Wi-Fi scenarios with low cloud access latency, and lower yet positive gains also for situations with high latency.

Further work is needed to conceive supervised learning approaches for prediction the mobility behavior of the user and hence improve the ULOOF prediction accuracy.

ACKNOWLEDGEMENTS

APPENDIX A
PROFILING THE DEVICE ENERGY CONSUMPTION

The energy consumption of the CPU and the radio interfaces must be derived empirically for each device. For the CPU, we must derive a function that maps the number of ticks of a method

into the energy consumed running that method. Similarly, for the wireless interfaces we must derive a function that maps the number of bytes transmitted into the energy consumption of that data transmission on a certain interface.

Preliminary tests using Android OS primitives showed that the accuracy of the energy estimations on the OS are very low. Hence, we performed hardware-based profiling using an off-the-shelf equipment (KCX-017 adapter) that emulates a charger while measuring the power drained by the device.

To obtain the l_{cpu} and l_{radio} empirical distributions, we created two distinct Android applications to measure separately the energy consumption of the CPU and the consumption of the network interfaces. All the tests were performed in a Samsung S5 with only the profiling application running. For the CPU test the network interfaces were disabled, and for the tests of the network interfaces only one interface is active at a time.

For the energy consumption of the CPU, we generated a constant CPU load by running an application which keeps multiplying random prime numbers using multiple threads. To generate partial loads on all the cores, a short sleep period is set for each thread, calibrated according to each mobile phone. With the given phone, a sleep period of 100 ms every $5^{(load \mod 25)/5}$ iterations was found to work well, where $0 \leq load \leq 100$. We chose the following set of loads for our tests in terms of number of executions: $L = [10, 25, 35, 50, 60, 75, 87, 100]$. Each load L_i was kept for two minutes in order to obtain sufficient amounts of data.

For radio, an application and a web server were developed to exchange traffic. Preliminary tests were first performed by varying the total transfer time, keeping the number of bytes fixed. This showed to be unfruitful and the current barely modified along the experiment, as verified in [19]. This behaviour is expected, since the radio can keep a low power state during small network loads.

A second batch of tests were performed, in which the server creates a delay of 1 millisecond after a certain number of transmitted bytes. This mechanism controls the throughput of the server, and effectively generated a variable energy consumption. A delay of 1 millisecond every $[0, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 76800, 153600]$ bytes was used. The duration of the experiment was set to 2 minutes.

Using the data gathered from the above experiments, we performed a polynomial curve fitting to find the energy profile of the components.

Figure 13 shows the empirical results for the energy consumption due to CPU usage (l_{cpu}), with max-min error bars. It shows an upward trend reaching a plateau at the end, as the CPU reaches the full load. The plotted fitting curve, having a coefficient of determination of 0.96605, is as follows, where t is the number of ticks in the computation:

$$
\begin{aligned}
l_{cpu}(s) = &+ 51.422 + 2.9076 \cdot t^1 + 0.019306 \cdot t^2 \\
&+ 6.7841 \cdot 10^{-5} \cdot t^3 - 8.4491 \cdot 10^{-8} \cdot t^4
\end{aligned}
\tag{20}
$$

Figure 14 shows the curve of 4G and Wi-Fi radio interface consumption, as a function of the amount of transferred bytes per second (b). The fitting curves, with a coefficient of determination of 0.99020 and 0.84887 for Wi-Fi and 4G respectively, are:
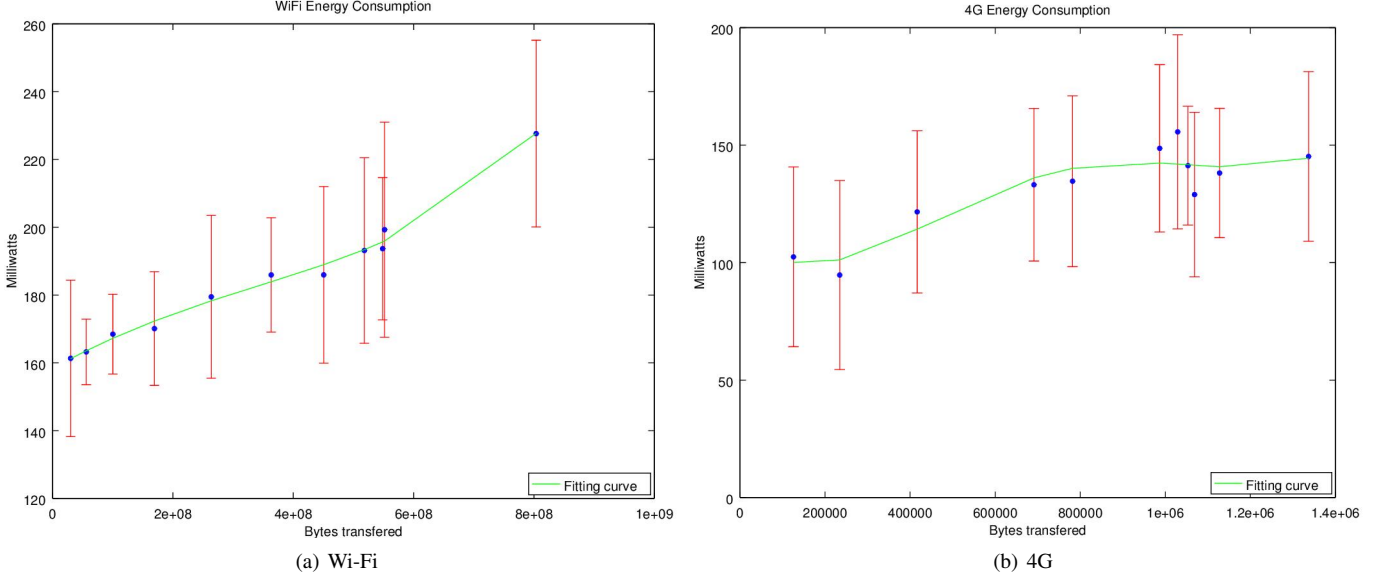
(a) Wi-Fi

(b) 4G

Fig. 14. Power consumption experimental distribution for different traffic loads and network interfaces.
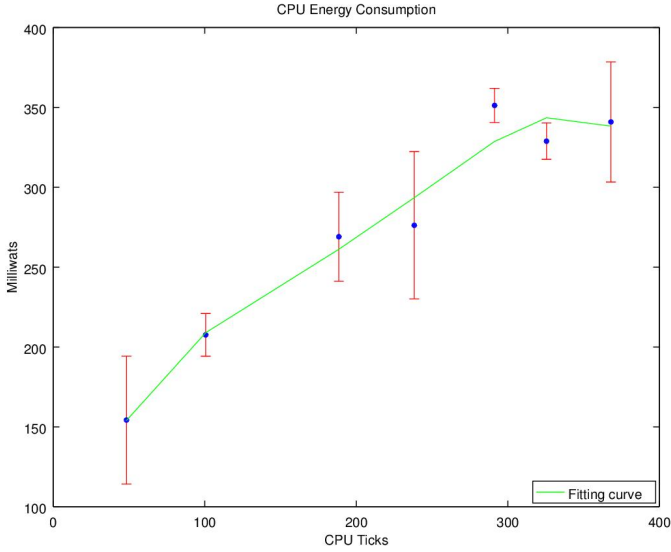

Fig. 13. Power consumption for CPU usage together with its standard deviation

$$
l_{radio}(s) = \begin{cases}
+111.24 - 7.9499 \cdot 10^{-5} \cdot b^1 \\
+1.5999 \cdot 10^{-10} \cdot b^2 - 8.3738 \cdot 10^{-17} \cdot b^3 \\
+1.3748 \cdot 10^{-23} \cdot b^4, \qquad \text{if using 4G} \\
\\
+158.37 + 1.1811 \cdot 10^{-5} \cdot b^1 \\
-1.4722 \cdot 10^{-12} \cdot b^2 + 6.1454 \cdot 10^{-20} b^3 \\
+1.8794 \cdot 10^{-26} b^4, \qquad \text{if using Wi-Fi}
\end{cases}
\tag{21}
$$

APPENDIX B
OFFLOADING DECISION TRADE-OFF EVALUATION

The α parameter defines how to prioritize between execution time and energy consumption when making an offloading decision. In this appendix, we show the impact of α on the offloading performance. These experiments were performed in a Wi-Fi network.

Besides the CityRoute application, we also use a Fibonacci application computing the Fibonacci number of a random number. In the following analysis we compare three scenarios: 'Always Offload', 'Never Offload', and 'ULOOF' with different values of α. Let us recall that $\alpha = 0$ means ULOOF considers energy saving only, and $\alpha = 1$ for saving execution time, while intermediate values give different trade-offs between these two objectives.

Figure 15 shows the results for the CityRoute application. The 'Never Offload' curve presents an application that runs all the computation locally, while the 'Always Offload' curve shows the results for the computations always being performed in the remote server. All plots have the horizontal α axis cut to the region where it changes of shape, in this case from 0.98 to 1. Before that ULOOF does not change its decision because of the unnormalised values for time and energy. In terms of execution time, as one can see ULOOF always performs better than the other two scenarios with any α value. The sweet spot is where both lines from the always offloading scenario and ULOOF cross in the energy plot: $\alpha = 0.993$.

In order to show that the ideal value of α depends on the application, the same test was performed using the Fibonacci application. This application demands a meager amount of network, however it is CPU intensive. Figure 16 shows the results. In this case ULOOF sits in between the two other cases in terms or execution time, and it is always better in terms of both energy consumption and execution time when $\alpha > 0.995$.

The last sensitivity test evaluated the effect of transferring larger amounts of data when offloading. This was performed using a modified version of the Fibonacci application, where we transfer a large argument to the offloading method. This forces the large size of argument to be transferred through the network each time the method is to be offloaded. The computation ignores this argument as it is only to increase the transfer size. Figures 16 and 17 reflect how this changes the performance of the ULOOF framework. The overhead of transferring the large argument impacts greatly the execution time plot, performing
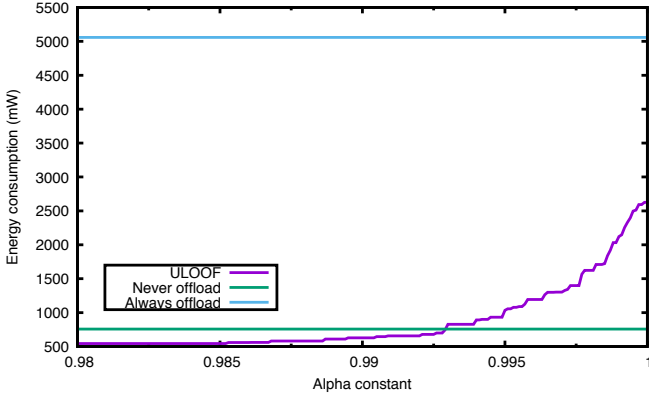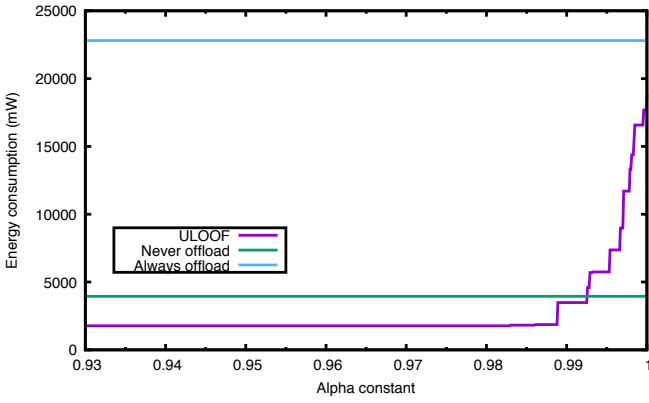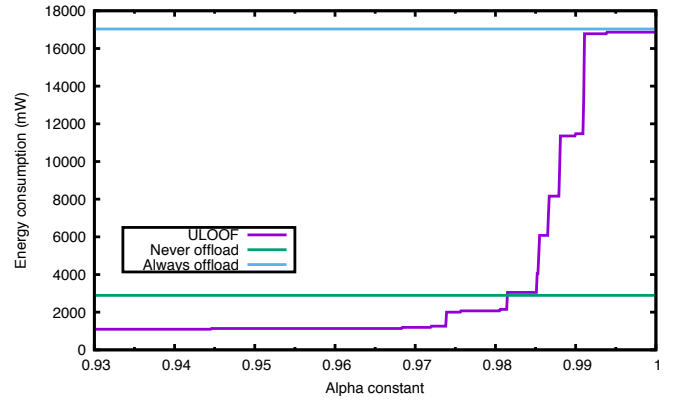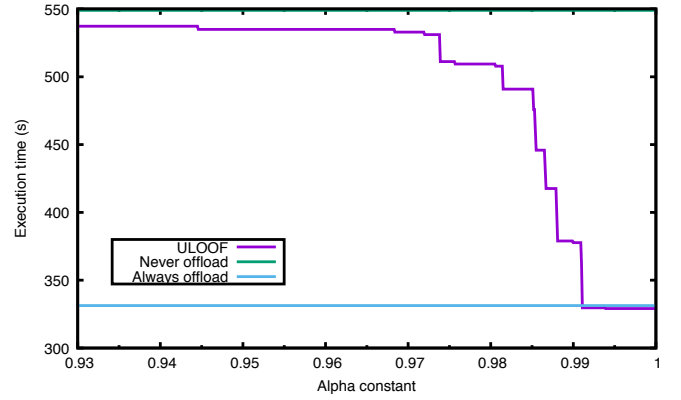
(a) Energy consumption
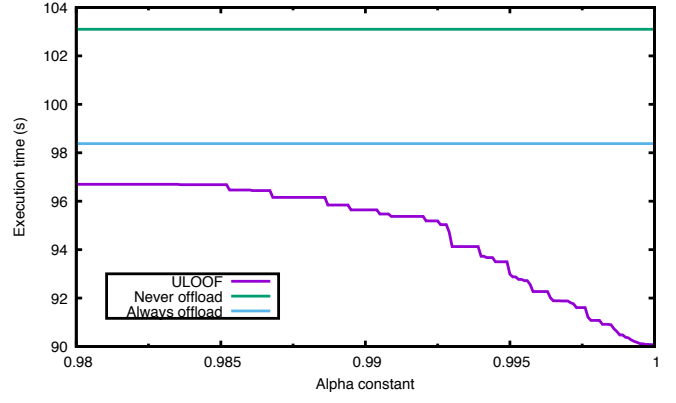
(b) Execution time

Fig. 15. ULOOF performance as a function of α (CityRoute).

(a) Energy consumption

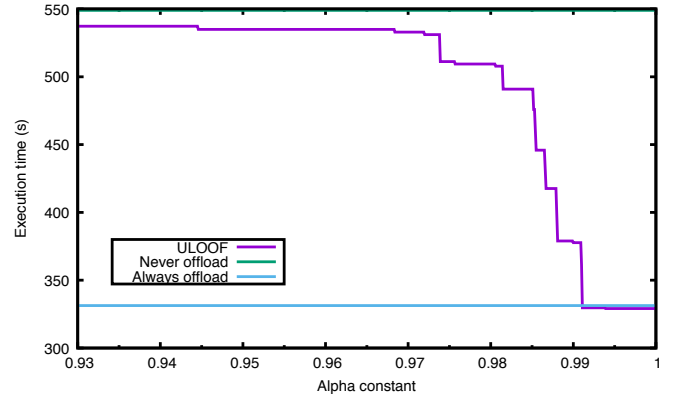(a) Energy consumption

(b) Execution time

Fig. 16. ULOOF performance as a function of α (Fibonacci).

(b) Execution time

Fig. 17. ULOOF performance as a function of α (modified Fibonacci).

time-wise worse than the never offload scenario when favouring energy but for $\alpha > 0,99$ for which it has close performance to it. This happens because of the transmission delay involved in transferring the arguments.

REFERENCES

[1] D. Chaffey, "Mobile marketing statistics 2016," Apr. 2016.

[2] X. Ma, "Characterizing the Performance and Power Consumption of 3d Mobile Games," *Computer*, vol. 46, no. 4, pp. 76–82, Apr. 2013.

[3] Y. Mao, J. Zhang, K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *arXiv preprint arXiv:1605.05488*, 2016.

[4] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: A Computation Offloading Framework for Smartphones," in *Mobile Computing, Applications, and Services*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, M. Gris and G. Yang, Eds. Springer Berltin Heidelberg, 2012, vol. 76, pp. 59–79.

[5] E. Cuervo et al., P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *Proc. of ACM MobiSys 2010*.

[6] A. Pamboris, "Mobile Code Offloading for Multiple Resources," Ph.D.

dissertation, Imperial College London, 2014.

[7] M. ETSI, "Mobile-Edge Computing," *Introductory Technical White Paper, September*, 2014.

[8] B.-G. Chun et al., "CloneCloud: Elastic Execution Between Mobile Device and Cloud," in *Proc. of ACM EuroSys 2011*.

[9] C. Shi et al., "COSMOS: Computation Offloading As a Service for Mobile Devices," in *Proc. of ACM MobiHoc 2014*.

[10] S. Kosta et al., "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. of IEEE INFOCOM 2012*.

[11] 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. of IEEE/IFIP NOMS 2016*.

[12] T. Verbelen, P. Simoens, F. D. Turck, B. Dhoedt, "AIOLOS: Middleware for improving mobile application performance through cyber foraging," *J. of Systems and Software*, vol. 85, no. 11, pp. 2629 – 2639, 2012.

[13] R. Esteves, M. McCool, C. Lemieux, "Real options for mobile communication management," in *Proc. of IEEE GLOBECOM Workshops*.

[14] M. Kristensen, "Scavenger: Transparent development of efficient cyber foraging applications," in *Proc. of IEEE PERCOM 2010*.

[15] T. L. Cignetti, K. Komarov, C. S. Ellis, "Energy Estimation Tools for the Palm," in *Proc. of ACM MSWIM 2000*.

[16] A. Pathak, Y. C. Hu, M. Zhang, "Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof," in *Proc. of ACM EuroSys 2012*.

[17] S. Hao et al., "Estimating Mobile Application Energy Consumption Using Program Analysis," in *Proc. of IEEE ICSE 2013*.

[18] L. Zhang et al., "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. of IEEE/ACM/IFIP CODES-ISSS 2010*.

[19] A. P. Miettinen, J. K. Nurminen, "Energy Efficiency of Mobile Clients in Cloud Computing," in *Proc. of USENIX HotCloud 2010*.

[20] R. Vallée-Rai et al., "Soot - a Java Bytecode Optimization Framework," in *Proc of CASCON 1999*.

[21] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, Sep. 1989.

[22] H. Akima, "A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures," *J. ACM*, vol. 17, no. 4, pp. 589–602, Oct. 1970.

[23] G. Wolberg, I. Alfy, "Monotonic cubic spline interpolation," in *Computer Graphics International, 1999. Proceedings*.

[24] "ReleaseNote 6.0-r1 - Android-x86 - Porting Android to x86."

[25] J. Leskovec, R. Sosič, "SNAP: A General-Purpose Network Analysis and Graph-Mining Library," *ACM Trans. on Intelligent Systems and Technology*, vol. 8, no. 1, p. 1, 2016.

[26] H. Jiang et al., "Understanding Bufferbloat in Cellular Networks," in *Proc. of ACM SIGCOMM 2012*, CellNet Workshop.

[27] ULOOF project website: https://uloof.lip6.fr.

[28] S. Secci, P. Raad, P. Gallard, "Linking Virtual Machine Mobility to User Mobility", *IEEE Trans. on Network and Service Management*, Vol. 13, No. 4, pp: 927-940, Dec. 2016.

[29] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[30] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30–36, March 1988.

[31] V. Paxson and M. Allman, "Computing TCP's Retransmission Timer," RFC Editor, RFC 2988, Nov. 2000, published: Internet Requests for Comments.

[32] L. Corral, A. B . Georgiev, A. Sillitti, G. Succi, "A Method for Characterizing Energy Consumption in Android Smartphones," *2nd International Workshop on Green and Sustainable Software (GREENS)*, pp. 38-45, 2013.