2017 IEEE 37th International Conference on Distributed Computing Systems

Modeling Mobile Code Acceleration in the Cloud

Huber Flores, Xiang Su,
Vassilis Kostakos, Jukka Riekki
University of Oulu
firstname.lastname@oulu.fi

Eemil Lagerspetz, Sasu Tarkoma
University of Helsinki
firstname.lastname@helsinki.fi

Pan Hui
HKUST
panhui@cse.ust.hk

Yong Li
Tsinghua University
liyong07@tsinghua.edu.cn

Jukka Manner
Aalto University
jukka.manner@aalto.fi

***Abstract*—The quality of service of a mobile application is critical to ensure user satisfaction. Techniques have been proposed to accomplish adaptation of quality of service dynamically. However, there is still a limited understanding about how to provide a utility model for code execution. One key challenge is modeling the level of quality in the code execution that can be provisioned by the cloud. Since the allocation of cloud resources has a cost, it is important to optimize cloud usage. We propose a software-defined networking approach that allows modeling and controlling code acceleration of a mobile application deployed across multiple type of devices. By segregating the computational requirements of the mobile application into groups, we were able to define the acceleration needed by each group of devices. As the computational requirements of a device can change across time, a mobile device can be re-assigned to another group based on demand. Our SDN approach implements a model that allows the system to predict workload based on acceleration groups. Evaluating our system in a real testbed showed that it is possible to predict workload and allocate optimal resources to handle that workload with 87.5% accuracy.**

***Index Terms*—Software-defined; Code Offload; Mobile Cloud**

I. Introduction

Nowadays, the deployment of large-scale pervasive and mobile applications is relatively easy by relying on cloud computing, which provides the platform and tools for centralized management. However, in these deployments, low-power devices, e.g., smartphones, wearables, etc., are subject to energy and performance issues. Thus, approaches for tuning those aspects from the cloud are critical.

Meanwhile, mobile applications are released at an exorbitant rate in the app stores. The success of a mobile application in the market depends on many factors that are perceived by the user [1], such as the energy drained, and the response time. Recent studies show that 80% of the mobile applications in the stores are installed, used for some time, and then uninstalled from the device [1, 2]. Therefore, tuning applications on the fly is critical for improving user perception and fostering better application adoption. The ultimate goal is to gain competitive advantages.

Tuning a mobile application is a complex task. In order to fix an issue, the application has to be taken back into the development stages and then re-deployed again across all the devices, e.g., via application updates. Moreover, given the variety of hardware in mobile devices, usually a mobile application needs to be troubleshot for each type of device. For instance, complex routines like decision making algorithms (e.g. minimax and nqueens) can be computed easily by last generation smartphones but can be expensive to compute on older devices and wearables. Thus, the routine needs to be optimized for execution based on device capabilities. This approach requires a considerable development effort and results in high costs. Moreover, the troubleshooting of an application can take a considerable amount of time.

Existing tools proposed for monitoring and adapting the quality of service of applications rely on improving communication protocols and optimizing data transfer for OTT (Over The Top) applications such as video delivery, voice over IP (VoIP) and real-time video calling [3]. While these techniques can be successfully applied in thin clients to ensure the quality of service across multiple devices, they cannot be applied for thick clients that provide independent functionality whose computation depends solely on the mobile resources. Hence, also techniques for dynamically allocating resources in the mobile device have been developed [4]. Unfortunately, the computational capabilities of a mobile device are limited, so these approaches can provide only a partial solution.

Similarly, approaches for outsourcing code to an external server have been widely explored in the literature, e.g., cloud and cloudlet [5]. Moreover, in practice, beta services such as *Amazon Lambda*, have the potential to tune the execution of thick applications. However, there is still a limited understanding regarding how to provide a utility model for outsourcing code to the cloud. Since a cloud can provide different quality of service based on the computational capabilities of the underlying resources, one key challenge is to model the level of quality in the code execution that can be provisioned by the cloud. This quality of execution can be expressed in terms of acceleration or from a user point of view as quality of the response time in the mobile application.

In this paper, we overcome the problem of tuning the performance of mobile applications on the fly. We use a software-defined approach [6] that provides the tools for modeling and controlling the traffic of code acceleration of mobile applications deployed across a diversity of devices in the wild. To the best of our knowledge, our work is the first to explore how to adjust the performance of applications dynamically with the cloud. We make the following contributions:

First, we develop a code offloading architecture that automatically accelerates a mobile application when the response time starts to degrade. The architecture includes a cloud-based SDN component, which dynamically routes application traffic into the right level of acceleration, and a client-side moderator component, which monitors the execution time of the code in the application, and promotes the execution of code to a higher level of acceleration when it detects that the response time of the application starts to degrade. We find that the SDN component introduces a very small overhead of ≈150 milliseconds in the total response time of a request, which is

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.242

IEEE computer society

a fair price to pay for tuning code execution on demand.

Second, we design and evaluate a model that abstracts the computational resources of the cloud into levels of code acceleration. Our model predicts the expected workload that the system needs to handle per each acceleration level. Based on this information, it optimizes the cost of allocating computational resources needed to handle that workload [7]. The model is able to predict expected workload with 87.5% accuracy.

The rest of the paper is organized as follows. Section II explains the terminology used in the work. Section III presents the literature review. Section IV explains the design of the system and defines our adaptive model. Section V describes the technical implementation of the system. Section VI presents the evaluation, conducted on a real testbed. Moreover, we collected real smartphone use traces for 3 months in order to induce realistic workload into the system. We discuss the impact of our results in Section VII, and Section VIII concludes the paper.

II. Background

A. Code Offloading

Code offloading (aka computational offloading or cyber foraging) is a technique that allows a low power device, e.g., a smartphone, to outsource the processing of a task to a server or the cloud [5]. A task is opportunistically [8] outsourced from a smartphone app when the handset can reach the server at low latency. A smartphone delegates a task to a remote server, if and only if, the computational effort required for the device to delegate the task is less than the actual effort required to process the task by itself. The ultimate goal of the technique is to reduce the overall amount of processing of the device to extend battery life [9].

Multiple flavors of code offloading can be implemented in practice. Figure 1 shows these implementation models. We classify the models into three groups. First, in the *homogeneous model* the runtime environments (RE) of the mobile and the server are identical, and the task code is present in both locations (Figure 1a). The mobile can execute the task independently when there is no network connectivity. One key aspect of this model is the same RE in the mobile and server that is necessary to encapsulate the application state (AS) in the mobile, such that AS can be transferred in the network and reconstructed in the cloud to execute the task. Second, in the *heterogeneous model*, the mobile and the cloud have a different RE (Figure 1b), and therefore a different implementation of the task. The mobile is independent of its server counterpart and has a simpler implementation of the task. Thus, the mobile can execute the task without network connectivity, but the result is not as accurate as that produced by the cloud. Moreover, in this model, only input parameters of the code are transferred in the communication. Finally, in the *neutral model* the RE is not relevant when outsourcing a task (Figure 1c). The code of the task is uniquely located in the server, but it can be called from the mobile client. Hence, the mobile application cannot provide independent functionality when there is no network connectivity. Our work uses the homogeneous model.

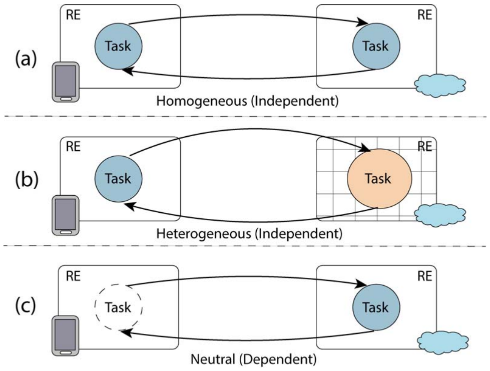


Fig. 1: Implementation flavors of code offloading in practice.

B. Computational Provisioning

Cloud computing is a style of computing in which, typically, scalable resources on demand are provided *as a service (aaS)* over the Internet to users who need not have knowledge of, expertise in, or control over the cloud infrastructure that supports them [10]. Cloud provisioning can occur at different levels, where each level enables the user to interact with the service at a certain granularity. The most common levels are the Infrastructure (IaaS), Platform (PaaS) and Software (SaaS).

Computational provisioning for smartphones happens at Infrastructure level through instances. An instance is a physical or virtualized server, which is associated to a type indicating the computational capabilities, e.g., memory size, number of processors, etc. An instance follows the utility computing model, where consumers pay on the basis of their usage and type preferences. The instance cost is dependent on its type.

The type of instance is important as it determines the acceleration in which a computational task is executed, which impacts the overall response time of the app perceived by the user. Unfortunately, the acceleration level is not obvious. The type of instance also defines its capacity to handle multiple code offloading requests at once. Certainly, speeding up code execution depends on how the code is written. However, higher types of instances can process a task faster than lower ones as higher types can rely on larger memory span and higher parallelization in multiple processors.

III. Related work

Network tuning — Several past work study how cellular network operators can estimate metrics to support better service provisioning for OTT apps. Measured impact of bitrate, jitter and delay on VoIP call session is used along with machine learning to estimate users' satisfaction of the service [11]. Session length and abandonment rate is analyzed from collected history data to understand and troubleshoot web browsing QoE in mobile browsers [12]. Also video streaming has been analyzed to develop adaptive models that improve user engagement [13]. Learning algorithms are used on collected traffic to relate network metrics with the QoE of arbitrary apps [3, 14]. In this work, we propose a technique to complement these studies. Our proposal does not focus on improving the network communication, but rather on accelerating the code of a mobile application on the fly though a software-defined approach. Different levels of code acceleration can be applied to mitigate high communication latency. We assume LTE communication (Refer to section VI-C).

Mobile instrumentation — There have been several efforts towards optimizing communication protocols and fixing inefficiencies in mobile apps. These include analyzing performance of different protocols [15, 16], e.g., TCP, HTTP, intercommunication of different layers [17], and developing better smartphone web browsers [18]. Our abstracting code offloading into a higher software-defined layer, such that the execution of code of a mobile application installed in multiple devices can be dynamically managed. The ultimate goal is to ensure that the response time of tasks that require a lot of computational processing can be normalized across different types of devices. The improvements in communication discussed above can be applied together with our system to speed up both communication and computation.

Resource allocation — The response time of an app has been modeled in terms of latency [19]. It has been demonstrated, in turn, that an application can be adapted to multiple levels of fidelity depending on the mobile resources allocated for execution [4], which translates into multiple scales of responsiveness. More recent work have studied extending battery life by outsourcing tasks to a more powerful infrastructure, e.g., cloud. Many frameworks for code offloading have been proposed [20–30]. Approaches for dynamic provisioning of cyber foraging [9] have been partially explored [31]. We provide a utility model for code execution in the cloud. We introduce a new software-defined component in a code offloading architecture to control code acceleration on the fly.

Besides a few works that focus on scaling up (vertical scaling) a server to parallelize the code of computational requests [29], we are not aware of architectures nor evaluations in the literature that attempt to scale a code offloading architecture in a horizontal fashion. This can clearly be seen as current frameworks propose a one server per smartphone architecture [5], which is infeasible if we consider the amount of smartphones nowadays and the provisioning cost of constantly running a server for each user. Most of the related work to optimize resource allocation in the cloud is oriented to web applications [7, 32]. Thus, we also complement these works by supporting multi-tenancy in our software-defined component. We incorporate a model that is able to predict the amount of users that the system needs to handle each hour. Moreover, the model also reduces overprovisioning by estimating the amount of resources needed to handle the predicted number of users.

IV. System Design

We design our system under the following assumptions.

(a) It uses a homogeneous model for code offloading.

(b) The granularity of code migration is at method level.

(c) Communication with the cloud happens via 3G LTE. Thus, the size of the data transferred and network latency do not incur overhead in the offloading process (See results, section VI-C).

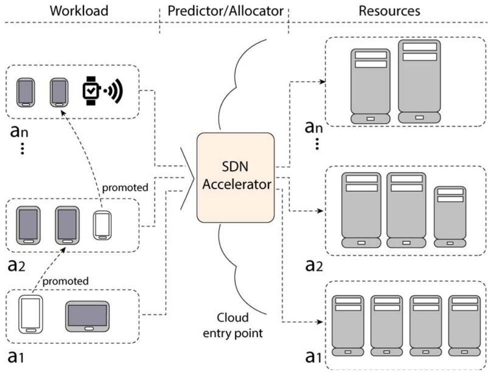


Fig. 2: Overview of the system for SDN code acceleration.

(d) Our system focuses on performance, specifically moderating response time.

An overview of our system is depicted in Figure 2. The system is modeled in three parts, 1) workload, 2) SDN-accelerator and 3) pool of computational resources. The workload contains the active mobile users that outsource a computational task to the cloud. The workload is dynamic, which means that the amount of users is variable across time. Moreover, as shown in the Figure, each mobile device in the workload can be promoted to request a higher level of acceleration when response times are getting longer. In this context, a promotion means that a device offloads to a higher acceleration group.

The SDN-accelerator is the gateway or front-end that receives the workload and routes the outsourced tasks to the right level of code acceleration requested by each mobile device. The pool of computational resources represents the back-end. The back-end is formed by multiple types of instances that are allocated per hour. A provisioned instance is billed by hour by most of the cloud vendors, e.g., on Amazon EC2 and Microsoft Azure.

Since the workload of incoming users is dynamic, the SDN-accelerator is equipped with an adaptive model that estimates the size of the workload at the end of each provisioning hour. Moreover, since the allocation of instances has a cost, the model uses the predicted workload to determine the combination of instances that need to be allocated in order to reduce the provisioning cost of the cloud. Naturally, the allocated instances contain enough computational resources to satisfy the acceleration demand of the workload.

A. Adaptive model

The model consists of two parts, workload prediction and dynamic resource allocation. The aim of the prediction is to estimate the number of incoming mobile users to the system during an interval of time t. The goal of the allocation is to minimize the cost of allocated cloud resources while handling the predicted workload during the interval.

Our model learns to predict users' workload based on previous observations extrapolated from the logs of the system.

The logs store information about each request processed by the system as a trace, which contains the following parameters as a key-value pairs, *<timestamp, user-id, acceleration-group, battery-level, round-trip-time>*. The traces are sorted in chronological order and transformed into a set of time slots. Let T be a set of time slots $T = \{t_i \in T : 1 \leq i \leq H\}$, where t_i are of equal length, and H is the amount of stored history available. The model supports any length of a time period, defined in (fractions of) hours.

Each time slot $t_i \in T$ consists of a set of acceleration groups. Let A represent the set of acceleration groups $A = \{a_n \in A : 1 \leq n \leq N\}$. The model encapsulates the servers of the cloud into acceleration groups. Each a_n is mapped to a set of servers that provide a specific level of code acceleration. For instance, the level of acceleration of a_1 can be provisioned by instances, m1.micro and m1.small, and the level of acceleration of a_2 can be provisioned by m1.large and m2.medium. The level of acceleration of a particular server is determined via benchmarking, discussed in subsection VI-A.

Let U depict the load of incoming mobile users. $U = \{u_m \in U : 1 \leq m \leq N\}$. Each user is assigned to an acceleration group a_n. Initially, each user u_m is located in the group that provides the lowest acceleration of code. A user u_m is gradually promoted in a sequential manner to a higher acceleration group each time the mobile detects lower quality of response time in the mobile application. As a result, each acceleration group at a time period t contains a certain number of users or an empty set. Thus, $a_n = \{\emptyset | W_{a_n} \subseteq U : 1 \leq n \leq N\}$, where W_{a_n} depicts the workload in terms of number of users in the system that require a level of acceleration a_n.

B. Prediction

T provides the evidence for the knowledge base $\vec{P}$ that is used for workload prediction. Given an input of a time slot t_h that models the current workload of the system, the model predicts the next time slot t'_h that represents the expected workload that the system needs to handle for the next period.

Let $\vec{P} = \{\vec{p_k} \in \vec{P} : 1 \leq k \leq| T |\}$, where $\vec{p_k}$ is the *edit distance* [33] between t_k and each $t_i \in T$.

1) Distance Metric: Given two timeslots, i.e., $t_x, t_z \in T$, where $t_x = \{(a_1^x), ..., (a_n^x)\}$ and $t_z = \{(a_1^z), ..., (a_n^z)\}$, we define the distance δ between two acceleration groups (a_r^x) and (a_r^z), where $1 \leq r \leq N$, as

$$\delta((a_r^x),(a_r^z)) = \begin{cases} 0, if((a_r^x) = (a_r^z)) \\ D, otherwise \end{cases}$$

where, $D > 0$ is the edit distance between (a_r^x) and (a_r^z) based on the assigned users u_m, respectively. Accordingly, we define the edit distance Δ between two time slots $t_x, t_z \in T$, as $\vec{p_k}$

$$\vec{p_k} = \Delta(t_x, t_z) = \sum_{r=1}^{n} \delta((a_r^x),(a_r^z))$$

2) Approximation: Once $\vec{P}$ is computed, t'_h is approximated to the timeslot t_k that has the minimum $\Delta \in \vec{P}$.

Note that since t_k is chosen from the history, dramatically growing loads are only ever matched to the largest load seen in the near history. This makes allocation more conservative. Figure 8 in section VI-B shows how our system behaves when a quickly growing load is introduced.

C. Allocation

Given an input t_h that defines the expected incoming workload to the system, the model minimizes the cost of allocating cloud resources to handle it. The dynamic amount of allocated resources is estimated using *Integer Linear Programming (LP)* [34]. The parameters of the model include:

- c_s, cost of an instance of type s. The cost of the instance is a pre-defined price that is set by the cloud vendor for one hour of usage[1].
- K_s, capacity of the instance of type s in requests per minute. This value is found via benchmarking the cloud servers. The results in section VI-A show a method to classify cloud servers into different levels of acceleration based on dynamic workload. We validate this finding with experiments using a static workload. In a real deployment, K_s can be determined from the request traces (logs) of the server.
- CC, number of instances that the cloud can launch for a single account. Amazon allows a maximum of 20 instances a standard level account. Generally, public clouds such Amazon AWS can launch at most 20 instances on demand, if more than 20 instances need to be launched the customer has to fill a form requesting the extra resources.
- W, the value of the workload to which the system needs to adapt. W is approximated by the prediction model presented in IV-B. W can be expressed in terms of sub workloads W_{a_n}, which represent the workload for a particular acceleration group $a_n \in A$. Hence, $W = \sum_{i=1}^{k} W_{a_i}$, where k is the number of acceleration groups in A.

To optimize the number of instances to be allocated, we denote the number of instances of type s to be allocated in the back-end x_s. We assume the front-end is provided by the cloud vendor, e.g. Amazon Autoscale.

The model tries to minimize the cost of all instances x_s of type s of cost c_s. The objective function is defined as the sum across all the instances types $s \in S$.

$$Min \sum_{i=1}^{n} x_{s_i} c_{s_i} \tag{1}$$

Lastly, the model comprises the following constraints:

- The workload constraint ∀ acceleration groups $a_n \in A$:

$$\sum_{n=1}^{k} \sum_{i=1}^{n} x_{s_i} K_{s_i} > W_{a_n} \tag{2}$$

[1] https://aws.amazon.com/ec2/instance-types/

- The cloud service's maximum number of instances:

$$\sum_{i=1}^{n} x_{s_i} < CC \tag{3}$$

The workload constraint states that the sum of all the capacity K_s across all the instances from x_{s_i} must be enough to satisfy the workload W_{a_n} requiring acceleration a_n. The cloud service's maximum number of instances limits the number of running instances of type s to the allocation capacity CC.

1) Software-defined Code Acceleration: Since our adaptive model abstracts the cloud resources into acceleration groups, it is possible to define on the fly, e.g., by the administrator, the minimum level of code acceleration provisioned in *as a service* fashion by the SDN component. This minimum level is modeled in terms of response time of the request, which is determined through a performance-based characterization over each instance available for allocation. This minimum value also influences the acceleration groups of the SDN-accelerator.

The response time of code execution of an instance depends on the capacity of the instance K_s to process concurrent requests (Refer to results in VI-A). Thus, when the minimum level of acceleration is defined, e.g., 500 milliseconds, all the available instances are sorted in an ascending manner based on their capacity to handle that response time, e.g., a small instance handles a maximum of 30 users under 500 milliseconds while a large instance handles a maximum of 90 users under 500 milliseconds. Once sorted, an acceleration group is created for each capacity. Instances with the same capacity are assigned to the same group.

V. Implementation

The components implemented in the system are depicted in Figure 3. To emulate a large number of smartphones offloading at the same time, we develop a simulator. The simulator instruments client code at method level using Java reflection, and generates different requests of mobile devices offloading code to cloud (aka workload).

The simulator creates workload in two different operational modes, 1) concurrent and 2) inter-arrival rate. In the concurrent mode, the simulator creates *n* concurrent threads that offload a random computational task loaded from a pool of common algorithms found in apps, e.g., quicksort, bubblesort. Each thread represents a mobile device offloading a task. This mode is utilized to benchmark cloud instances. In an inter-arrival rate mode, the simulator takes as parameters the number of devices (workload), the inter-arrival time between offloading requests and the time that the workload is active. This mode is utilized to produce a realistic time-varying workload.

The SDN-accelerator contains a *Request Handler* (RH) that is the entry point to process code requests from the workload. When the RH receives a request, it creates a new thread to handle that request via the *Code Offloader* (CO). The CO determines the level of acceleration required and routes the request to the corresponding group of instances. The CO also logs information about each request processed into a MySQL database. The *Workload Predictor* (WP) implements the prediction model described in IV-B. The model is implemented in R using the *RecordLinkage* package. Lastly, the predicted workload obtained by WP is passed to the *Resource Allocator* (RA) that runs the allocation model described in subsection IV-C. Similarly, the allocation model is implemented by relying on the *lpSolveAPI* of R.

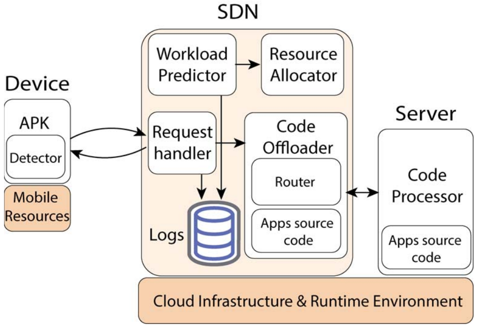


Fig. 3: Components of the system.

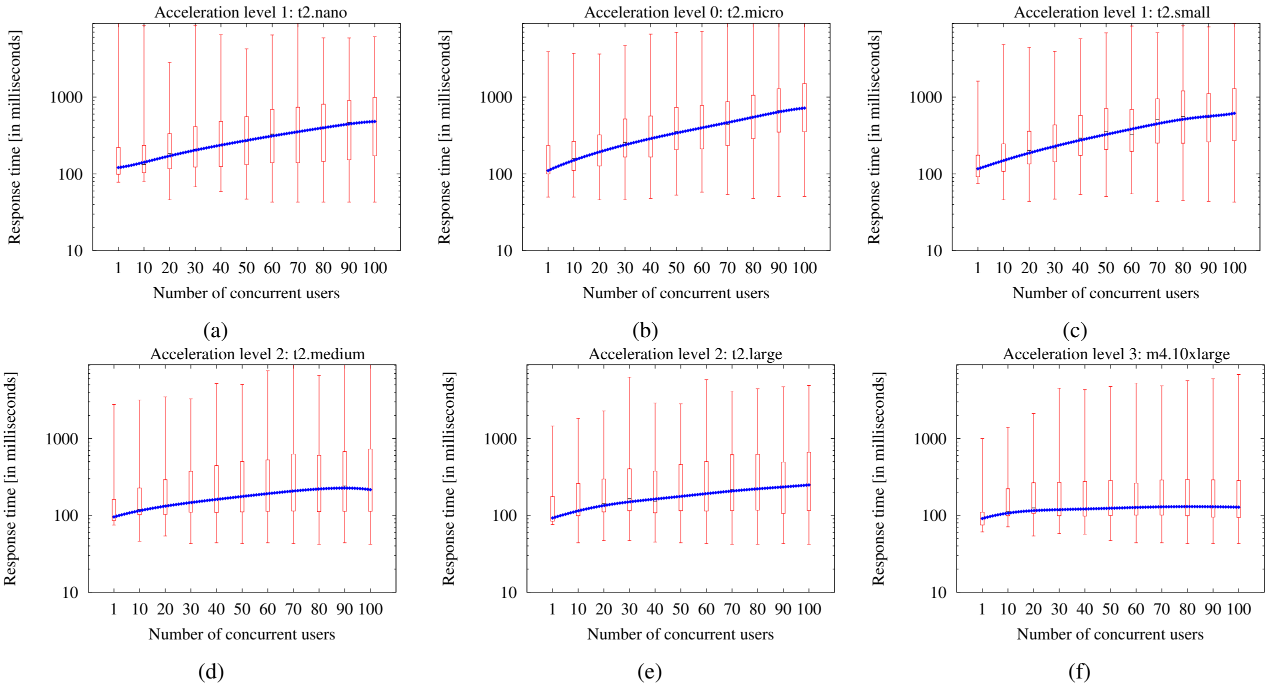
Since our system implements the *homogeneous model* for code offloading, each instance in the back-end contains a customized Dalvik-x86 (Released as public in Ireland region of Amazon EC2 as *ami-ac8813df*) that can be launched on demand. It is built from the source code of *A*ndroid *O*pen *S*ource *P*roject (AOSP) over the instance to target a x86 server architecture, and removing the *Applications* and *Application Framework* layers from the Android software architecture.

Dalvik-x86 is necessary to characterize the processing level of each cloud server when facing a large number of offloading requests. This is not possible to achieve solely with Android-x86 as the mobile operating system imposes restrictions, e.g., not allowing multiple instances of the same application. Our Dalvik-x86 is lighter when compared with other surrogates used by others, e.g., Android-x86. This reduced the storage size required by our Dalvik-x86 surrogate by 40%. Moreover, it does not active any default processes of the OS, e.g., Zygote or GUI Manager, which are not needed by the surrogate. The surrogate creates a *dalvikvm* process in the host machine per each offloading request that needs to be handled. This approach enables troubleshooting problematic requests by process id without restarting or stopping the system.

Dalvik-x86 implements an executable script wrapper at the core of libraries that boot the compiler. The wrapper provides an interface to push APK files into the virtual machine, such that the code inside of the APK can be executed. When the server initiates, the available APK files (in a OS folder) are pushed into the Dalvik-x86 as the process is waiting for a request. Each APK can be instantiated multiple times in different ports.

VI. Evaluation

In this section, we evaluate our proposed approach, which is implemented and deployed in a real testbed in Amazon EC2 (Ireland). We used general purpose instances for our deployment (t2.nano, t2.micro, t2.small, t2.medium, t2.large



(a)

(b)

(c)

(d)

(e)

(f)

Fig. 4: Cloud-based servers are grouped by acceleration level, determined by performance degradation as more users are added.

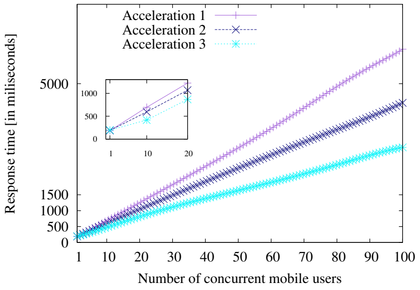


Fig. 5: Differences between the levels of acceleration.

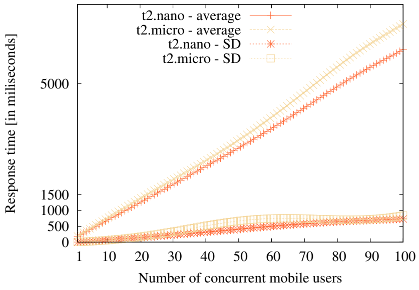


Fig. 6: Anomaly between t2.nano and t2.micro.

and m4.10xlarge). We equipped our simulator with a pool of 10 independent tasks for creating computational workload. The simulator is configured in different operation modes with each experiment to evaluate different aspects of the system, e.g., the concurrent mode is used for characterizing the cloud servers and inter-arrival mode to analyze load distribution.

A. Cloud Provisioning and Acceleration Levels

Past work demonstrated the offloading potential in lab setups, where one device offloads to one server (without heavy workload). However, in the wild, heavy workload caused by multiple active users impacts the system's throughput. Thus, to characterize the execution of code in the cloud, first we answer the question: *what is the effect of code execution when outsourced to the cloud by multiple devices?*

1) Setup and methodology: We configure our simulator in concurrent mode to stress the instances with a heavy load of requests. In this experiment, each request that is created by the simulator is taken randomly from the pool. The processing required for each task is also determined randomly. The random nature of the experiment is important to verify all possible cases that can influence the processing of a task by a server. We conduct a 3 hours experiment per server to ensure coverage. We verify this characterization by relying on static load. We evaluate the influence of increasing users' load by configuring the workload from 1 to 100 in intervals of 10 users for each server (load levels 1,10,20,30,40,50,60,70,80,90 and 100). Concurrent load is generated with an inter-arrival rate of 1 minute. This means that the maximum load (load=100) used for characterizing a server is ≈ 18000 (3*60*100) requests and the minimum load (load=1) is ≈ 180 requests. The purpose of the 1 minute inter-arrival is to give the server enough cool down time before stressing it again.

2) Benchmark: Figure 4 shows the results of the experiments. We can observe how the response time of the requests is distributed through the interpercentile range of the processed load. The slope of the mean response time becomes less steep as we use more powerful instances. This suggests that the response time of a request is defined by the type of the server. Based on this property, we characterize each server into an acceleration group. We find that servers can be classified to 3 acceleration groups. This is important as servers with different costs provide the same level of acceleration. Thus, server selection needs analysis a priori.

3) Acceleration Levels: It is well known that an outsourced task gets accelerated as a cloud server can process a task faster than a mobile device. Our results also confirm that statement. However, *is it possible to determine how fast a task can be executed?*. We answer this question by analyzing the levels of acceleration of static load. We use a minimax algorithm with static input as load. Figure 5 presents the results: we can observe the differences between acceleration levels. A task is executed ≈1.25 times faster by a server of level 2 when compared with one of level 1. Similarly, a task is accelerated ≈1.73 times by a server of level 3 compared with level 1. The difference between levels 3 and 2 is also significant (≈1.36 times acceleration).

4) Cloud Vendor: While the characterization shows that the response time of a request stabilizes as the capabilities of the server increase, we found out that this is not the case of a particular instance (t2.nano). Figure 6 shows this anomaly. The resources of a nano server are lower than a micro server according to Amazon. However, a nano server provides better performance to handle load than a micro server. Since there is no information about how Amazon allocates its resources, it is difficult to deduce the cause of this anomaly. Interestingly, a micro server is provided as *free tier eligible*, while a nano server is not in this category. Consequently, we assigned a micro server in a lower acceleration level (group 0).

B. Mobile Performance and SDN Routing

1) Setup and Methodology: Our system introduces an extra front-end component (SDN-accelerator) in the architecture. We explore *how SDN-accelerator influences the total response time of a request*. As a result, timestamps are taken across the system as the request is processed in each of its components. We configure our simulator to generate a concurrent load of 30 users to the SDN-accelerator. In addition, we analyze the effect of dynamic workload processed by a server (t2.large). We explore how drastic changes in inter-arrival rate of requests impact in the throughput of the server. We configure our simulator to produce dynamic load that increments the inter-arrival rate of requests each 5 minutes from 1 to 1024 Hz.

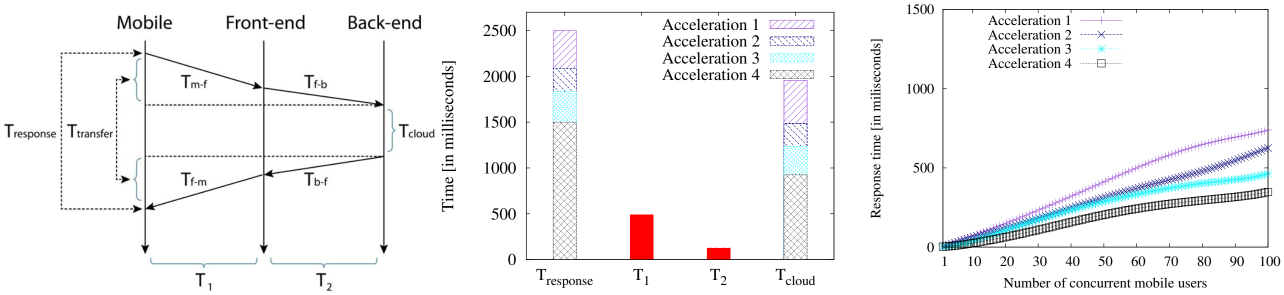
2) Dynamic Forwarding: Figure 7a models the response time $T_{response}$ of a computational request. The response time considers the time it takes to connect from the mobile to the front-end T_{m-f}, the time to route the request to a particular instance in the back-end T_{f-b}, the execution time of the code in the instance T_{cloud}, the time to send the result back to the front-end T_{b-f}, and finally, the time for the result to arrive at the mobile device T_{f-m}. We assume that $T_{m-f} = T_{f-m}$ and $T_{f-b} = T_{b-f}$ are equal as the same communication channel remains open both ways until the operation finishes. In this context, we define $T_1 = T_{m-f} + T_{f-m}$ and $T_2 = T_{f-b} + T_{b-f}$. Thus, the response time $T_{response} = T_1 + T_2 + T_{cloud}$.

Figure 8a shows the routing time of load by the SDN-accelerator. We can observe that the overhead introduced by the front-end is ≈ 150 milliseconds. We explore the influence of each component in the total response time using its average processing time. Figure 7b shows the timestamps taken across the system. We can see that the total communication time $T_1 + T_2$ is less than a second. Naturally, higher latency can increase the communication time and vice versa, which impacts T_1. On the other hand, T_2 is less likely to change drastically as the latency results from the internal cloud communication, between servers in the same private network. Lastly, the diagram shows that T_{cloud} is the most time consuming operation. Fortunately, the total time of the code invocation in the cloud can be decreased by adjusting the tradeoff between utilization price and computational capabilities of the instance. Figure 7c shows how stability of code execution speed based using the standard deviation of each group of acceleration. To corroborate the veracity of this statement, we allocate an additional c4.8xlarge instance, a memory optimized server with higher performance than the other instances. This instance surpassed our previous acceleration levels, so we classified it as Acceleration level 4.

3) Overwhelming Workload: Figure 8b shows the average response time of the workload as request arrival rate is doubled every 5 minutes. We can observe that when the server reaches its maximum capacity to handle a particular inter-arrival rate ($32Hz$ in our case study), each consecutive increment in workload will dramatically reduce server performance and cause worse response time. If we assume that the system is static and not dynamic, then we can also observe that the server continues degrading performance as workload increases until it collapses. Saturating the server also leads to dropped requests, shown in Figure 8c. Beyond 32 Hz, an increasing amount of requests is dropped. The results show that a severe increase in workload (from 2 Hz to 128 Hz) is needed to bring down the system. Fortunately, such changes in the offloading workload are easily detected, as its effect can be directly seen in the response time of applications [35]. Thus, it is possible to adapt with minimal loss of performance without the need to over-provision the system. We can see the effect of dynamic workload adjustment in Figure 9b, compared with no adjustment in 9c. These are discussed in detail in Section VI-C.

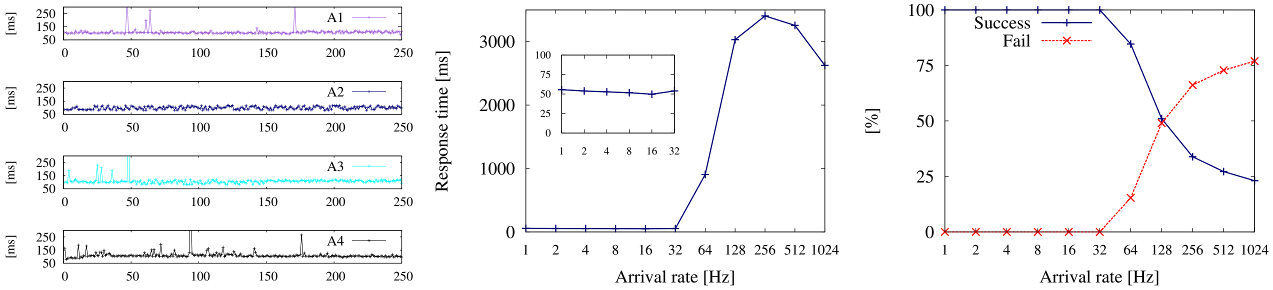
C. Model Evaluation

In addition to simulated load, we were interested to analyze the influence of real patterns of smartphone usage in large scale scenarios. Hence, we developed a mobile application, which tracked and recorded the sessions of the mobile applications initiated in a particular mobile device.



(a) (b) (c)

Fig. 7: System performance. (a) Timestamps taken across the system in each of its components, (b) Actual times to handle the request in each component, (c) Standard deviation of the each acceleration level.



(a) (b) (c)

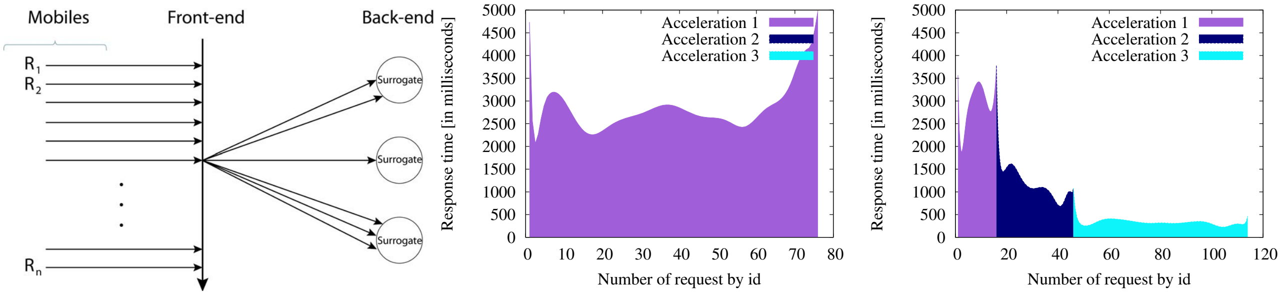
Fig. 8: Workload management. (a) Routing time of SDN-accelerator by acceleration level. (b) Response time of requests in terms of server throughput when speeding up arrival rate. (c) Amount of requests processed by a server based on arrival rate (success vs fail).

1) Setup and Methodology: The application was deployed in the smartphone of 6 participants during 3 months. The data from this study was used to detetermine a realistic inter-arrival rate that can be used by the simulator to produce load. By combining the data of these 6 participants, we find that an inter-arrival rate between $(100-5000)$ milliseconds. Naturally, we removed long inactive periods of a user (during night) from the data. We used this time-varying inter-arrival rate to create a load of 100 users. The goal of this experiment is to answer the question, *is it possible to control the level of code execution dynamically?*. For this experiment, we rely on 3 acceleration groups, 1, 2, and 3, handled by instance types t2.nano, t2.large, m4.4xlarge, respectively. We conduct an 8-hour experiment, which produced ≈4000 incoming requests to the SDN-accelerator. Figure 9a depicts the setup. Each request is the same static task (minimax) used previously for the benchmarking of the servers. Additionally, to ensure demonstrating the stability of acceleration performance, we induced a load of 50 concurrent users in each server in the back-end. This concurrent load is created each 2 seconds during the 8-hour experiment.

2) Prediction Accuracy: To determine the accuracy of the model to predict workload, we perform a 10-fold cross validation of the model over history traces produced by a workload of 16 hours using the same inter-arrival rate found from the smartphone usage experiment. Figure 10a shows the results, we can observe that our model requires a bootstrap time before producing high accuracy results. All in all, our model predicts workload with $\approx$ 87.5% accuracy.

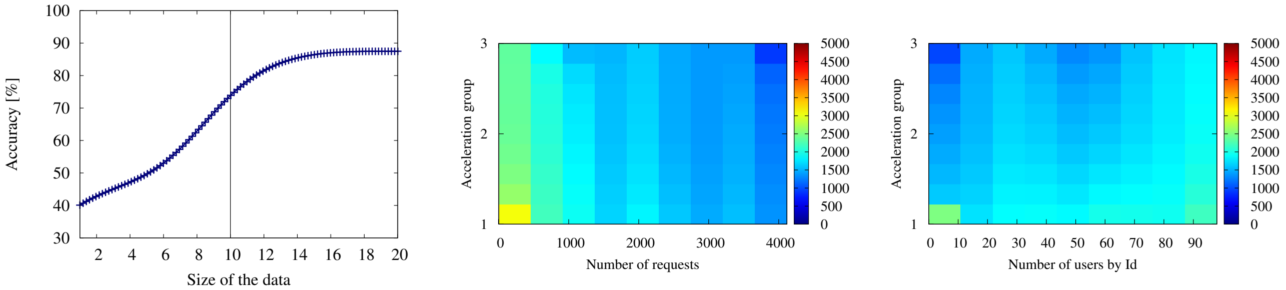
3) User Perception: One key aspect of our approach is that a user is assigned to an acceleration group based on the decision of the mobile client. This opens a wide range of possibilities to improve perceived response time of an application depending on different events sensed by the device. For instance, if the processing of a task in certain device requires more than t milliseconds, then the mobile promotes the user to higher acceleration level. Moreover, by using this method, the SDN-accelerator is released from the overhead of monitoring and tracking users. In this paper, we rely on a static probability of $1/50$ to promote a user. We plan to develop a context-based decision model as future work.

Figures 9b and 9c show the results of the experiment. To demonstrate how the perception changes dynamically, from the load of 100 users, we selected two users, user 32 that was not promoted throughout the experiment (Figure 9b) and user 8 that was promoted to each available level of acceleration (Figure 9c). We can observe that user 32 perceived a stable response time of $\approx$ 2.5 seconds on average, while user 8



(a) (b) (c)

Fig. 9: Dynamic acceleration changes in mobile application performance. (a) System deployment to evaluate the dynamic acceleration of the code, (b) Response time perceived by user 32 whose acceleration group remains the same throughout the experiment, (c) Response time perceived by user 8 whose acceleration group changed from level 1 to 3.



(a) (b) (c)

Fig. 10: (a) Accuracy of the prediction model to estimate the number of users in each acceleration group. Since the accuracy depends on the amount of data available for learning, it is also shown the minimal amount of data to predict workload. (b) Response time perceived by 100 users. (c) Promotion rate of the workload of 100 users.

perceived a shorter response time with each promotion.

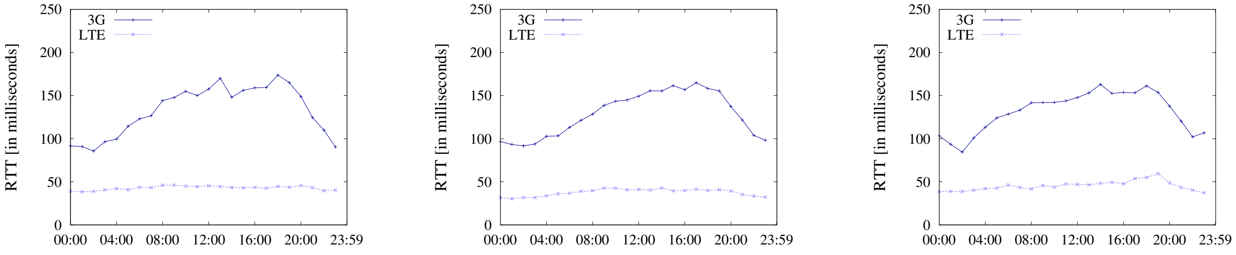
Lastly, Figure 10b summarizes how the perception of the 100 users was dynamically changed during the experiment. As we increase the number of requests, the response time rises until more resources are dynamically allocated. Then response time quickly decreases and stays relatively low as prediction keeps up with increasing load. Figure 10c shows the promotion rate of users. We observe that as users are promoted to higher acceleration groups, overall response time decreases. This result is also confirmed by Figure 10b.

4) Results: communication latency (3G/LTE): — Communication latency is a key factor for the adoption of offloading in the wild. Thus, we analyze 3G and LTE as a means for accessing remote clouds. We rely on the dataset provided by NetRadar [36]. The dataset is collected from 2015 and includes measurements from multiple regions of Finland. Since the latency in the cellular network also depends on the quality of service provided by the vendor, we analyze three different mobile providers α (Samples, 3G=205762, LTE=182549), β (Samples, 3G=448942, LTE=493956), and γ (Samples, 3G=191973, LTE=152605), anonymized for impartiality. Figures 11a (α), 11b (β) and 11c (γ) show the average latency of the communication (RTT) for each provider, respectively. From the results, we can observe that the average RTT using 3G for each cellular operator is about 128 milliseconds for α (SD≈362, median≈51), about 141 milliseconds for β (SD≈376, median≈60) and about 137 milliseconds for γ (SD≈379, median≈56). We also can observe that the average RTT using LTE is about 41 milliseconds for α (SD≈56, median≈34), about 36 milliseconds for β (SD≈70, median≈25) and about 42 milliseconds for γ (SD≈84, median≈27). While there is a notable difference between 3G and LTE, both provide high latency communication to achieve offloading support. In our system, we assume that offloading happens using LTE.

VII. Discussion

Based on our testbed results, we present in this section the benefits, limitations, and lessons learned of our approach.

1) Levels of code acceleration: While we demonstrated that it is possible to control the response time of a task at multiple levels of acceleration using the cloud, the processing of a task also depends on how the code is written for execution. A task may be unable to take advantage of the computational resources of a particular server, e.g., multiple cores and large memory span. Therefore, there is an acceleration limit that a task can achieve. Naturally, this limit can be surpassed by



(a)

(b)

(c)

Fig. 11: Network communication latency and number of collected samples by mobile operator for 3G and LTE technologies. From left to right, a: α, b: β, and c: γ, respectively.

applying techniques of code parallelization [25, 29]. In such a case, other issues arise when modeling the acceleration of code, e.g., optimal splitting and result merging. We leave code parallelization for future work.

2) Offloading to save battery life: Several work have demonstrated that offloading code to cloud can increase the battery life of the mobile device [5]. The effect of offloading in cellular networks (3G/LTE) and WiFi networks has been widely explored [37]. Moreover, we conduct a large-scale analysis under 3G and LTE technologies. As consequence, we assume stable LTE communication that provides a cloudlet-like latency. Thus, by cloudlet definition [20], there is no overhead from drastic changes in communication or size of the data transferred. We focus on a specific problem: dynamic moderation of mobile application performance. Hence, the trade-off between latency and energy is out of scope.

3) Code acceleration based on policies: Notice that our work is not ruled by a fixed and simple load balancing algorithm, e.g., round-robin. Our work highly differs from these techniques, as we consider the users' perception when optimizing resources. One key advantage of our approach is that a mobile device can promote itself to a higher acceleration group based on its response time requirements. Naturally, this approach can be generalized to adjust mobile support based on any contextual need. For instance, as the battery level of a device gets lower, it needs to reduce the effort to handle network communication. Thus, it can promote itself to higher acceleration level in order to decrease the amount of time that the connection needs to be open waiting for a result. However, to consider new policies, our model needs to be slightly modified to consider energy measurements. This is necessary to keep the prediction accuracy high.

4) Code acceleration as a service (CaaS): The possibility of accelerating the execution of an application at a specific level opens new opportunities to monetize software. For instance, a user can acquire from the cloud a service to improve the response time of a game instead of buying a new higher capability device. This way, by using our approach, the lifespan of the mobile hardware can be extended.

Naturally, CaaS differs from existing SaaS. In short, SaaS is not optimized for consumption by mobile devices. This is clear as many resource allocation models for cloud applications have been proposed [7, 32]. However, these models do not take into account how the processing time of a cloud service influences the perception of the user. Thus, cloud models for resource allocation in SaaS applications cannot be applied in the context of mobile offloading. For instance, results in section VI-A show that a cloud server is able to handle a large amount of requests from different mobiles. However, as the workload increases, the processing time of the server also increases. Thus, to keep a suitable response time for users, the amount of workload needs to be moderated.

VIII. Conclusions

The analysis of code execution during runtime is a tough challenge as the code by nature is non-deterministic. Thus, it is hard to determine the expected time of a task's execution. While static methods can provide an approximation, the accuracy of those methods is low. In addition, the type of acceleration that can be obtained in the cloud is not simply determined by a server's capabilities. In this paper, we modeled the different levels of acceleration that can be achieved by outsourcing code to cloud. We found that cloud servers can be classified into different groups to provide multiple levels of acceleration as a service. We analyzed general purpose servers, but a wide variety of servers can be benchmarked using the same method. Our work advances the art by proposing a SDN approach to model and control the level in which a task is processed. By using SDN, no extra instrumentation nor modification in software is required to tune the response time of an application. Our work opens a wide variety of opportunities to monetize the acceleration of code in the cloud. Lastly, we provide the source code, Dalvik-x86, model and case study in GitHub[2].

IX. Acknowledgments

The authors thank the anonymous reviewers for their insightful comments. This work is partially funded by the Academy of Finland (Grants 276786-AWARE, 286386-CPDSS, 285459-iSCIENCE, 304925-CARE), the European Commission (Grant 6AIKA-A71143-AKAI), and Marie Skłodowska-Curie Actions (645706-GRAGE).

[2] https://github.com/mobile-cloud-computing/ScalingMobileCodeOffloading

REFERENCES

[1] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?," *IEEE Software Magazine*, vol. 32, no. 3, pp. 70–77, 2015.

[2] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti, "Your installed apps reveal your gender and more!," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 18, no. 3, pp. 55–61, 2015.

[3] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan, "Prometheus: toward quality-of-experience estimation for mobile apps from passive network measurements," in *Proceedings of the 15th ACM Workshop on Mobile Computing Systems and Applications (HotMobile 2014)*, (Santa Barbara, California, US), February 26–27, 2014.

[4] M. Satyanarayanan and D. Narayanan, "Multi-fidelity algorithms for interactive mobile applications," *Wireless Networks*, vol. 7, no. 6, pp. 601–607, 2001.

[5] 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. 4, pp. 80–88, 2015.

[6] K. Kirkpatrick, "Software-defined networking," *Communications of the ACM*, vol. 56, no. 9, pp. 16–19, 2013.

[7] M. Mazzucco, D. Dyachuk, and R. Deters, "Maximizing cloud providers' revenues via energy aware allocation policies," in *IEEE 3rd International Conference on Cloud Computing (CLOUD 2010)*, (Miami, Florida, USA), July 5–10, 2010.

[8] B. Han, P. Hui, V. A. Kumar, M. V. Marathe, J. Shao, and A. Srinivasan, "Mobile data offloading through opportunistic communications and social participation," *IEEE Transactions on Mobile Computing*, vol. 11, no. 5, pp. 821–834, 2012.

[9] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *Proceedings of ACM SIGOPS European Workshop 2002*, (Saint-Emilion, France), July 01, 2002.

[10] H. Flores and S. Srirama, "Mobile cloud middleware," *Journal of Systems and Software*, vol. 92, pp. 82–94, 2014.

[11] P. Brooks and B. Hestnes, "User measures of quality of experience: why being objective and quantitative is important," *IEEE Network Magazine*, vol. 24, no. 2, 2010.

[12] A. Balachandran, V. Aggarwal, E. Halepovic, J. Pang, S. Seshan, S. Venkataraman, and H. Yan, "Modeling web quality-of-experience on cellular networks," in *Proceedings of the Annual international conference on Mobile computing and networking (MobiCom 2014)*, (Maui, Hawaii), September 7–11, 2014.

[13] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang, "Developing a predictive model of quality of experience for internet video," vol. 43, no. 4, pp. 339–350, 2013.

[14] M. Volk, J. Sterle, U. Sedlar, and A. Kos, "An approach to modeling and control of qoe in next generation networks," *IEEE Communications Magazine*, vol. 48, no. 8, 2010.

[15] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 3, pp. 22–36, 1996.

[16] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving tcp performance over wireless links," *IEEE/ACM Transactions on networking*, vol. 5, no. 6, pp. 756–769, 1997.

[17] R. Hsieh and A. Seneviratne, "A comparison of mechanisms for improving mobile ip handoff latency for end-to-end tcp," in *Proceedings of the 9th ACM annual international conference on Mobile computing and networking (MobiCom 2003)*, (San Diego, California, USA), September 14–19, 2003.

[18] L. A. Meyerovich and R. Bodik, "Fast and parallel webpage layout," in *Proceedings of the 19th ACM international conference on World wide web (WWW 2010)*, (Raleigh, NC, USA), April 26–30, 2010.

[19] D. Narayanan, J. Flinn, and M. Satyanarayanan, "Using history to improve mobile application adaptation," in *IEEE Workshop on Mobile Computing Systems and Applications*, (Monterey, CA, USA), December 7–8, 2000.

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

[21] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: Bringing the cloud to the mobile user," in *Proceedings of ACM MobiSys Workshop 2012*, (Low Wood Bay, Lake District, United Kingdom), June 25–29, 2012.

[22] H. Flores and S. Srirama, "Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning," in *Proceedings of ACM MobiSys Workshop 2013*, (Taipei, Taiwan), June 25–28, 2013.

[23] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the ACM International Conference on Mobile systems, applications, and services (MobiSys 2010)*, (San Francisco, CA, USA), June 15–18, 2010.

[24] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the Annual Conference on Computer systems (EuroSys 2011)*, (Salzburg, Austria), April 10–13, 2011.

[25] 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 Annual IEEE International Conference on Computer Communications (INFOCOM 2012)*, (Orlando,

Florida, USA), March 25–30, 2012.

[26] H. Flores and S. Srirama, "Mobile code offloading: should it be a local decision or global inference?," in *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services 2013 (MobiSys 2013)*, (Taipei, Taiwan), June 25–28, 2013.

[27] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "Cosmos: Computation offloading as a service for mobile devices," *Procedings of the ACM International Symposium of Mobile Ad Hoc Networking and Computing (MobiHoc 2014)*, August 11–14, 2014.

[28] 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 USENIX Conference on Operating Systems Design and Implementation (OSDI 2010)*, (Hollywood, CA, USA), October 08–10, 2012.

[29] M.-R. Ra *et al.*, "Odessa: enabling interactive perception applications on mobile devices," in *Proceedings of the ACM International Conference on Mobile systems, applications, and services (MobiSys 2011)*, (Washington, DC, USA), June 28 – July 1, 2011.

[30] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading for pervasive computing," *IEEE Pervasive Computing*, vol. 3, no. 3, pp. 66–73, 2004.

[31] M. D. Kristensen and N. O. Bouvin, "Scheduling and development support in the scavenger cyber foraging system," *Pervasive and Mobile Computing*, vol. 6, no. 6, pp. 677–692, 2010.

[32] M. Mazzucco and M. Dumas, "Achieving performance and availability guarantees with spot instances," in *Proceedings of the International Conference on High Performance Computing and Communications (HPCC 2011)*, (Banff, Canada), September 2–4, 2011.

[33] A. Marzal and E. Vidal, "Computation of normalized edit distance and applications," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 9, pp. 926–932, 1993.

[34] H. A. Taha, *Operations research: an introduction*, vol. 557. Pearson/Prentice Hall, 2007.

[35] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD 2011)*, (Washington DC, USA), July 4–9, 2011.

[36] S. Sonntag, J. Manner, and L. Schulte, "Netradar-measuring the wireless world," in *Proceedings of IEEE International Symposium on Modeling & Optimization in Mobile, Ad Hoc & Wireless Networks (WiOpt 2013)*, (Tsukuba Science City, Japan), May 13–17, 2013.

[37] H. Flores, R. Sharma, D. Ferreira, V. Kostakos, J. Manner, S. Tarkoma, P. Hui, and Y. Li, "Social-aware hybrid mobile offloading," *Pervasive and Mobile Computing Journal*, vol. 36, pp. 25–43, 2017.