Evidence-Aware Mobile Computational Offloading

Huber Flores, *Member, IEEE*, Pan Hui, *Fellow, IEEE*, Petteri Nurmi, Eemil Lagerspetz, Sasu Tarkoma, Jukka Manner, Vassilis Kostakos, Yong Li, and Xiang Su

Abstract—Computational offloading can improve user experience of mobile apps through improved responsiveness and reduced energy footprint. A fundamental challenge in offloading is to distinguish situations where offloading is beneficial from those where it is counterproductive. Currently, offloading decisions are predominantly based on profiling performed on *individual devices*. While significant gains have been shown in benchmarks, these gains rarely translate to real-world use due to the complexity of contexts and parameters that affect offloading. We contribute by proposing *crowdsensed evidence traces* as a novel mechanism for improving the performance of offloading systems. Instead of limiting to profiling individual devices, crowdsensing enables characterizing execution contexts across a community of users, providing better generalisation and coverage of contexts. We demonstrate the feasibility of using crowdsensing to characterize offloading contexts through an analysis of two crowdsensing datasets. Motivated by our results, we present the design and development of the EMCO toolkit and platform as a novel solution for computational offloading. Experiments carried out on a testbed deployment in Amazon EC2 Ireland demonstrate that EMCO can consistently accelerate app execution while at the same time reduce energy footprint. We also demonstrate that EMCO provides better scalability than current cloud platforms, being able to serve a larger number of clients without variations in performance. Our framework, use cases, and tools are available as open source from GitHub.

Index Terms—Mobile cloud computing, code offloading, cloud offload, big data, crowdsensing, mobile context modeling

1 Introduction

Computational offloading is a powerful mechanism for improving responsiveness and battery efficiency, two critical factors in the acceptability and usability of mobile applications [1]. By outsourcing (parts of) computationally intensive tasks to dedicated computing infrastructure, devices can reduce burden on their own resources while benefiting from resources provided by the dedicated infrastructure. Several platforms for providing offloading support have been proposed in the literature with the main difference being the level at which offloading is enabled. Examples of proposed mechanisms include annotation [2], [3], thread-migration [4], thread-migration through data shared memory (DSM) [5], and various others [6], [7], [8].

- *H. Flores is with the Department of Computer Science, University of Helsinki, Helsinki 00100, Finland. E-mail: huber.flores@helsinki.fi.*
- *P. Hui is with the University of Helsinki, Helsinki 00100, Finland, and the Hong Kong University of Science and Technology, Hong Kong. E-mail: panhui@cse.ust.hk.*
- *P. Nurmi, E. Lagerspetz and S. Tarkoma are with the University of Helsinki, Helsinki 00100, Finland. E-mail: {petteri.nurmi, eemil.lagerspetz, sasu.tarkoma}@helsinki.fi.*
- *J. Manner is with Aalto University, Espoo 02150, Finland. E-mail: jukka.manner@tkk.fi.*
- *V. Kostakos is with the University of Melbourne, Parkville, Victoria 3010, Australia. E-mail: vassilis.kostakos@oulu.fi.*
- *Y. Li is with Tsinghua University, Beijing 100084, China. E-mail: liyong07@tsinghua.edu.cn.*
- *X. Su is with the University of Oulu, Oulu 90014, Finland. E-mail: xiang.su@ee.oulu.fi.*

Manuscript received 28 Mar. 2017; revised 26 Oct. 2017; accepted 7 Nov. 2017. Date of publication 24 Nov. 2017; date of current version 29 June 2018. (Corresponding author: Pan Hui.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMC.2017.2777491

A key challenge in computational offloading is to distinguish situations where offloading is beneficial from those where it is counterproductive. For example, offloading network intensive computing tasks is only beneficial when the network link is sufficiently fast and stable whereas offloading parts of applications that require interactivity (e.g., augmented reality) is often counterproductive as energy gains are offset by decreased responsiveness resulting from increased latency [9]. Making optimal decisions, however, is far from straightforward as a wide range of factors can influence the effectiveness of offloading. For example, the extent of energy savings from reduced CPU usage depend on temperature and the number of applications running on a device [10]. Similarly, network type, device model, and the capacity and current load of the cloud surrogate have a major impact on offloading performance [11], [12]. Quantifying and characterizing the effect of these factors on a *single* device is clearly infeasible due to the inherent complexity and diversity of these factors. Indeed, in our experiments we demonstrate that quantifying individual factors, such as characterizing whether a particular mobile application requires heavy processing or characterizing the effect of network type, requires a large amount of samples (data from over 100 different apps or over 6,000 RTT samples). While individual devices cannot capture such amounts easily, harnessing a *community* of devices would make the process feasible. A population of 100 devices, which is easily obtainable within a university or even a company, would be capable of collecting the required measurements within a week or less, and a community of 1,000 devices, which is increasingly possible through app store deployment [13], could collect the required measurements in less than a day.

We contribute by developing *evidence-aware mobile computational offloading* (EMCO) as a novel mechanism for optimizing the performance of computational offloading

frameworks. The key novelty in EMCO is the use of *crowdsensed evidence traces* to characterize the influence of different contextual parameters and other factors on offloading decisions [14]. Contrary to existing solutions, which either rely on static code profiling performed on individual devices [2], [3], [4], [5], [6], [11], [15], [16], [17] or on parametrized models that consider a handful of parameters such as network latency, remaining energy, and CPU speed [9], [18], the use of crowdsensing enables EMCO to quantify and characterize the effect of a wide range of parameters and how they vary over execution contexts. EMCO models the context where offloading decisions are made is through simple dimensions that are easy to scale, and determines optimal dimensions using an analytic process that characterizes the performance of offloading based on contexts captured by the community. EMCO also supports identifying reusable and profitable code invocations that can be cached locally to further accelerate performance. We evaluate EMCO through extensive benchmarks conducted on an Amazon EC2 testbed deployment. Our results demonstrate that EMCO significantly accelerates mobile app performance, resulting in over five-fold improvements in response time and over 30 percent reductions in energy. When pre-caching on the client is used, the magnitude of savings is even higher with up to 30-fold improvements in response rate being possible.

The contributions of the paper are summarized as follows:

- We propose *crowdsensed evidence traces* as a novel mechanism for optimizing offloading performance. By aggregating samples from a larger community of devices, it is possible to learn optimal execution contexts where to offload. We analyze two crowdsensing datasets, Carat [19] and NetRadar [20], to demonstrate that it is possible to use crowdsensing to accurately quantify the effect of different factors on offloading performance. We also demonstrate that carrying out the profiling on a single device is not feasible due to the number of samples required.
- We present the design and development of the Evidence-Aware Mobile Computational Offloading platform and toolkit. We demonstrate that EMCO enables apps to accelerate their response time up to five-fold, and up to thirty-fold with pre-caching. We also demonstrate that EMCO can achieve further reductions in energy footprint through optimal selection of the cloud surrogate.
- We develop LAPSI[1] as a novel cloud-based runtime environment that supports characterization of surrogate performance. LAPSI is based on the Dalvik virtual machine which we have directly integrated with a cloud server. This creates a lightweight compiler for executing code and provides good scaling performance over a large community of clients.
- We develop and deploy a real testbed in Amazon EC2 and show that simply by dynamically adapting the resource allocation of the surrogate, EMCO increases gains in energy from offloading by 30 percent. In addition, we show that EMCO considerably reduces the energy required in the communication with cloud through the use of pre-caching.

1. Finnish for child, motivated by our runtime environment's parent-child structure.

2 Related Work

The origins of computational offloading are in cyberforaging, which aimed at distributing computations from a resource constrained (mobile) device to nearby infrastructure [21]. Since then, most of the work has focused on linking the device with remote infrastructure or a hybrid that combines remote and local connectivity. An example of the latter are Cloudlets [15] in which constrained devices act as thin clients that connect to remote infrastructure through a nearby gateway service (the cloudlet).

In most of the early works, offloading took place at application level, i.e., the execution of the entire app was shifted to another device. Current state-of-the-art solutions provide finer-grained control over the offloading process by instrumenting the apps at code level. For example, MAUI [2] proposes a strategy where the software developer annotates the code and during runtime methods are offloaded when suitable conditions are detected by the MAUI profiler. Similarly, CloneCloud [4] exploits a dynamic approach, where a code profiler extrapolates pieces of bytecode at thread level to a server. CloneCloud uses static analysis to partition code. However, this is inadequate to verify runtime properties of code, which usually cause unnecessary code offloading. COMET [5] is another framework for code offloading that offloads at thread level, but introduces a DSM mechanism that allows the device to offload unmodified multi-threads. ThinkAir [3] addresses scalability by creating virtual machines (VMs) of a complete smartphone system in the cloud. However, since the development of mobile application uses annotations, the developer must follow a brute-force approach to adapt his/her application to a specific device. Moreover, the scalability claimed by ThinkAir does not support multi-tenancy, but the system creates multiple virtual machines based on Android-x86 within the same server for parallelization. Other frameworks for computational offloading include [16], [22], [23], [24], [25]. COSMOS [6] provides offloading as a service at method level using Android-x86. The framework introduces an extra layer in an existing offloading architecture to solve the mismatch between how individual mobile devices demand computing resources and how cloud providers offer them. However, COSMOS relies on the user to manually solve this problem, which is inefficient. Moreover, end users seldom are familiar with different cloud options and their costs.

Studies of computational offloading in the wild have mostly shown that offloading *increases* computational effort rather than reduces it [26]. This is due to the large amount of mobile usage contexts, and the poor understanding of the conditions and configurations in which a device offloads to the cloud. Some recent work attempts to overcome this problem by dynamically alleviating the issues of inferring the right matching between mobile and cloud considering multiple levels of granularity [11], [27]. Similarly, CDroid [26] attempts to improve offloading in real scenarios. However, the framework focuses more on data offloading rather than computational offloading.

The EMCO framework proposed in the present paper differs from existing solutions by harnessing mobile crowdsensing for characterizing context parameters of mobile apps. By relying on samples from a community, EMCO reduces the effort and time of learning when to offload. This also improves the detection of offloading opportunities as it captures a large spectrum of possibilities from the

TABLE 1
Summary Statistics of Application Usage and Network Connectivity Datasets

Dataset	Carat	NetRadar
Data source	App store deployment (Multiple locations)	Finland (Metropolitan Helsinki area)
Number of samples	$2.9 * 10^9$ (from 328,000 apps)	135,000
Start time	01/08/2013	01/01/2015
Duration	1 year	1 year

community. Lastly, unlike other work that just uses the cloud as computational infrastructure to offload, EMCO takes advantage of the capabilities of cloud infrastructure to optimize the diagnosis of offloading contexts.

3 Feasibility of Crowdsensing for Quantifying Offloading Contexts

The focus of our work is on improving the effectiveness of offloading through the use of crowdsensing. Before detailing our approach, we investigate two fundamental questions in the use of crowdsensing. First, we examine the accuracy and generality of crowdsensed evidence traces and demonstrate that measurements provided by a community of devices can indeed be used to characterize offloading contexts. Second, we determine the amount of measurements required to characterize execution contexts. Specifically, we demonstrate that collecting a sufficient amount of measurements is infeasible on individual devices, but can be captured even from a (reasonably) small community of devices. We carry out our analysis using two large-scale datasets of mobile crowdsensing: Carat [19] and NetRadar [20]; see Table 1 for a summary of the datasets. The former captures the processing requirements and energy use of applications, whereas the latter captures network performance related parameters across different devices. Additionally, we carry out a crowdsensing simulation to characterize cloud resource allocation performance. In summary, we show that characterization via crowdsensing captures richer context information about smartphone usage (mobile context), network (communication context) and task acceleration (execution context).

3.1 Smartphone Application Usage

We first consider the feasibility of using crowdsourcing to quantify the computational performance on applications, a critical factor in deciding when to offload. We perform the analysis by considering a large-scale data set containing energy footprints of mobile apps collected through the Carat application [19]. While many offloading frameworks operate on finer level of code granularity, i.e., on a method or class level, collecting such data is infeasible as it requires a large-scale deployment of apps which have been instrumented with the appropriate mechanism. Nevertheless, we demonstrate that even when considering application level granularity, it is possible to identify the conditions in which mobile applications require intensive processing. The analyzed data contains data from approximately 328,000 apps, collected over a period of one year. For each app, the data contains the expected % of energy drain which is calculated over the entire community of devices; see [19] for details of the Carat methodology.

We first consider the minimum number of apps that are needed for identifying conditions for resource intensiveness. We analyze the distribution of expected energy drain, as given by the Carat application, and identify convergence of the resulting energy distribution. The results of our analysis are shown in Fig. 1a. From the results we can observe that measurements from around 120 different apps are required. The average energy drain of apps within the community is < 0.004, which depicts battery consumption in % per second. In our experiments we consider this value as a lower bound for the detection of resource-intensive behavior.

To further illustrate the practical value of our approach, we assess the fraction of apps that could benefit from offloading through two representative examples of application categories that are generally believed to benefit from offloading: photo face manipulation and games. We also separately consider puzzle and chess games, two subcategories of games with heavy computational requirements, to demonstrate that our methodology applies to smaller sets of applications. For each category, we extract all unique apps. Overall there are about 550 photo face manipulation apps, about 7,805 game apps, about 717 puzzle apps, and about 166 chess apps, i.e., each category meets the minimum requirement for estimating resource intensiveness. Each group is compared with the average energy drain of the subset, which excludes its own energy drain. From the results in Fig. 1 we can observe that around (1b) 43.84 percent, (1c) 44.56 percent, (1d) 42.75 percent and (1e) 33 percent of apps implement computational operations that require higher energy drain than normal, which is a significant number of apps. This result suggests that computational offloading could be beneficial for a large fraction of apps.

As the final step of our analysis, we consider the amount of samples required from the community to accurately characterize high energy drain. According to Fig. 1f, we find that about 10,000 samples are gathered to estimate high energy drain within interval of 0.0040 ± 0.000035. As Carat collects samples each time the battery of a device changes, we get on average $50 - 100$ samples per device in a day. Collecting the required 10,000 samples on a single device would thus clearly be infeasible, but a community of 100 devices could achieve this in a day or two.

A limitation of our analysis is that we considered app level offloading instead of method, class or thread level migration. Intuitively, we would expect finer-granularity to further improve decision making. However, this requires instrumenting the offloading mechanisms with the ability to monitor its own local and remote execution. To enable this kind of analytics, our implementation of EMCO integrates required mechanisms to capture both local and remote execution performance; see Section 4 for details.

3.2 Network Connectivity

Communication latency during application usage is another key building block for making optimal offloading decisions. Latency is typically monitored through a network profiler, which is also responsible for characterizing different communication interfaces, e.g., WiFi, 3G, LTE. Characterizing networking performance, however, can easily become energy-intensive, especially in suboptimal network conditions as more samples and and a longer duration of monitoring are required. In this section we examine the potential of replacing the network profiler with crowdsensing. The main question we investigate is how many samples are

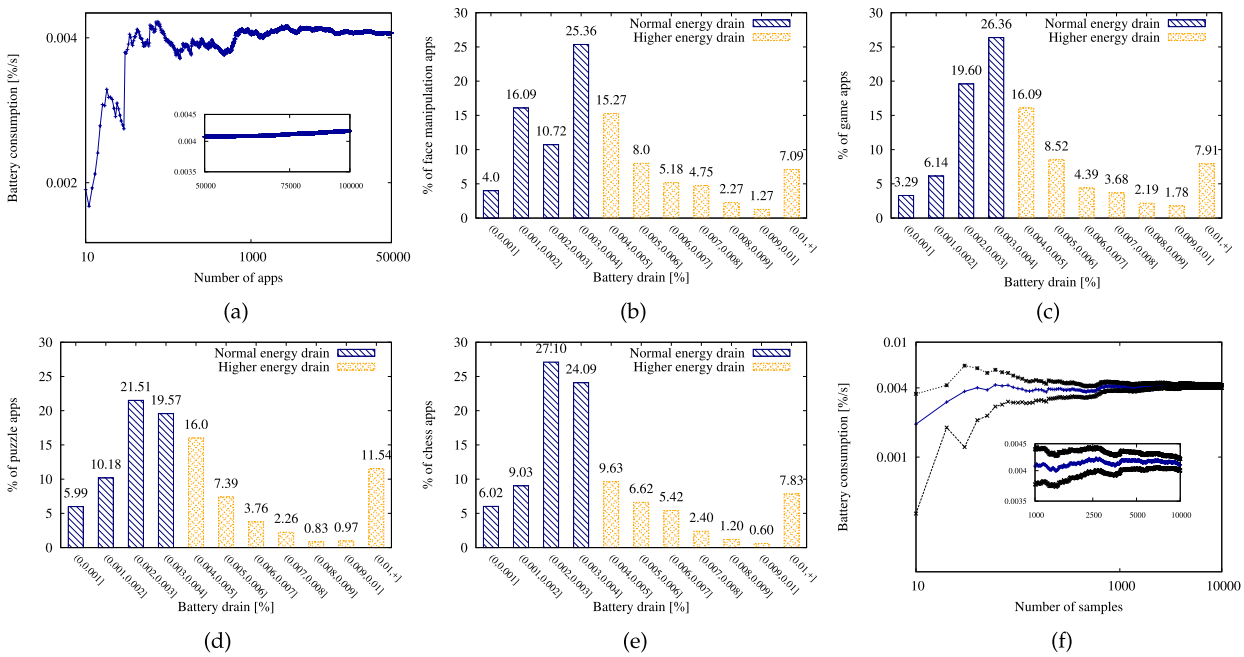


Fig. 1. Characterization of smartphone app usage via crowdsensing. (a) Number of applications for community characterization. (b), (c), (d), and (e) Smartphone apps that depict higher energy drain at different community granularity. (f) Number of samples for energy characterization.

required to characterize latency accurately at a particular time or location. We answer this question by analyzing the NetRadar dataset[2] [20] and characterizing the performance of 3G and LTE communications. Since latency in the cellular network also depends on the provider, we consider three different mobile operators, which are the main network providers in Finland. For anonymity, we refer to the operators using pseudonyms Operator-1, Operator-2 and Operator-3.

The topmost row of Fig. 2 shows the average latency of the communication (RTT) for each provider. From the results, we can observe that the average RTT varies across the operators: 107 ms (SD≈283) for Operator-1, 130 ms (SD≈387) for Operator-2, and 129 ms (SD≈403) for Operator-3. For LTE, the averages are 43 ms (SD≈35) for Operator-1, 33 ms (SD≈65) for Operator-2, and 39 ms for (SD≈90) Operator-3. While there is a notable difference between 3G and LTE in terms of speed, we can observe higher stability in the communication link of LTE. In terms of quality of service, in the case of 3G, Operator-1 provides the most stable communication and Operator-3 the most unstable one. In the case of LTE, Operator-2 provides the most stable communication and Operator-1 the most unstable one, i.e., both the operator and the communication technology should be taken into account when making offloading decisions.

Regarding the amount of data needed for characterization, we calculate the number of samples required to estimate latency. The average and median latency as a function of the number of samples for 3G and LTE are shown on the two bottommost rows of Fig. 2. We can observe that the amount of data required to characterize the latency of communication depends on the stability of the communication link. For instance, for 3G, to characterize the most stable communication (Operator-1) with an interval of 100 ± 7.24 at least 6,300 samples are needed. In contrast, if the same amount of samples is used to characterize the most unstable channel (Operator-3), then an interval of 132 ± 10.06 is achieved. Thus, to improve accuracy and reduce error at comparable rates as Operator-1, an addition of 3,700 samples is required (10,000 in total). By generalizing the characterization time as a function of samples required, we have $T = S/(S_t * n)$, where T is the time that takes to complete the characterization, S is the total number of samples required for the characterization, S_t is the number of samples collected in time t, and n is the number of devices available to collect samples. In terms of sampling complexity, assume we want to characterize latency of 3G for Operator-3 and that we receive 5 samples per day from a device. A single device would then need about 2,000 days to complete the task. For a single device to gather the data in one year, it would have to collect 370 samples per day, significantly reducing battery life. Even if this was done, we would only be sampling latency at locations the device visited. In contrast, 100 devices (easily found at University Campus) can complete the characterization in about three weeks. By piggybacking the sample collection as part of application data requests, significantly larger communities of devices could be reached, making it possible to complete the characterization in a matter of hours.

3.3 Augmented Resource Allocation

Computational provisioning takes place at infrastructure level through instances, which are physical or virtualized servers associated to a specific type of resource [27]. An instance follows a utility computing model, whereby consumers pay on the basis of their usage and type preferences. The cost of the instance is proportional to its type, which in turn determines the level of acceleration at which a task is executed and energy required in the communication [28]. Moreover, the type of the instance also defines its capacity to handle multiple offloading requests at once. Speeding up code execution certainly depends on how 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 [29].

To thoroughly characterize cloud resources, we conduct a simulation study where a large number of clients create a

2. https://www.netradar.org/en

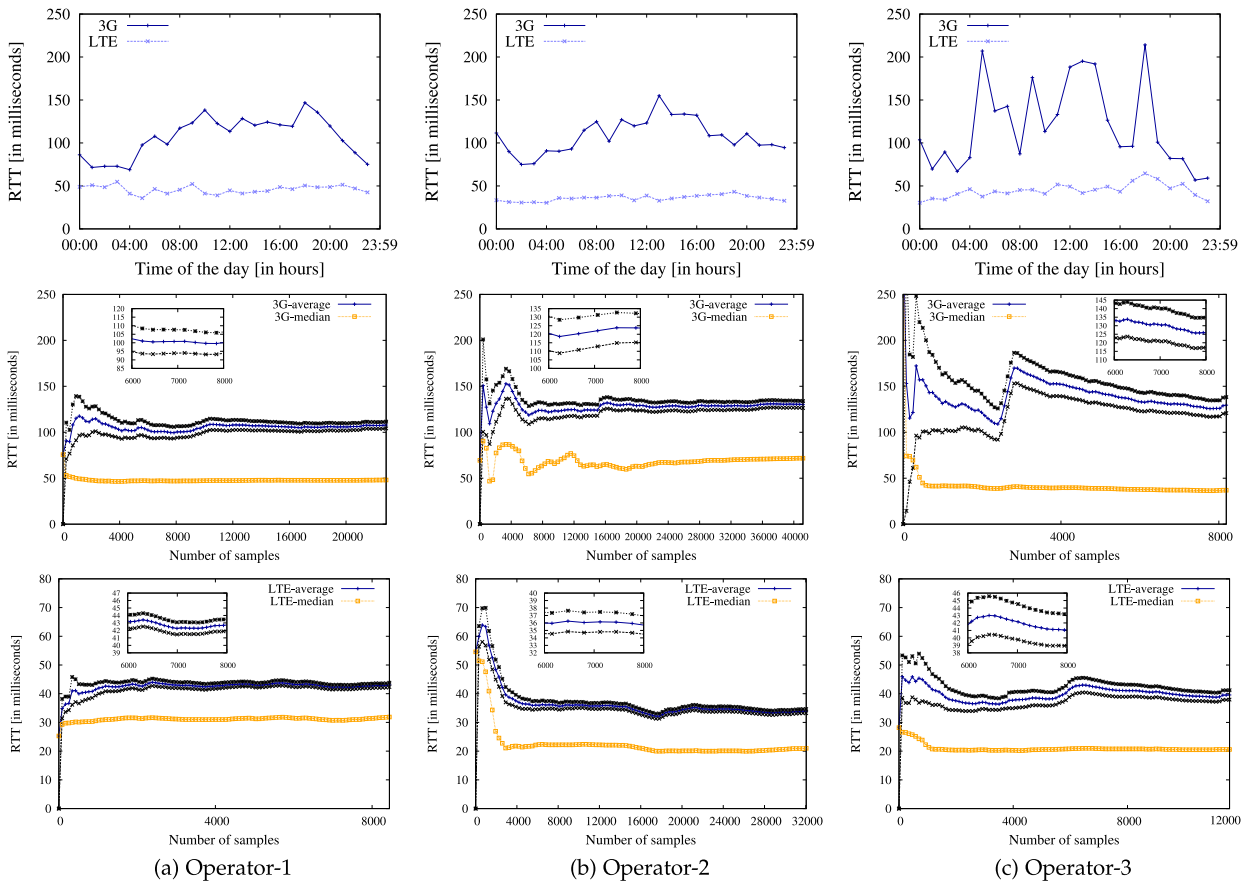


Fig. 2. Network characterization: (top and middle-top) Number of samples collected at different times of day. (middle-bottom) Latency for 3G and LTE technologies (Operators 1-3). (bottom) Amount of samples required to characterize latency.

dynamic workload which is sent to the cloud [30]. Our simulator operates in two different operational modes: (i) concurrent and (ii) inter-arrival rate. In the former, 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 outsourcing a task to a remote server. This mode is utilized to benchmark the instances of the cloud. In 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 synthetic workloads. We note that while throughput could be analyzed through server logs, such logs are (to our best knowledge) not freely available and we are not aware of any studies that would have done so. Thus, we focus instead on performing the characterization on the clients with the help of a community of devices.

To simulate crowdsensing in the wild, we configure our simulator in concurrent mode to stress the instances with a heavy workload 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-hour experiment per server to ensure coverage. We also check 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; per each server loads 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 enough cool down time to the server before stressing it again.

Fig. 3 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 when server utilization is low. However, there is also a correlation between server acceleration and utilization, the response time of an offloading request degrades based on the resource capabilities of the server as utilization increases.

Lastly, we used a minimax algorithm with a static input as workload to confirm our results. We can observe the differences between acceleration levels. A task is executed $\approx$1.25 times faster by a server of level 2 when compared with one of level 1. Similarly, a task is accelerated $\approx$1.73 times by a server of level 3 compared with level 1. The difference between levels 3 and 2 is also significant ($\approx$1.36 times acceleration). Our results thus demonstrate that crowdsensing is capable

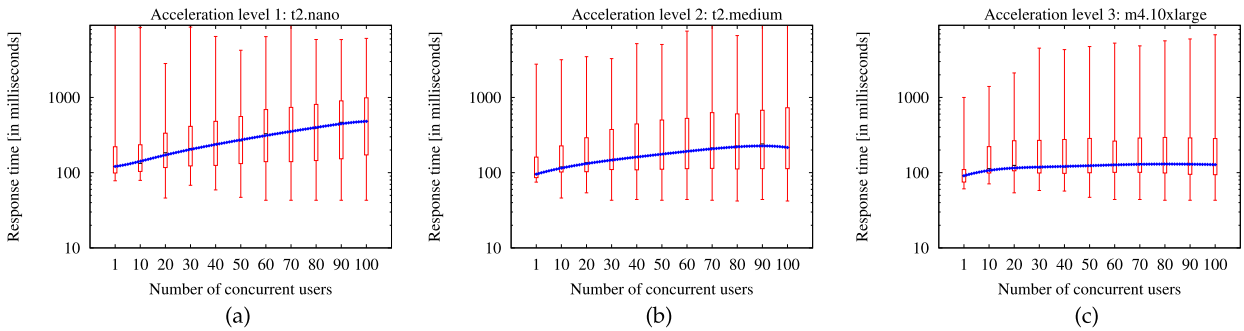


Fig. 3. Cloud-based characterization via crowdsensing gives different acceleration levels to servers based on performance degradation as more users are added.

of characterizing both cloud throughput and acceleration across a wide range of contexts.

4 EMCO Toolkit and Architecture

In this section we describe the EMCO development toolkit and platform, which integrate novel mechanisms for harnessing crowdsensed evidence traces in offloading decision making. EMCO handles two types of mobile communication with the cloud: synchronous code offloading and asynchronous injection of data analytics via push notifications. The overall architecture of EMCO is shown in Fig. 4. EMCO follows a model where the source code of an app resides both on the mobile and at the server, but it releases the mobile from a fixed cloud counterpart as implemented by other frameworks. EMCO encourages a scalable *provisioning as a service* approach where EMCO is deployed on multiple interconnected servers and responds on-demand to computational requests from any available server. EMCO also provides a toolkit that facilitates the integration of the mobile and cloud mechanisms in the development life-cycle. In the following we describe the different mechanisms and components of EMCO in detail.

4.1 Development Toolkit

To integrate offloading support into applications, EMCO provides a toolkit that consists of a GUI conversion tool that prepares the apps for client/server deployment. Our conversion tool receives as input an Android project and produces as output two projects, one for the mobile, and another for the cloud. Each project is defined with Maven-Android[3] so that for each project an Android Application Package (APK) file is built automatically. The tool also provides means for automatic deployment of the APK to the cloud. Our conversion tool has been designed to operate with existing frameworks and hence does not introduce any additional complexity, such as learning a new API.

The conversion tool transforms all methods that fulfill offloading requirements by wrapping each task (or method) into a container, which retrieves its execution details from a *descriptor* during runtime (see Section 4.3) [4]. The container is named with the name of the task, while the original task is renamed by adding the prefix "local". Similar instrumentation is followed in [3], [6], [8] with the main difference in our approach being the inclusion of a descriptor that governs the execution of tasks based on the community analysis. Besides equipping the app with offloading mechanisms, the conversion tool integrates the device with crowdsensing mechanisms, i.e., mechanisms for collecting context information and sharing it with the wider community. As has been shown in previous works [19], such monitoring has minimal impact on the battery lifetime of the mobile client.

4.2 EMCO Server

The *EMCO Server* provides computational resources for offloading and analyzes crowdsensed measurements to identify contexts where offloading is beneficial. We next detail the server's subcomponents and their key functionalities.

4.2.1 Evidence Analyzer

The Evidence Analyzer analyzes evidence traces collected through crowdsensing and identifies conditions that are beneficial for offloading. Our implementation of the Evidence Analyzer is based on a dual layer analysis pipeline. The first layer, dimension discovery, identifies factors that are most relevant for creating a *code offload descriptor* for a particular app. The second layer, classification, uses the evidence obtained by the first layer to build a classifier which integrates the decision logic used to construct the offload descriptor. The constructed classifier can then be deployed on the client to make offloading decisions directly without interfacing with the cloud.

The first layer, dimension discovery, operates on traces collected from the community. A trace consists of a list of parameters or features such as device model, app identifier, the name of the executed method, input of the method, local execution time, RTT, type of server, and remote execution time. Additionally, when the code is offloaded, each trace includes a binary variable that indicates whether the remote invocation succeeded. The invocation can fail, e.g., if the surrogate in unable to invoke the code (e.g., missing dependencies) or communicate back the result of that invocation (e.g., malformed serialization). To analyze evidence, we transform the evidence into *rate distributions* which model

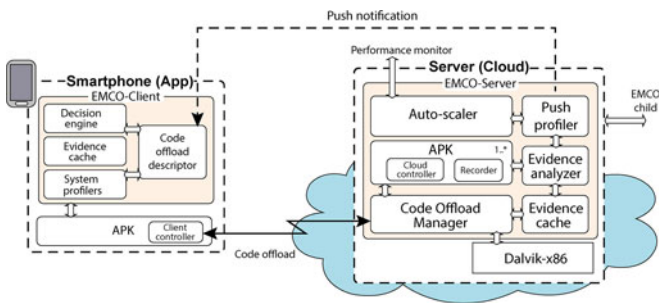


Fig. 4. Evidence-aware mobile code offloading architecture.

3. http://code.google.com/p/maven-android-plugin/

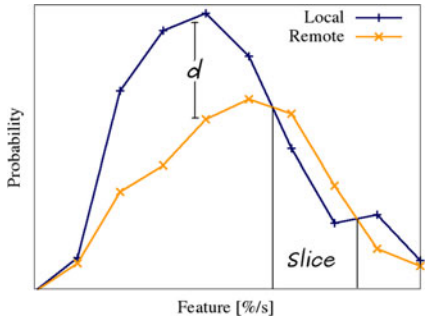


Fig. 5. Comparison of local and remote distributions.

variations of a specific feature f over time. For each parameter, we construct separate rate distributions for local and remote execution and compare them to identify when offloading is beneficial. Rate distributions are created for all obtainable parameters with the only requirement being that the feature and its value can be estimated during the execution of local and remote computations.

Formally, we construct rate distributions by converting evidence traces into a set of samples. Let $s_t = (c, p, \hat{f})$ denote a sample taken at time t of code execution with identifier c (e.g., app id or method name) which is processed at p (local or remote) and which is characterized by the set of features $\hat{f}$. In our implementation we consider the features to be represented as key-value pairs, e.g., "response-time=2s", "energy-consumed=180J". We sort the samples based on time and split them into two groups, one for local execution and one for remote. For each group, successive measurements (s_{t_1}, fs) and (s_{t_2}, fs), are converted into a rate distribution $u = \frac{(fs_2 - fs_1)}{(t2 - t1)}$, which later is associated with the rest of features to create a pair $R = (u, \hat{f})$. This association is important in order to keep track about the values of the other features with fs. Once the rate distributions are calculated for a given feature, pairs R_{local} and R_{remote} are compared as shown in Fig. 5. The comparison consists of finding an overlap, or a *slice*, between the two distributions. Typically, R_{remote} values that are higher than R_{local} ones suggest that remote execution is counterproductive for the device in the context specified by the slice.

When slices that are counterproductive for offloading are encountered, we remove the corresponding samples from analysis and recalculate the rate distributions. We then analyze the next feature fs_i over the remaining set of samples, i.e., we gradually refine the samples and features to identify conditions that are favorable for offloading. Once the refinement process completes, we construct a *code offload descriptor* which incorporates the most important dimensions. To create the descriptor, we compare the distance d between the local and remote distributions as this determines the level of improvement; see Fig. 5. We aggregate the distances d into bins, and select the bin with the highest count for the descriptor. Note that the process thus identifies the most frequent conditions for obtaining benefits instead of selecting the ones with the highest performance increase. This is essential for avoiding overfitting to highly specific conditions which are encountered infrequently. EMCO also extracts various alternatives for surrogates based on the levels of frequency of d. Additionally, the set of rules is created at the end of the descriptor following a simple heuristic method presented in [31]. Since processing in the cloud is not constrained to service response time, we choose the heuristic solution. Naturally, any other method can be implemented and the modularity of EMCO provides an easy way to extend the functionality of the components.

Once the first layer has finished refining the evidence, the remaining samples are passed on the second layer, classification, to train a classifier which can be deployed on the mobile device as a decision engine. As classifiers we currently use C4.5 decision trees as they can be learned efficiently in the cloud from crowdsourced data and as the decisions made by the classifier can be easily explained [32]. An additional benefit from the use of decision trees is that it provides a mechanism for app developers to scrutinize which parts of their app have been offloaded and why.

The overall EMCO analysis is described in Algorithm 1 and consists of $\approx$ 1500 LOC written in Scala. To avoid overloading the surrogate with extra processing, the analysis is outsourced to a separate monitoring server, which also collects performance metrics about EMCO execution.

4.2.2 Evidence Cache

To facilitate bootstrapping of offloading decisions, the server maintains a cache of previous method invocations together with results of the invocations. We take advantage of the fact that each offload request (including method name, input parameters, and other information relevant for an invocation) received by EMCO server is serializable and calculate a digest key from the output of the serialization (SHA-1 checksum). We then index the resulting invocation results with the checksums. Whenever a new request is received, the server compares the checksum of the new request against the previously seen requests and returns the results from the cache whenever a match is found.

Algorithm 1. Evidence Analysis

require: path to the *traces*
for each *app* in *traces* **do**
  Create samples.
  Specify features to compare $< fs >$
  for each *feature* in fs **do**
    Compute rate distribution using samples.
    $< R_{local}, R_{remote} >$
    Compare R_{local} with R_{remote}
    Remove slices where R_{remote} is higher
    Update samples
  end for
  Calculate d using filtered samples.
  Create *code offload descriptor*
  EMCO policy creates the descriptor based on higher frequency of d
  Push descriptor to subscribed smartphone app
end for

4.2.3 Code Offload Manager

The *Code Offload Manager* is responsible for handling code offload requests arriving at the server. The component extracts from each incoming request an *object package* which contains details about the app and details of invocation (method, type of parameters, value of parameters, and so forth). As outlined above, we first calculate a checksum of the request and compare it against the evidence cache. When a match is found, we return the result associated with

the match. Otherwise the offload manager identifies the app and its associated (Dalvik) process in the server. The connection is forwarded to the process so that the *Cloud Controller* associated with the APK file instance can invoke the method via Java reflection. The result obtained is packed and sent back to the mobile device. In parallel, a trace is created by recording the execution of the method, e.g., execution time, type of server, CPU load, etc., and stored along with the data of the request and its result. Finally, the trace is passed to the *Evidence Analyzer* for analysis.

4.2.4 Push Profiler

The *Push Profiler* is the messaging component of the cloud server. Our implementation is based on GCM (Google Cloud Messaging) service[4] which is a notification service provided by Google to send messages to Android devices. We rely on an existing technique as messaging mechanisms are widely used by cloud vendors to trigger events on the mobiles to synchronize data with their cloud counterparts, e.g., Gmail. These mechanisms are well integrated with current handsets and they have been carefully optimized to have low energy consumption [33].

The EMCO server is registered to GCM service using a key obtained from the Google API. Multiple servers associated with one deployment of EMCO can use the same key. A mobile app subscribes to EMCO to receive notifications, obtaining a sender ID as a result. The ID is valid until the app unsubscribes (or GCM service is refreshed). EMCO then sends a message to the smartphone client which consists of a push request to the GCM service. The request consists of the *API key*, the *sender ID* and the message payload. Requests are queued for delivery (with maximum of five attempts) or stored if the mobile is offline. Once the message reaches the device, the Android system executes a *broadcast intent* for passing the raw data to the *code offload descriptor*.

EMCO relies on push requests to construct code offload descriptors for smartphone apps. As each message sent through GCM is limited to 1024 bytes, multiple messages are needed to form a descriptor. Messages are sent to the mobile based on different events, for instance, dynamic cloud allocation, periodic trace analysis, location detection of the mobile, etc. Since GCM is a popular service, it does not guarantee the delivery of a message and is unreliable to be used in real-time applications. To counter this problem, EMCO provides a generic interface, which can be used to integrate easily other push technologies, e.g., MQTT (Message Queue Telemetry Transport). We have also developed our own push server[5] based on XMPP, which can overcome the issues of proprietary push technologies and performance reliability of the notifications.

4.2.5 Auto Scaler

The *Auto Scaler* enables EMCO to support multi-tenancy. It uses the *AMI EC2 tools* wrapped into Java, to manage dynamic resource allocation of servers. The mechanism allows the framework to scale dynamically in a public or private cloud infrastructure. Note that while public cloud vendors provide mechanisms for scaling, e.g., Amazon Autoscale and Amazon Lambda, these mechanisms are designed for scaling SOA applications on demand. Resource requirements in computational offloading are driven by users' perception of app performance, which is governed by throughput of the cloud infrastructure. This differs from traditional SOA applications that focus on energy savings and reduced computational time without concern for throughput variations. EMCO addresses this gap by utilizing a hierarchical control and supervision scheme as shown in Fig. 6. In our scheme, EMCO server acts as a parent, and a separate monitor server collects performance metrics from the available EMCO servers (based on collectD[6]). When the parent reaches its service utilization limits, it creates another child. In this case, the list of subscribed devices is split between the parent and child, and a push notification message is sent to each device to update the information of its assigned surrogate. In this manner, code offloading requests are balanced among the servers. Respectively, a child can become a second level parent when its service utilization limits are reached. When a child is destroyed, e.g., when it has served all requests and there are no incoming requests, the subscribed mobile devices are passed to the parent.

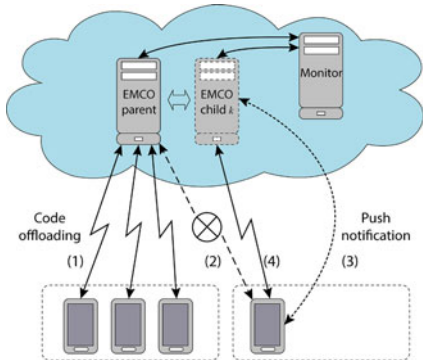


Fig. 6. Multi-tenancy and horizontal scaling in EMCO.

4.2.6 Surrogate Runtime Environment

EMCO Server is deployed on top of Lightweight App Performance Surrogate Image (LAPSI), a customized Dalvik-x86[7] virtual machine for Android. LAPSI has been built by downloading and compiling the source code of Android Open Source Project (AOSP) over the instance to target a x86 server architecture, and removing the Applications and Application Framework layers from the Android software architecture. Contrary to other frameworks [3], [4], EMCO does not rely on the virtualization of the entire mobile platform to execute the code remotely (such as with Android-x86). Full virtualization has been shown to consume substantial CPU resources and to slow down performance of code execution [34]. Moreover, full virtualization limits the number of active users offloading to a single server. In our case we also found that maintaining a complete execution environment requires substantial storage on the server (>100 GB) to install the software which includes AOSP code and Android SDK, among others. Moreover, once the OS is running in the server, all default features are activated (e.g., Zygote and GUI Manager) which are unnecessary for operating as surrogate and which result in higher CPU utilization of the surrogate. To avoid these overheads, LAPSI uses a Dalvik compiler extracted from the mobile platform which is deployed directly onto the underlying physical resources. This reduces the storage size

4. developer.android.com/google/gcm/
5. https://github.com/huberflores/XMPPNotificationServer
6. http://collectd.org/
7. Released as public in Ireland region of Amazon EC2 as *ami-b2487bd4*

of our surrogate to 60 GB. Moreover, our surrogate does not activate any default processes from the OS. Besides reducing overall system load, a major advantage of our approach is that offload requests getting stuck or lost can be killed individually without restarting the entire offloading system.

4.3 EMCO Smartphone Client

The *EMCO Client* manages the offloading process on mobile devices. We next detail the subcomponents of the EMCO client and their key functionalities.

4.3.1 Context Profilers

Profilers are local monitors that gather information about parameters that can potentially influence the outcome of the offloading process. Examples of profilers include system profilers (e.g., network and CPU monitors usage), device monitors (e.g., battery state), and sensor data profilers (e.g., location or remaining battery). We rely on built-in Android APIs to get the status of each resource. In addition, EMCO also includes a code profiler that quantifies the execution of a task (i.e., a method or a process). A task is quantified in terms of response time and energy. The latter is estimated using PowerTutor [35], which has been shown to have low estimation error [19]. We also collect details about the task, e.g., cast type and parameters which allows more fine-grained analytics of offloading performance.

4.3.2 Code Offload Descriptor

The code offload descriptor controls the management (creation, updating, and deletion) of the data analytics injected from the cloud for a particular app running on the device. The descriptor consists of a characterization of the computational offloading process in JSON format which is stored as part of the app space; see Snippet 1 for an example. The descriptor effectively characterizes the logic that is used to determine which parts of the app are suitable of offloading, which acceleration levels are suitable for executing these parts, and which contexts are beneficial for offloading.

Formally, let D represent the dimensions contained in the descriptor, $D = \{d_i \in D : 1 \leq i \leq N\}$. Each $d_i \in D$ consists of sets S of attributes s^{d_i}, where $S^{d_i} = \{s_j^{d_i} \in S^{d_i} : 0 \leq j \leq N\}$, and their values a^{d_i}, where $A^{d_i} = \{a_k^{d_i} \in A : 1 \leq k \leq N\}$. Our current implementation assumes attributes are represented as key-value pairs to ensure their storage and processing is lightweight. The descriptor can also contain other types of parameters, such as an indicator of whether the results of a specific method request are reusable and thus a candidate for caching, or parameters that describe user preferences regarding which surrogate to use.

4.3.3 Decision Engine

The decision engine predicts whether it will be beneficial to offload code at any particular moment. The engine relies on the offloading dimensions defined in the code offload descriptor and a C4.5 decision tree classifier learned on the EMCO server. We emphasize that EMCO is agnostic to the underlying decision engine and other techniques, including fuzzy logic, probabilistic models, logic-based reasoners and so forth, can equally well be integrated as with EMCO. The motivation for using C4.5 is that it can be effectively learned on the cloud, the output can be serialized and communicated back to the client with minimal data costs, the decision making mechanism is lightweight to run on the client, and the decision paths can be easily scrutinized and communicated to developers whenever needed [32].

4.3.4 Evidence Cache

Similarly to the server, the EMCO client utilizes a cache to maintain results of methods that are executed in the cloud. We store the results of method invocations into a hash table as key-value pairs, where the key is the name of the app appended with the name of the method, and the value contains the object sent from the cloud, the type in which the object should be cast during runtime and a time to live (TTL) value. The TTL value prevents to store permanently a result in the mobile. Thus, the result is assigned with a temporal lifespan. In this context, if the result is not used during that lifespan, then it is automatically removed from the hash table. The TTL value is updated every time the cache result is utilized in order to increase its lifespan.

Snippet 1. Computational Offloading Descriptor in JSON

```
{
"mobileApplication": "Chess",
"deviceID": "i9300",
"what-to-offload": [
  {
  "candidates-methods": "miniMax, betaPruning",
  %META-DATA
  }
],
"when-to-offload": [
  {
  "bandwidth-threshold": "50", %META-DATA
  "interface": "LTE", %META-DATA
  }
],
"where-to-offload": [
  {
  "option1-surrogates": "{(miniMax, m1.large),
     (betaPrunning, m2.4xlarge)}", %META-DATA
  "option2-surrogates" : "{(miniMax, m1.micro)}",
  %META-DATA
  }
],
"location": [
  {
  "GPS-coordinates":   "60.20453,   24.96185",
  %META-DATA
  "Preferable SSID" : "HUPnet", %META-DATA
  }
]
}
```

4.4 Worked Examples

To illustrate EMCO in action, we consider two example scenarios, one related to a chess app that allows playing against an AI agent and the other related to a backtracking-based 3D model loading app; see Section 5.1 for description of the apps. We consider context descriptors associated with these apps shown in Snippets 2 and 3, respectively. The values of the context variables in the examples correspond to the average values of the parameters in the crowdsensing experiments described in Section 3 and are representative of the real-world circumstances where offloading takes place.

For the context associated with the chess app (Snippet 2), WiFi and LTE have similar latencies. Hence, EMCO's decision of whether to offload (and if so, where to offload) largely depends on the computational capability of the surrogate and the mobile device. In Snippet 3, on the other hand, WiFi has high latency (147 ms) which would result in LTE being used as the communication technology (latency 39 ms). In both scenarios the surrogate used for offloading can be chosen based on estimated response time to minimize latency. Note that since both scenarios represent computationally intensive tasks, the effect of network bandwidth is not apparent. For data-intensive, real-time tasks, such as face manipulation on a live camera stream, the combined effect of bandwidth, latency, energy, and remote surrogate response time need to be taken into account. We have included the type of the device and its current CPU usage as part of the examples as both the savings from offloading and the energy footprint of the method depend on device characteristics and CPU load.

Snippet 2. Chess Experimental Context.

```
label=Ctx1, device=i9300, CPU=20%
LTE=40 ms, 3G=139 ms, WiFi=27 ms
method=Minimax, parameters="chessboard-15",
energy=533J,
surrogates=["m1.small" (response time (load=1)),
  "m1.medium" (load=1), "m1.large" (load=1),
  "m3.medium" (load=3), "m3.large" (load=2),
  "m3.xlarge" (load=6), "m3.2xlarge" (load=10),
  "m2.4xlarge" (load=8)]
```

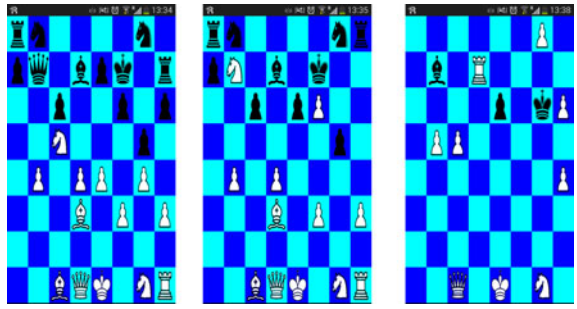
Snippet 3. Backtracking Experimental Context.

```
label=Ctx0, device=i9250, CPU=43%
LTE=39 ms, 3G=138 ms, WiFi=147 ms
method=ObjLoader, parameters="model.3ds",
energy=212J,
surrogates=["m1.small" (load=1),
  "m1.medium" (load=1), "m1.large" (load=11),
  "m3.medium" (load=30), "m3.large" (load=10),
  "m3.xlarge" (load=22), "m3.2xlarge" (load=70),
  "m2.4xlarge" (load=50)]
```

Snippet 4. Two Alternative Chess Contexts.

```
label=Ctx2, device=i9300, CPU=80%
LTE=73 ms, 3G=365 ms, WiFi=80 ms
method=Minimax, parameters="chessboard-30",
energy=97J
label=Ctx3, device=i9300, CPU=40%
LTE=65 ms, 3G=411 ms, WiFi=287 ms
method=Minimax, parameters="chessboard-19",
energy=528J
```

Another benefit of EMCO is that it can react to dynamic changes in the offloading context. For example, the offloading decision can be adapted based on the location where the app is being used. To illustrate this, we consider two additional contexts shown in Snippet 4. The latencies in these contexts correspond to average latencies within two distinct cities (Ctx2 and Ctx3) and have been derived from of our study in Section 3.2. We also consider different stages of the game, chessboard-19 and chessboard-30, where 19 and 30 correspond to the number of moves made. The former corresponds to an intermediary state which has many possible moves whereas the latter corresponds to a late stage of the game with fewer possible moves and hence lower computational requirements; see Fig. 7. We next analyze how EMCO behaves when the configuration of the chess application changes from Ctx1, to Ctx2 and Ctx3.



(a) chessboard-15 (b) chessboard-19 (c) chessboard-30

Fig. 7. Three execution states for the chess app: (a) and (b) intermediate stage (15th and 19th moves) that are computationally intensive, and (c) a later stage of the game (30th move) which is computationally lightweight to solve.

Assume first that the context of the chess application changes to Ctx2. This triggers an automated update of the values of the network interfaces using information from the community of devices running EMCO. We assume that the status of the cloud infrastructure remains the same. In Ctx2, LTE performs better than WiFi whereas in Ctx1 WiFi performed better. Consequently, EMCO would trigger a change of network interface to LTE. The example in Ctx2 corresponds to a later stage of the game (See Fig. 7b) and thus the computational requirements are likely to be lower than in Ctx1. While in theory the request could be served on the local device, the high CPU load on the client in the example context would trigger EMCO to offload the request. As the computational requirements are likely to be low, any available surrogate can be chosen without significant influence on performance. Alternatively, the diagnosis generated by EMCO can be useful for optimizing resource allocation [36] to select the server with the cheapest cost that provides the same quality of service. Consider next a change from Ctx2 to Ctx3, the second in Snippet 4. The context corresponds to an intermediate state of the game and hence computing requirements are higher than in Ctx2. This would result in EMCO optimizing surrogate selection, e.g., to use a m2.4xlarge surrogate to optimize response time. In summary, these examples highlight EMCO's capability to make fine-grained adjustments depending on the currently prevailing offloading context and determine not only when to offload, but to which surrogate to offload.

5 Evaluation

EMCO has been designed to improve the user experience of mobile apps through better offload decision making. This results in reduced energy footprint and improved responsiveness. This section evaluates EMCO, demonstrating that it achieves its design goals and improves on current state-of-the-art solutions. Our main experiments have been carried out through a a real-world testbed deployment of EMCO.

We also consider simulation-based experiments carried out using crowdsensing and offline profiling of applications.

5.1 Experimental Setup and Testbed

Our main experiment considers a typical scenario where a mobile app is associated with cloud infrastructure to improve energy and performance on the users' devices. We consider two simple examples of such apps: a backtracking-based 3D model loader app and a chess app with an AI agent. These apps were designed to incorporate common computational operations that are used in apps with heavy computational requirements and battery footprint. Our implementations are freely available for research purposes.

Backtracking[8] overlays a 3D model on the surface of the mobile camera. A user can interact with a model once is displayed, e.g., moving and turning. Our implementation is equipped with ten models (in 3ds format) with the size of each model being between 500 KB and 2000 KB. Loading and displaying the mesh of each model requires a significant amount of processing and hence each model is loaded individually as required by the user. The heavy processing requirements of model loading make the app a good candidate for offloading. Loading the model in the cloud enables splitting it into parts so that the only parts the user interacts with are sent to the mobile device. Examples of similar apps include Augment http://www.augment.com/augmented-reality-apps/and Glovius 3D CAD viewer https://www.glovius.com/.

Chess[9] allows user to play (chess) against an AI agent on multiple levels of difficulty. The agent uses a minimax algorithm to determine the best move to make. The portion of code that corresponds to the minimax logic is the most intensive task in the application and hence a good candidate for offloading. Note, that the processing requirements of the minimax vary considerably depending on the state of the chess board with early and late moves typically being less intensive than intermediate ones; see Fig. 7.

To conduct our experiments, we have deployed EMCO in Amazon (Ireland region / eu-west-1) using latest general purpose instances[10] (m3.medium, m3.large, m3.xlarge, m3.2xlarge) and a memory-optimized instance (m2.4xlarge). As client devices we consider two smartphone models: i9300 (Samsung Galaxy S3) and i9250 (Google Nexus). We follow the principle that an offloading request should both improve (or at least preserve) app responsiveness while reducing energy consumption of the client [29]. The appropriate level of responsiveness depends on the type of application. For example, real-time games or video applications would expect to have a speed of 30-60 frames per second, while interactive non-realtime applications can have response times up to one second. The energy footprint of the client is measured using a Monsoon Power Monitor.[11]

5.2 Characterizing Offloading Context

As the first step of our evaluation we consider a benchmark where we collect evidence for characterizing the execution of our test apps. We execute both apps 48 times on each of the five surrogates (i.e., 240 remote execution trials) and 100 times on each mobile device (i.e., 200 local execution trials). For the chess app, the number of offloading requests depends on the length of the chess match. In total we observed 7,200 requests, i.e., approximately 30 moves per game. For the backtracking app we loaded all 10 models during each trial, resulting in 2,400 offloading requests (48 trials $\times$ 5 surrogates $\times$ 10 models). The surrogate used for offloading requests was selected in a round-robin fashion from the instances which were available.

The results of our benchmarks are shown in Fig. 8. For both apps we observe that offloading generally improves both response time and energy footprint. However, we can also observe that the magnitude of the benefits depends on the execution context. This is particularly true for the chess app where we can see a large variation in response time depending on the complexity of the game state (Fig. 7). For later stages of the game the benefits of offloading start to be marginal compared to local execution - unless the request is found in the pre-cache which results in latency becoming the dominant factor for offloading decisions.

5.3 Response Time and Energy Footprint

We next evaluate the gains in response time and energy obtained by using EMCO. As part of the experiment we also assess the performance gains provided by caching results of offloading requests and reusing them. We perform the evaluation by executing the apps locally on the device and offloading them into all available surrogates and measuring the associated response time and energy consumption.

Figs. 8c and. 8f show the energy consumed in each offloading case for each application, respectively. From the results we can observe that the effort to offload is different depending on the time of active communication. From the figure we can also observe caching to further reduce energy consumption and response time.

Existing offloading methods bind mobile devices to any available server in the deployment and it remains static server during application usage. In contrast, EMCO implements a mechanism that allows requests to be treated independently, which means that can be handled by multiple servers based on its processing requirements. To compare EMCO against current state-of-the-art, we have developed a generic code offloading framework which (i) makes offloading decisions based on network latency and estimated energy, but (ii) has no support for diagnosing cloud performance and allocates requests to a random server instead (in the experiments these correspond to m3.medium for the chess app and m3.2xlarge for the backtracking app). Our baseline does not directly follow any single system, but incorporates mechanisms commonly found in current offloading frameworks.[12] [2], [3], [6], [8]

The response times of EMCO and the baseline are shown in Fig. 9a. For the chess app (top figure), we can observe that EMCO is capable of adjusting the offloading environment according to characteristics of the computing task and current context, which results in a stable and fast response time throughout. The performance of the baseline suffers during complex intermediate states as the system cannot adapt to the changing computational requirements. Furthermore, the

8. https://github.com/huberflores/MeshOffloading
9. https://github.com/huberflores/CodeOffloadingChess
10. We have also considered general purpose instances of previous generation: m1.small, m1.medium, and m1.large. These instances consistently resulted in worse performance than local execution on the mobile client and hence we have excluded then from our discussion.
11. http://www.msoon.com/LabEquipment/PowerMonitor/
12. https://github.com/huberflores/CodeOffloadingAnnotations

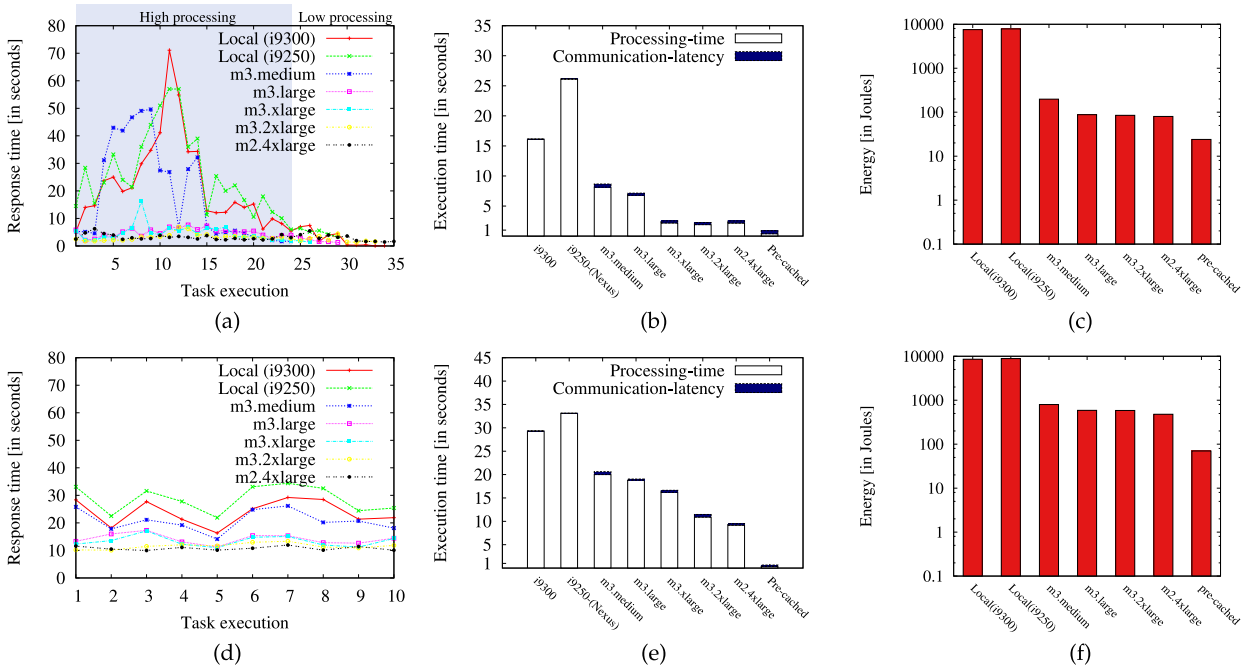


Fig. 8. App performance: (a), (b), and (c) from the chess application and (d), (e), and (f) from the backtracking app. The first column of each row corresponds to response time with a specific execution environment (surrogate or local) and app state (the number of moves since game start for chess and the index of the model being loaded for backtracking). The second column is the average execution time of the task, and the third column show the energy consumption of the client.

baseline is incapable of adjusting its performance to different load conditions on the surrogate which further causes fluctuations in performance. In particular, the performance of the lower-end general purpose instances, m3.medium and m3.xlarge, degrades in heavy workload conditions (Fig. 10a) and thus ideally the app must offload to m3.large, m3.2xlarge or m2.4xlarge. Since our goal is to minimize response time and energy, EMCO offloads in these situations to m2.4xlarge. By using the diagnosed configuration, the chess app accelerates its responsiveness five-fold compared to local execution. Moreover, dynamic surrogate selection reduces 30 and 10 percent of the energy required in the communication with the cloud when compared with random surrogate selection in the chess and backtracking app scenarios, respectively.

For the backtracking app (bottom of Fig. 9a) offloading should ideally use as surrogate m3.medium, m3.large, m3.xlarge, m3.2xlarge or m2.4xlarge. However, under heavy workload conditions, the app should be offloaded to m3.large as it is the lowest general purpose instance which is capable of providing major gains in throughput in such circumstances. Since the generic framework is allocated with a powerful machine randomly, it can provide similar performance especially when the number of surrogates to choose from is small. However, the likelihood of achieving that same level of performance every time depends on the available cloud resources. Moreover, despite the fact that both frameworks provide similar response times rates to users, only EMCO is able to achieve further acceleration through the identification of reusable offloading results from its crowdsensing process. Thus, by using pre-cached functionality, EMCO accelerates about 30 times the response time of applications installed in devices such as i9250.

Fig. 9b compares the energy consumption of the client under EMCO and the baseline system. We observe accumulative gains in energy obtained by EMCO in a continuous offloading environment. This also can be seen as the difference between plain instrumentation and instrumentation with data analytic support. Based on these results, we can conclude that EMCO is more effective as it saves more energy in comparison with existing offloading frameworks. EMCO's decision engine also appears to be lighter in comparison to the mechanisms used by MAUI and related frameworks. Specifically, when we compare the MAUI linear programming (LP) mechanism with our decision engine using the same amount of parameters, the results show that EMCO consumes ≈12 percent less energy. The processing steps required to take a decision using LP depends on the number of parameters introduced in the engine.

The utilization of a powerful server is naturally associated with a cost. To mitigate the cost of running increasingly powerful servers, after pre-cache results are stored, they can be shared with lower capability servers, which can then respond to duplicated offloading requests. For instance, pre-cached results processed in m3.2xlarge can be transferred to m1.small, such that m1.small can then serve the result without any processing, yielding rates comparable to a m3.2xlarge. We conduct multiple experiments to determine the response time of a reusable result using different instances types. Fig. 9c shows the results. We can observe that the response time of a reusable result is stable on different instances types on Amazon, average ≈0.253, standard deviation (SD) ≈0.016. We also notice that a cached result served from any surrogate outperforms a fresh result computed by the most powerful surrogates shown in Figs. 8b and 8e.

5.4 Generalizability

We next demonstrate that the benefits of EMCO translate to a wider range of apps and that EMCO is capable of optimizing their performance in a wide set of execution contexts. We accomplish this through an evaluation of 10 apps in simulated offloading contexts. In our evaluation, each app creates virtual offloading requests whose context parameters are determined using the crowdsensed datasets

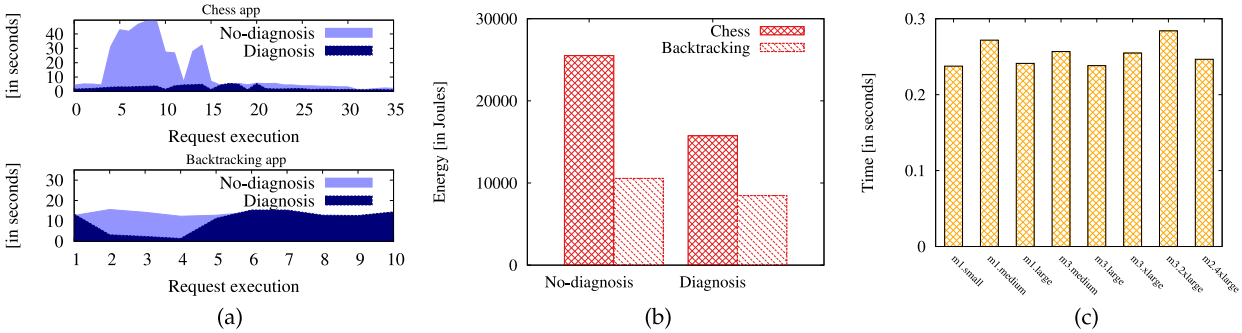


Fig. 9. (a) Request response time of both scenarios with and without diagnosis, (b) energy spent by the client, and (c) response time of pre-cached results in different instances types.

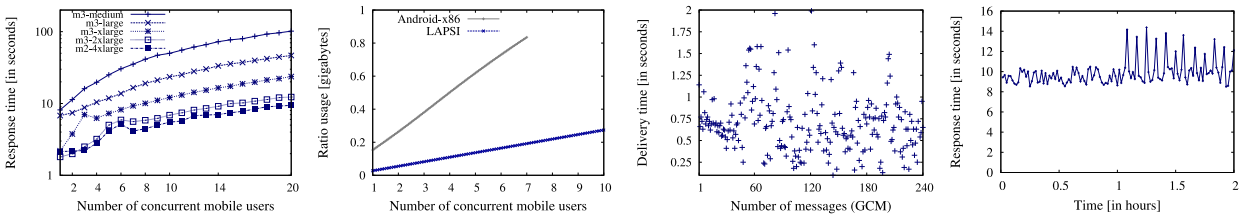


Fig. 10. (a) EMCO's ability for handling concurrent users. (b) Memory usage comparison between Android-x86 and LAPSI. (c) Response time of app reconfiguration via push notification. (d) CPU load of a EMCO server when handling up to 20 concurrent requests, first hour without execution monitoring second hour with monitoring.

considered in our work. By simulating the context we can ensure the parameters are representative of real world situations encountered by users. Simulation also helps to provide reproducibility for our analysis.

For the evaluation, we chose 10 open source apps from the F-droid[13] repository; see Table 2. We chose the most popular applications from the repository that had computationally intensive parts that benefit from offloading. We used a combination of community analysis and static profiling to determine which apps contained computationally intensive tasks. Specifically, Carat community analysis [19] was first used to assess the overall performance of the app. Only apps with a rate of 0.004 or higher were considered, which translates to roughly 7 hour battery lifetime for the device while the app is running continuously. Next, we performed static profiling over the app's code and validated its resource intensiveness using the *AT&T Video Optimizer*.[14] For each app, we chose all so-called outlier methods, which are methods with a high processing step count and whose execution time exceeds the average processing requirement of the rest of the app significantly. Note that, as we use execution time and processing counts for choosing which methods to simulate, this also helps to ensure the methods can indeed be accelerated. As an example, app1 contains one outlier method that requires ≈ 2 seconds to complete, while all remaining methods can be executed in 800 milliseconds or less. The final set of apps were from the following categories: puzzles (app1, app3, app6, app7, app8), games (app2, app4, app5, app10) and tools (app9); see Table 2 for the package names of the apps.

For each computationally intensive method we create virtual offloading requests where network latency was determined based on the NetRadar dataset (see Section 3). We assume that the cloud deployment consists of three instances (t2.nano, t2.medium, and m4.10xlarge). To determine the response time of the application, we calculated an expected runtime for the app and combined it with latency values of the simulated request. The expected runtime was determined by (i) calculating a bound for the execution of the method through static profiling, and (ii) scaling the estimated runtime with an acceleration level corresponding to the currently selected cloud instance derived through benchmarking of each cloud instance; see [30] for validation of this method. For each method we created 1,000 virtual requests, and calculated response time using both the diagnosis approach of EMCO and a no-diagnosis variant used in existing systems. The results in [30] show that, while the acceleration levels of cloud instances contain variation, the magnitude of performance improvements (or decrease) is stable between different types of cloud instances.

The results of our evaluation are shown in Table 2. In the evaluation we consider the upper bound on execution time obtained through static profiling as the response time requirement of each app, and calculate the fraction of times this can be achieved. EMCO is capable of satisfying the requirements in 78.88 percent of cases, whereas existing techniques succeed only in 47.28 percent of the requests. From the table we can also observe that the extent of delay in failed cases is significantly smaller for EMCO (454 ms) than for the no-diagnosis baseline (843 ms). In other words, EMCO not only performs significantly better in terms of meeting the response time requirements, but significantly reduces delays in cases where the requirements cannot be met.

To put the results into context, we performed a cost-benefit analysis by deriving utilities for EMCO and the baseline solution. First we calculate the overall cost of offloading by associating a monetary cost for serving each simulated request. The cost was determined so that higher cost reflects access to better resources. As each request was offloaded, we normalized the total cost of all requests to the range $[0, 1]$ so

13. https://f-droid.org/
14. https://developer.att.com/video-optimizer

TABLE 2
Crowd-Sensed Context Evaluation, Results of 1,000 Requests Per Method and Ratio of Request Times Below Responsiveness Requirements (% Fulfilled)

Id/App package/No.Methods	Resp. Time Req. [s]	Fulfilled [%]		Cost [0-1]		Exceeded by [s]	
		No-diag	Diag	No-diag	Diag	No-diag	Diag
app1 / org.ligi.gobandroid / 1	2	56.8%	90.5%	0.35	0.49	0.67	0.5
app2 / emre.android.atomix / 3	1	19.5%	47.7%	0.33	0.88	0.84	0.38
app3 / com.uberspot.a2048 / 2	2	58.9%	89.4%	0.34	0.50	0.92	0.49
app4 / org.bobstuff.bobball / 1	2.5	69%	92.9%	0.35	0.32	0.73	0.33
app5 / org.scoutant.blokish / 4	1.5	25.4%	56.1%	0.34	0.87	0.99	0.43
app6 / name.boyle.chris.sgtpuzzles / 1	2	50.8%	88%	0.33	0.60	0.86	0.56
app7 / io.github.kobuge.games.minilens / 3	2	61.6%	89.7%	0.34	0.50	0.88	0.51
app8 / com.petero.droidfish / 2	2.5	69.1%	94.5%	0.35	0.38	0.98	0.41
app9 / fr.ubordeaux.math.paridroid / 5	2	44.2%	91.5%	0.36	0.64	0.78	0.61
app10 / net.minetest.minetest / 1	1	17.5%	48.5%	0.36	1.0	0.78	0.32
Average for all apps	**1.85**	**47.28%**	**78.88%**	**0.34**	**0.62**	**0.843**	**0.454**

that a value of 0 corresponds to always using the slowest instance, and a cost of 1 to always using the fastest instance. For normalizing the costs, we assign weights $0.5, 1.0$, and 150.0 to the three instance types, which reflects the current pricing structure of Amazon Ireland and serves as representative example of real-world costs. To determine an utility for each request, we assume meeting the response time requirement provides a reward R, whereas failing to meet the requirement results in a penalty $P = \alpha t R$. Here α is a scaling factor that determines the importance of succeeding relative to failing to meet the user specified response rate requirements. The variable t is the extent at which the request exceeds the response rate requirement. Using the values in Table 2, it can be shown that EMCO provides better utility-cost ratio than existing approaches whenever $\alpha > 0.0808$, i.e., whenever the penalty for failing the response is at least 8 percent of the value of succeeding. If the penalty of exceeding the response time is very low, offloading to the slowest instance is naturally the best policy. To this end, the higher cost of EMCO compared to the no-diagnosis case is a direct consequence of having higher success rate, and hence using more resources. In practice we would expect the penalty of failure exceeding the reward of meeting the response time requirement (i.e., $\alpha > 1$), implying that EMCO performs better in most cases of practical interest.

5.5 Scalability

We next consider the scaling performance of the server under heavy load of incoming requests. We carry out our evaluation by straining the cloud infrastructure with synthetic requests created by our trace simulator (described in Section 3.3) and measuring throughput as a function of load. In our benchmarks, we only consider server instances that provide better response time than local execution (i.e., m1.small, m1.medium, and m1.large were excluded). As part of the evaluation we also compare the scaling performance of our LAPSI surrogate against a deployment using Android-x86 surrogates.

Fig. 10a shows the response rate of the different cloud instances as the number of concurrent users increases. We observe that the capability to handle concurrent requests mirrors the available resources with the response rate slowing linearly as the number of concurrent requests increases. As expected, we can also observe that the magnitude of the slope depends on the available resources of the server with the most powerful servers having the smallest coefficients. Three surrogates (m3.xlarge, m3.2xlarge, m2.4xlarge) can support over 20 concurrent users before the response rate drops below local execution, whereas two of the surrogates (m3.medium, m3.large) can support 10 concurrent users before dropping below local execution. The response times include also latency induced by EMCO, which typically is around 100 milliseconds. Fig. 10b shows the memory use of LAPSI surrogates compared to Android-x86 surrogates. The memory use of the Android-x86 surrogates rapidly increases, with 7 concurrent users requiring over a gigabyte of writable memory. The total memory footprint (including shared memory) of these 7 users fills the 8 GB memory of a m3.large instance, so that no more Android-x86 surrogates can run on the instance. In contrast, with our LAPSI deployment the increase in memory use is modest. To put our result into context, an Android-x86 deployment would need to boot up a new instance to serve each 7 users wishing to use the same application at the same time, whereas our deployment can serve over 28 users simultaneously with the same amount of resources. This makes LAPSI better suited for deployments targeting communities of users. In EMCO, horizontal scaling is achieved using a notification server as load balancer. The notifications allow the mobile to reconfigure the surrogate in which it is currently subscribed. As part of the experiment we also benchmark our push notification mechanism by measuring its delivery rate. We fix the message size of the notifications to 254 bytes, which is the minimum amount of data required to reconfigure the surrogate. We first send 15 consecutive messages (once per second), after which we have a half an hour period of inactivity. We repeat these phases over a 8 hour period, resulting in a total of 240 messages. The sleep periods are used to mitigate the possibility of the cloud provider associating our experiments with denial of service attack and the duration of the experiments was designed to collect samples at different times of day.

Fig. 10c shows the response time results for the notification messages. The average time is 0.75 seconds (median 0.66, SD 0.69). Note that the response time depends on the underlying messaging protocol and the availability of the server at the cloud provider [33]. The overall delay in surrogate migration depends also on the events taking place on the server side. The slowest case is when a new child node is created (i.e., when server utilization reaches maximum limit) which results in approximately one minute reconfiguration time. This consists of allocating a new server by the cloud provider (approximately 40 seconds on Amazon) and

reconfiguring clients through push notifications. When a child node is removed (i.e., when server utilization falls below a lower limit), reconfiguration is faster as the subscribed devices are simply moved to the parent node and the only delay results from push notifications.

Finally, we demonstrate the benefits of using a separate monitoring server as part of the EMCO architecture. Fig. 10d shows the performance of a m2.4xlarge surrogate during a 2h period with 20 concurrent users. During the first hour, no performance metrics are gathered, whereas during the second hour the surrogate was configured to collect performance measures. From the figure we can clearly see that monitoring cloud execution burdens the surrogate, degrading its performance, whereas the performance of the surrogate is stable when no metrics are gathered. Accordingly, our experiment demonstrates that separating the monitoring of cloud execution to a separate server enables faster and more stable response time.

5.6 Optimality of Decision Making

As the final step of evaluation we consider the optimality of the EMCO decision engine. To carry out the evaluation, we consider our benchmark evaluation in Section 5.2 and establish a ground truth by comparing the computational effort of the device between remote and local execution. We use crowsourced data to characterize execution contexts to allow EMCO to make informed decisions.

We conduct our evaluation using five-fold cross-validation, i.e., all measurements in four folds are used to train our decision engine (i.e., C4.5 decision tree classifier) and its predictions are compared to ground truth for the remaining fold. In 99.82 percent of the situations, EMCO was correctly able to choose the option with the smallest energy cost between local and remote execution. Generally the performance depends on the extent of intersecting areas in the rate distributions, which represent conditions where the benefit of offloading is uncertain or counterproductive. In our experiments, the amount of training data was sufficient for learning a model that has few intersections. During early stages of the app, the performance is likely to be lower (worst-case performance 77 percent). However, as shown in Section 3, a community of moderate size typically produces the necessary amount of measurements for minimizing intersections arising from uncertainty within a day or two.

6 Discussion

A key benefit of EMCO is that it facilitates bootstrapping of offloading decision making. While the amount of data that is required for a characterizing the execution of a particular app depends on the complexity of the app, its usage contexts, and execution environment, in most cases EMCO can provide substantial improvements in the bootstrapping process. Indeed, the more variability there is in the execution, the more substantial the benefits EMCO can bring through community analysis. We note that, while not a core focus for the present paper, another benefit of EMCO is that the number of intersecting segments between local and remote distributions provides an estimate of the uncertainty in the app execution, which in turn can be used to estimate the point where sufficient amount of data has been collected. Estimating this on individual devices is difficult as one device typically only experiences a small set of execution contexts and the performance of the system is likely to degrade when a previously unseen context occurs.

As with any other history-based approach, the performance of EMCO gradually improves as more evidence is gathered from the community of devices. When no data is available to bootstrap the system, the decision inference considers only contextual parameters collected in real-time from the context profiles similarly to other proposed frameworks. However, even in such case, major performance improvements can be obtained compared to other frameworks. For example, our dynamic surrogate migration mechanism is capable of providing access to surrogates in a service-oriented style. We have also experimentally demonstrated that, once sufficient amounts of evidence have been collected, our framework outperforms existing solutions by providing better support for making offloading decisions. As most of the evidence analysis takes place in the cloud, the effort required by the mobile device to execute EMCO is minimal compared with other frameworks. For example, both linear programming [2] and Hidden markov models [37] require processing raw data from system profilers in order to determine when to offload. In contrast, EMCO avoids raw data processing as the constructed models are sent from the cloud to mobile devices. While transferring data to the cloud requires some energy, these costs typically are minimal [19] and their effect on battery can be mitigated by piggybacking data as part of application requests.

Recently, the *Android runtime* (ART)[15] has been introduced as a novel runtime compiler that is intended as replacement for Dalvik. ART has been designed to operate using Dex bytecode, which is the same format as used by Dalvik and hence LAPSI and EMCO can be easily ported to ART. Compared to our LAPSI implementation, the main difference with ART is that it implements *Ahead-of-Time* (AOT) compilation instead of *Just-in-Time* (JIT). This means that an application that uses ART is compiled once during the first execution, and the static portions of the app will not be compiled again for subsequent executions. This is closely related to the pre-caching functionality used as part of EMCO. However, our solution is more general as it has been designed to support also collaborative reuse of method invocations between devices.

Our results have shown the potential of migrating cloud surrogates dynamically based on concurrent load, execution context, and other factors. This differs from existing systems which either provide no means of effectively matching the client with the cloud infrastructure or make the decision the responsibility of the app designer [6]. In the former case, the effectiveness depends on the optimality of the initial allocation, whereas in the latter case the effectiveness is reliant on the designers knowledge of the intricacies of offloading contexts. Our use of crowdsensing to monitor cloud execution provides more flexibility in the process, allowing us to dynamically determine the optimal acceleration level and to adapt to varying loads on the cloud size. EMCO also lends itself to more elaborate cloud allocation, e.g., by associating a cost (or utility) for energy, surrogate type, and responsiveness, we can minimize the cost of surrogate allocation for each client.

EMCO can be applied in many variable context scenarios. For example, an interstate bus line or a high-speed train with onboard WiFi is suspectible to constantly changing network conditions, which can result in poor app responsiveness. Routing offloading requests of connected clients with EMCO mitigates response time variations and provides a superior

15. https://source.android.com/devices/tech/dalvik/

experience for latency-critical applications. As another example, EMCO can diagnose situations for opportunistic offloading. For example we can use the community to identify application combinations that increase CPU usage while running in the background. From our study presented in section 3.1, we can observe 60-70 percent CPU use with 13.65 percent of apps (TripAdvisor, Whatsapp, Facebook, Waze, among others). By detecting currently active apps, we can utilize offloading for less intensive tasks to reduce energy drain caused by heavy CPU usage. However, since resource allocation has a higher impact in computational offloading compared to other opportunities, we focus our diagnosis on resource allocation. A wider range of context values for opportunistic offloading can be handled, e.g., using a decision tree [32] and ranking the context values in order of their impact on offloading.

EMCO also implements an energy profiler that estimates the energy required from a portion of code. However, estimation of energy consumption with any kind of software tool, e.g., PowerTutor, suffers from an estimation error. EMCO overcomes this problem by supplementing local estimates made on the device with community-based estimates that aggregate over a large community of devices. By collecting many samples from the same task, EMCO is able to reduce the error to determine when a computational task requires high energy consumption for the device or not.

7 Summary

The present paper contributed by proposing crowdsourced evidence traces as a novel mechanism to optimize offloading decision making. As our first contribution we performed an analysis of two crowdsensed datasets to demonstrate that crowdsensing is a feasible mechanism for characterizing parameters relevant for offloading decision making. As our second contribution, we proposed a toolkit and platform, EMCO, that analyzes these traces to identify reusable and generic contexts where offloading is beneficial. Further performance gains are achieved through caching and a novel cloud surrogate LAPSI that we have developed and integrated as part of EMCO. Experiments carried out on a testbed deployment in Amazon EC2 Ireland demonstrated that EMCO can consistently accelerate app execution while at the same time reducing application footprint. In addition, we evaluate offloading optimization through crowd-sensed context diversity over multiple off-the-shelf applications. We also demonstrated that EMCO provides better scalability than current cloud platforms, being able to serve a larger number of clients without variations in performance. We provide our framework, use case and tools as open source in GitHub along with the LAPSI located in Amazon EC2.

Acknowledgments

The authors thank the anonymous reviewers for their insightful comments. This research has been supported, in part, by projects 26211515 and 16214817 from the Research Grants Council of Hong Kong. This work also was supported by the Jorma Ollila Grant 201620040.

References

[1] S. Ickin, K. Wac, M. Fiedler, L. Janowski, J. Hong, and A. K. Dey, "Factors influencing quality of experience of commonly used mobile applications," *IEEE Commun. Mag.*, vol. 50, no. 4, pp. 48–56, Apr. 2012.

[2] E. Cuervo, et al., "MAUI: Making smartphones last longer with code offload," in *Proc. ACM Int. Conf. Mobile Syst. Appl. Serv.*, Jun. 2010, pp. 49–62.

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

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proc. Annu. Conf. Comput. Syst.*, Apr. 2011, pp. 301–314.

[5] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, Oct. 2012, pp. 93–106.

[6] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "COSMOS: Computation offloading as a service for mobile devices," in *Proc. ACM Int. Symp. Mobile Ad Hoc Netw. Comput*, Aug. 2014, pp. 287–296.

[7] A. Saarinen, M. Siekkinen, Y. Xiao, J. K. Nurminen, M. Kemppainen, and P. Hui, "SmartDiet: Offloading popular apps to save energy," in *Proc. ACM Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, Aug. 2012, pp. 297–298.

[8] H. Flores, et al.,"Social-aware hybrid mobile offloading," *Pervasive Mobile Comput.*, vol. 36, pp. 25–43, 2017.

[9] M. Conti and M. Kumar, "Opportunities in opportunistic computing," *IEEE Comput.*, vol. 43, no. 1, pp. 42–50, Jan. 2010.

[10] E. Peltonen, E. Lagerspetz, P. Nurmi, and S. Tarkoma, "Energy modeling of system settings: A crowdsourced approach," in *Proc. Int. Conf. Pervasive Comput. Commun.*, Mar. 2015, pp. 37–45.

[11] H. Flores and S. Srirama, "Mobile code offloading: Should it be a local decision or global inference?" in *Proc. ACM Int. Conf. Mobile Syst. Appl. Serv.*, Jun. 2013, pp. 539–540.

[12] Y. Kwon, et al.,"Mantis: Efficient predictions of execution time, energy usage, memory usage and network usage on smart mobile devices," *IEEE Trans. Mobile Comput.*, vol. 14, no. 10, pp. 2059–2072, Oct. 2015.

[13] S. Sigg, E. Lagerspetz, E. Peltonen, P. Nurmi, and S. Tarkoma, "Sovereignty of the Apps: There's more to Relevance than Downloads," 2016. [Online]. Available: https://arxiv.org/abs/1611.10161

[14] H. Ma, D. Zhao, and P. Yuan, "Opportunities in mobile crowd sensing," *IEEE Commun. Mag.*, vol. 52, no. 8, pp. 29–35, Aug. 2014.

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

[16] M.-R. Ra, et al., "Odessa: Enabling interactive perception applications on mobile devices," in *Proc. ACM Int. Conf. Mobile Syst. Appl. Serv.*, Jul., 2011, pp. 43–46.

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

[18] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Comput.*, vol. 43, no. 4, pp. 51–56, 2010.

[19] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma, "Carat: Collaborative energy diagnosis for mobile devices," in *Proc. ACM Conf. Embedded Netw. Sensor Syst.*, Nov. 2013, Art. no. 10.

[20] S. Sonntag, J. Manner, and L. Schulte, "Netradar-measuring the wireless world," in *Proc. IEEE Int. Symp. Model. Optim. Mobile Ad Hoc Wireless Netw.*, May 2013, pp. 29–34.

[21] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *Proc. ACM SIGOPS Eur. Workshop*, Jul. 2002, pp. 87–92.

[22] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Comput. Syst.*, vol. 29, no. 1, pp. 84–106, 2013.

[23] P. Bahl, R. Y. Han, L. E. Li, and M. Satyanarayanan, "Advancing the state of mobile cloud computing," in *Proc. ACM MobiSys Workshop*, Jun. 2012, pp. 21–28.

[24] M. A. Hassan, K. Bhattarai, Q. Wei, and S. Chen, "POMAC: Properly offloading mobile applications to clouds," *Energy J.*, vol. 25, 2014, Art. no. 50.

[25] Y. Kwon, H. Yi, D. Kwon, S. Yang, Y. Cho, and Y. Paek, "Precise execution offloading for applications with dynamic behavior in mobile cloud computing," *Pervasive Mobile Comput.*, vol. 27, pp. 58–74, 2016.

[26] M. V. Barbera, S. Kosta, A. Mei, V. C. Perta, and J. Stefa, "Mobile offloading in the wild: Findings and lessons learned through a real-life experiment with a new cloud-aware system," in *Proc. Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2014, pp. 2355–2363.

[27] H. Flores and S. Srirama, "Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning," in *Proc. ACM MobiSys Workshop*, Jun. 2013, pp. 9–16.
[28] F. A. Silva, et al., "Benchmark applications used in mobile cloud computing research: A systematic mapping study," *J. Supercomput.*, vol. 72, no. 4, pp. 1431–1452, 2016.
[29] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile code offloading: From concept to practice and beyond," *IEEE Commun. Mag.*, vol. 53, no. 4, pp. 80–88, Apr. 2015.
[30] H. Flores, et al., "Modeling mobile code acceleration in the cloud," in *Proc. Annu. IEEE Int. Conf. Distrib. Comput. Syst.*, Jun. 5–8, 2017, pp. 480–491.
[31] K. Nozaki, H. Ishibuchi, and H. Tanaka, "A simple but powerful heuristic method for generating fuzzy rules from numerical data," *Fuzzy Sets Syst.*, vol. 86, no. 3, pp. 251–270, 1997.
[32] E. Peltonen, E. Lagerspetz, P. Nurmi, and S. Tarkoma, "Constella: Crowdsourced system setting recommendations for mobile devices," *Pervasive Mobile Comput.*, vol. 26, pp. 71–90, 2016.
[33] H. Flores and S. N. Srirama., "Mobile cloud messaging supported by XMPP primitives," in *Proc. ACM MobiSys Workshop*, Jun. 25–28, 2013, pp. 17–24.
[34] M. Toyama, S. Kurumatani, J. Heo, K. Terada, and E. Y. Chen, "Android as a server platform," in *Proc. IEEE Consum. Commun. and Netw. Conf.*, Jan. 2011, pp. 1181–1185.
[35] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Int. IEEE/ACM/IFIP Conf. Hardware/Software Codesign Syst. Synthesis (CODES+ ISSS 2010)*, Scottsdale, AZ, USA, Oct. 2010.
[36] M. Mazzucco, M. Vasar, and M. Dumas, "Squeezing out the cloud via profit-maximizing resource allocation policies," in *IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, Aug. 2012, pp. 19–28.
[37] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, "Sociablesense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing," in *Proc. Annu. Int. Conf. Mobile Comput. Netw.*, Sep. 2011, pp. 73–84.

Huber Flores received the PhD degree in computer science from the Faculty of Mathematics and Computer Science, University of Tartu, Estonia. He is a postdoctoral research scientist in the Department of Computer Science, the University of Helsinki, Finland. He is also member of the ACM (SIGMOBILE) and the IEEE. His major research interests include mobile offloading, mobile middleware architectures, and mobile cloud computing.

Pan Hui received the PhD degree from the Computer Laboratory, University of Cambridge. He is the Nokia chair in data science and professor of computer science with the University of Helsinki, the director of the HKUST-DT System and Media Lab with the Hong Kong University of Science and Technology, and an adjunct professor of social computing and networking with Aalto University. He worked for Intel Research Cambridge and Thomson Research Paris and was a distinguished scientist with the Deutsche Telekom Laboratories. He has published more than 200 research papers and with more than 13,000 citations. He is an associate editor of the IEEE Transactions on Mobile Computing and the IEEE Transactions on Cloud Computing, an IEEE Fellow, and an ACM distinguished scientist.

Petteri Nurmi received the PhD degree in computer science from the University of Helsinki, in 2009, and he has been an adjunct professor of computer science with the University of Helsinki since 2012. He is a senior researcher with the University of Helsinki and science advisor at Moprim. His research focuses on sensing, covering system design, algorithms, theoretical models, and new application areas. He has published more than 80 scientific publications.

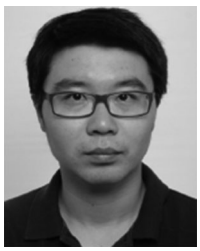
Eemil Lagerspetz received the PhD degree from the University of Helsinki, in 2014. He is a postdoctoral scholar with the University of Helsinki. His research pertains to the analysis of large datasets, mobile computing, and energy efficiency. He received the University of Helsinki Doctoral Dissertation Award in 2015, Jorma Ollila Grant in 2015, and the Marc Weiser Best Paper Award at IEEE PerCom in 2015.

Sasu Tarkoma received the PhD degree in computer science from the University of Helsinki, in 2006. He has been a professor of computer science with the University of Helsinki since 2009. He has worked in the IT industry as a consultant and chief system architect as well as principal researcher and laboratory expert at the Nokia Research Center. He has more than 140 scientific publications, 4 books, and 4 US Patents.

Jukka Manner received the PhD degree in computer science from the University of Helsinki, in 2004. He is a full professor (tenured) of networking technology with the Aalto University, Department of Communications and Networking (Comnet). His research and teaching focuses on networking, software and distributed systems, with a strong focus on wireless and mobile networks, transport protocols, energy efficient ICT, and cyber security.

Vassilis Kostakos is a professor of human computer interaction in the School of Computing and Information Systems, University of Melbourne. He has held appointments with the University of Oulu, University of Madeira, and Carnegie Mellon University. He has been a fellow of the Academy of Finland Distinguished Professor Programme, and a Marie Curie Fellow. He conducts research on ubiquitous and pervasive computing, human-computer interaction, and social and dynamic networks.

Yong Li received the PhD degree in electronic engineering from Tsinghua University, Beijing, China, in 2012. He is currently a faculty member of the Department of Electronic Engineering, Tsinghua University. He has previously worked as a visiting research associate with Telekom Innovation Laboratories, The Hong Kong University of Science and Technology. His research interests include the areas of networking and communications, including mobile opportunistic networks, device-to-device communication, software-defined networks, network virtualization, and future Internet.



Xiang Su received the PhD degree in technology from the University of Oulu. He is a postdoctoral researcher in computer science in the Center of Ubiquitous Computing, University of Oulu. His research interests include semantic technologies, the Internet of Things (IoT), knowledge representations, and context modeling and reasoning.