Mobile Code Offloading: From Concept to Practice and Beyond

Huber Flores, Pan Hui, Sasu Tarkoma, Yong Li, Satish Srirama, and Rajkumar Buyya

ABSTRACT

The emerging mobile cloud has expanded the horizon of application development and deployment with techniques such as code offloading. While offloading has been widely considered for saving energy and increasing responsiveness of mobile devices, the technique still faces many challenges pertaining to practical usage. In this article, we adopt a systemic approach for analyzing the components of a generic code offloading architecture. Based on theoretical and experimental analysis, we identify the key limitations for code offloading in practice and then propose solutions to mitigate these limitations. We develop a generic architecture to evaluate the proposed solutions. The results provide insights regarding the evolution and deployment of code offloading.

INTRODUCTION

Mobile and cloud computing convergence is shifting the way in which telecommunication architectures are designed and implemented [1]. In the presence of network connectivity to bind mobile and cloud resources, the potential of code offloading lies in the ability to sustain power-hungry applications by releasing the energy consuming resources of the smartphone from intensive processing. Multiple research works have proposed different code offloading strategies to empower smartphone apps with cloud-based resources [5–9]. However, the utilization of code offloading is debatable in practice as the approach has been demonstrated to be ineffective in increasing the remaining battery life of mobile devices [2].

The effectiveness of an offloading system is determined by its ability to infer where the execution of code (local or remote) represents less computational effort for the mobile, such that by deciding *what*, *when*, *where*, *and how to offload* correctly, the device obtains a benefit. Code offloading is productive when the device saves energy without degrading the normal response time of the apps, and counterproductive when the device wastes more energy executing a computational task remotely rather than locally. Current works offer partial solutions that ignore the majority of these considerations in the inference process. Most of the proposals demonstrate the utilization of code offloading in a controlled environment by connecting to nearby low-latency servers (e.g., lab setups) and inducing the code to become resource intensive during runtime [3] (e.g., passing an input that requires lot of processing). As a result, in practice, in most cases, computational offloading is counterproductive for the device [2, 7]. Thus, at this point, the main questions about the strategy are *can code offloading be utilized in practice?* and *what are the issues that prevent code offloading from yielding a positive result?*

This article takes a systemic perspective to answer these questions by analyzing in detail the components that form an offloading architecture. The work highlights the drawbacks of code offloading architecture and proposes solutions that exploit intrinsic cloud features (e.g., massive data analysis) in order to overcome the issues. Unlike other offloading architectures that focus on specific issues in the offloading process, such as *what or when*, the aim of our proposal is to generalize an offloading service that can estimate the most effective offloading outcome for the device by focusing on multiple perspectives at once (*what*, *when*, *where*, *how*, etc.) Based on the solutions, we present and evaluate use cases, which give insights about how the computational offloading process can be managed to accelerate the response time of mobile applications without inducing extra overhead in the device, and to reduce the offloading traffic of computational requests. Moreover, the work also emphasizes the importance of the cloud beyond the ordinary provisioning of services.

The rest of the article is structured as follows. We describe the essential elements and functionality of an offloading architecture. We describe an overview of the obstacles that inhibit the adoption of code offloading. We present the solutions we envision to potentiate the applicability of the approach, and finally, we evaluate how much gain can be obtained by leveraging the proposed solutions.

BACKGROUND

Code offloading has evolved considerably from its initial notion of cyber-foraging [1]. Table 1 describes most relevant proposals in code offloading. The table compares the key features of the offloading architectures: the main goal,

Huber Flores and Satish Srirama are with the University of Tartu, Estonia.

Pan Hui is with the Hong Kong University of Science and Technology. He is also affiliated with Telekom Innovation Laboratories and Aalto University.

Sasu Tarkoma is with the University of Helsinki.

Yong Li is with Tsinghua University.

Rajkumar Buyya is with the University of Melbourne.

0163-6804/15/$25.00 © 2015 IEEE

	Code offloading strategies				Mobile perspective	Cloud perspective
Framework	Main goal	Code profiler	Offloading adaptation context	Offloading characterization	Applications effect	Features exploited (Besides server)
MAUI [5]	Energy saving	Manual annotations	Mobile (*what*, *when*)	None	Low resource consumption, Increased performance	None
CloneCloud [6]	Transparent code migration	Automated process	Mobile (*what*, *when*)	None	Accelerate responsiveness	None
ThinkAir [8]	Scalability	Manual annotations	Mobile + Cloud (*what*, *when*, *how*)	None	Increased performance	Dynamic allocation and destruction of VMs
COMET [9]	Transparent code migration (DSM)	Automated process	Mobile (*what*, *how*)	None	Average speed gain 2.88×	None
EMCO [10]	Energy saving, Scalability (Multi-tenancy)	Automated process	Mobile + cloud (*what*, *when*, *where*, *how*)	Based on historical crowdsourcing data	Based on context (Low resource consumption, increased responsiveness, etc.)	Dynamic allocation and destruction of VMs, Big data processing, Characterization-based utility computing

Table 1. Code offloading approaches from the mobile and cloud perspectives.

how code is profiled, the adaptation context, the characterization of the offloading process, and how code offloading is exploited from the mobile and cloud perspectives. From the table, the main goal defines the actual benefit of using the associated framework. The mechanism used to profile code provides information about the flexibility and integrability of the system. The adaptation context specifies the considerations taken by the system to offload. The characterization means whether the offloading system has a priori knowledge or not about the effects of code offloading for the components of the system. Finally, the exploitation highlights the mobile benefits obtained from going cloud-aware, and the features of the cloud that are leveraged to achieve those benefits. Moreover, we can also observe that currently, most of the effort has been focused on providing the device with an offloading logic based on its local context. To gain an understanding of the actual benefits and functionality of code offloading, we described the key properties of an offloading architecture.

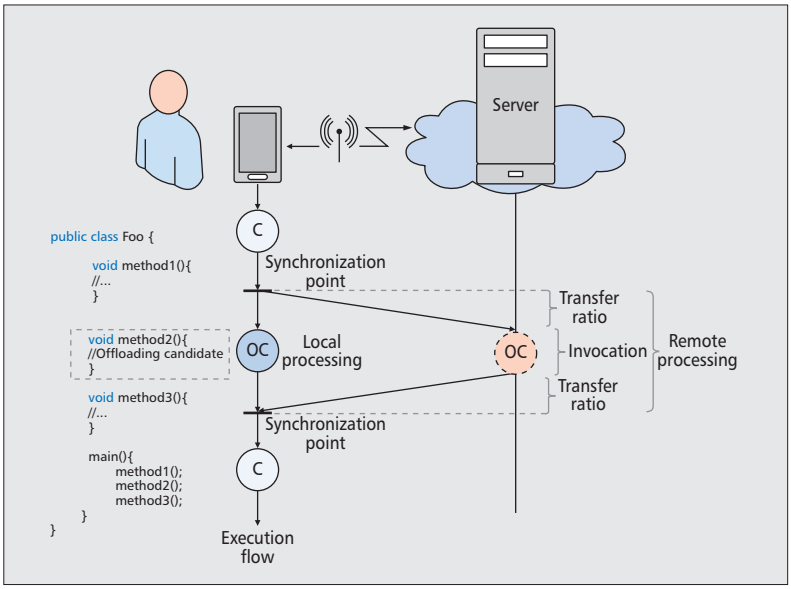

Figure 1. A code offloading architecture: components and functionalities.

Code Offloading

Offloading is the opportunistic process that relies on remote servers to execute code delegated by a mobile device. In this process, the mobile is granted the local decision logic to detect resource-intensive portions of code, such that in the presence of network communication, the mobile can estimate where the execution of code will require less computational effort (remote or local), which leads the device to save energy [9]. The evaluation of the code requires consideration of different aspects, for instance, *what code to offload* (e.g., method name); *when to offload* (e.g., round-trip times thresholds); *where to offload* (e.g., type of cloud server); *how to offload* (e.g., split code into *n* processes); and so on. Most of the proposals in the field do not cover all these aspects; thus, we describe a basic offloading architecture, which is shown in Fig. 1. The architecture consists of two parts, a client and a server. The client is composed of a code profiler, system profilers, and a decision engine. The server contains the surrogate platform to invoke and execute code. Each component is described below.

The *code profiler* is in charge of determining *what to offload*. Thus, portions (*C*) of code — *Method, Thread*, or *Class* — are identified as offloading candidates (OCs). Code partitioning requires the selection of the code to be offloaded. Code can be partitioned through a diversity of strategies; for instance, a software developer can explicitly select the code that should be offloaded using special static annotations [5] (e.g., *@Offloadable*, *@Remote*). Other strategies

Annotations can cause poor flexibility to execute the app in different mobile devices. Similarly, automated strategies are shown to be ineffective and require major low-level modifications in the core system of the mobile platform, which may lead to privacy and security issues.

analyze the code implicitly during runtime by an automated mechanism [6]. Thus, once the application is installed in the device, the mechanism selects the code to be offloaded. In order to estimate whether or not a portion of code is intensive, the mechanism implements strategies such as static analysis and history traces. Automated mechanisms are preferable over static ones as they can adapt the code to be executed in different devices. Thus, automated mechanisms overcome the problem of brute force development in code offloading, which consists of adapting the application every time it is installed in a different device.

System profilers are responsible for monitoring multiple parameters of the smartphone, such as available bandwidth, data size to transmit, and energy to execute the code. These parameters influence *when to offload* to the cloud. Conceptually, the offloading process is optional, and should take place when the effort required by the mobile to execute the OC is lower in the case of remote invocation than local execution. Otherwise, offloading is not encouraged as excessive amount of energy and time is consumed in transmission of data to the cloud.

The *decision engine* is a reasoner that infers *when to offload* to the cloud. The engine retrieves the data obtained by the system and code profilers, and applies certain logic over them (linear programming, fuzzy logic, Markov chains, etc.) such that the engine can measure whether or not the handset obtains a concrete benefit from offloading to the cloud. If the engine infers a positive outcome, the mechanism to offload is activated, and the code is invoked remotely; otherwise, the code is executed locally. A mobile offloads to the cloud in a transfer ratio that depends on the size of the data and the available bandwidth [4]. Usually, when code offloading is counterproductive for the device, it is due to a wrong inference process, which is inaccurate based on the scope of observable parameters that the system profilers can monitor [7].

The *surrogate platform* is the remote server located in proximity of the device or in the cloud, which contains the environment to execute the intermediate code sent by the mobile (e.g., Android-x86). The remote invocation tends to accelerate the execution of code as the processing capabilities of the surrogate are higher than those of most smartphones, which is translated into higher app responsiveness for the mobile user.

Challenges and Technical Problems

Computational offloading for smartphones has not changed drastically from its core principles [3]. However, the effectiveness of its implementation in practice shows it to be mostly unfavorable for the device outside controlled environments. In fact, the utilization of code offloading in real scenarios proves to be mostly negative [7], which means that the device spends more energy in the offloading process than the actual energy that is saved. As a consequence, the technique is far from being adopted in the design of future mobile architectures. Our goal is to highlight the challenges and obstacles in deploying code offloading. The issues are described below.

Inaccurate Code Profiling — Code profiling is one of the most challenging problems in an offloading system, as the code has non-deterministic behavior during runtime, which means that it is difficult to estimate the running cost of a piece of code considered for offloading. A portion of code becomes intensive based on factors [7] such as the user input that triggers the code, type of device, execution environment, available memory, and CPU. Moreover, once code is selected as *OC*, it is also influenced by many other parameters of the system that come from multiple levels of granularity (e.g., communication latency, data size transferred). As a result, code offloading suffers from a sensitive trade-off that is difficult to evaluate; thus, code offloading can be productive or counterproductive for the device. Most of the proposals in the field are unable to capture runtime properties of code, which makes them ineffective in real scenarios.

Integration Complexity — The adaptation of code offloading mechanisms within the mobile development life cycle depends on how easily the mechanisms are integrated and how effective the approach is in releasing the device from intensive processing. However, implementation complexity does not necessarily correlate with effective runtime usage. In fact, some of the drawbacks that make code offloading fail are introduced at development stages; for example, in the case of code partitioning, which relies on the expertise of the software developer, portions of code are annotated statically, which may cause unnecessary code offloading that drains energy. Moreover, annotations can cause poor flexibility to execute the app in different mobile devices. Similarly, automated strategies are shown to be ineffective and require major low-level modifications in the core system of the mobile platform, which may lead to privacy and security issues.

Dynamic Configuration of the System — Next generation mobile devices and the vast computational choices in the cloud ecosystem make the offloading process a complex task, as depicted in Fig. 2a. Although savings in energy can be achieved by releasing the device from intensive processing, a computational offloading request requires meeting the requirements of a user's satisfaction and experience, which is measured in terms of responsiveness of the app. Consequently, in the offloading decision, a smartphone has to consider not just potential savings in energy, but also has to ensure that the acceleration in the response time of the request will not decrease. This is an evident issue as the computational capabilities of the latest smartphones are comparable to some servers running in the cloud. For instance, consider two devices, Samsung Galaxy S (i9000) and Samsung Galaxy S3 (i9300), and two Amazon instances, m1.xlarge and c3.2xlarge. In terms of mobile application performance, offloading intensive code from

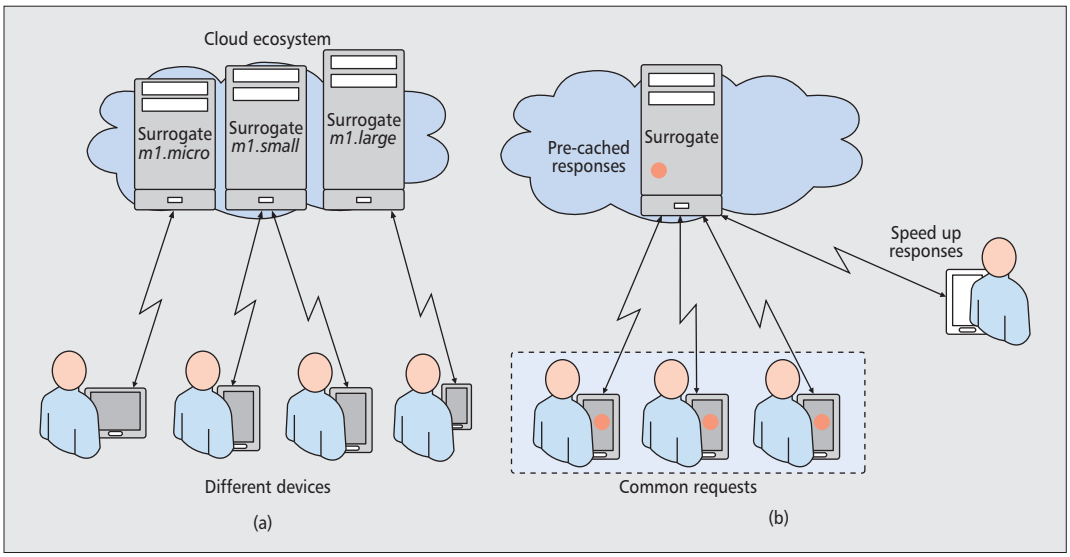

Figure 2. a) Characterization of the offloading process that considers smartphone diversity and the vast cloud ecosystem; b) acceleration of a code offloading request via pre-cached results.

From a cloud point of view, allocation of resources cannot occur indiscriminately based on processing capabilities of the server as the use of computational resources is associated with a cost. Consequently, the need for policies for code offloading systems are necessary considering both the mobile and the cloud.

i9000 to m1.xlarge increases the responsiveness of a mobile application at comparable rates to an i9300. However, offloading from i9300 to m1.xlarge does not provide the same benefit. Thus, to increase responsiveness it is necessary to offload from i9300 to c3.2xlarge (refer to results later). It is important to note, however, that constantly increasing the capabilities of the back-end does not always speed up the execution of code exponentially, as in some cases, the execution of code depends on how the code is written; for example, code is parallelizable for execution into multiple CPU cores (parallel offloading) or distribution into large-scale graphics processing units (GPU offloading).

Offloading Scalability and Offloading as a Service — Typically, in a code offloading system, the code of a smartphone app must be located in both the mobile and server as in a remote invocation, a mobile sends to the server, not the intermediate code, but the data to reconstruct that intermediate representation such that it can be executed. As a result, an offloading system requires the surrogate to have a similar execution environment as the mobile. To counter this problem, most of the offloading systems propose relying on the virtualization of the entire mobile platform in a server (Android-x86, .Net framework, etc.), which tends to constrain the CPU resources and slows down performance. The reason is that a mobile platform is not developed for large-scale service provisioning. As a result, offloading architectures are designed to support one user at a time, in other words, one server for each mobile [8, 11]. This restrains the features of the cloud for multi-tenancy and utility computing. Moreover, while a cloud vendor provides the mechanisms to scale service-oriented architectures (SOAs) [12] on demand, such as Amazon autoscale, it does not provide the means to adapt such strategies to a computational offloading system as the requirements to support code offloading are different. The requirements of a code offloading system are based on the perception that the user has of the response time of the app. The main insight is that a request should increase or maintain a certain quality of responsiveness when the system handles heavy loads of computational requests. Thus, a code offloading request cannot be treated indifferently. The remote invocation of a method has to be monitored under different systems' throughput to determine the limits of the system to not exceed the maximum number of invocations that can be handled simultaneously without losing quality of service. Furthermore, from a cloud point of view, allocation of resources cannot occur indiscriminately based on the processing capabilities of the server as the use of computational resources is associated with a cost. Consequently, the need for policies for code offloading systems are necessary considering both the mobile and the cloud.

Mobile Crowdsourcing for Computational Offloading: Vision

Despite the large amount of related works [5–11], the instrumentalization of apps alone is insufficient to adopt computational offloading in the design of mobile systems that rely on the cloud. Computational offloading in the wild can impose more computational effort on the mobile rather than reduce processing load [2]. In contrast to existing works, we overcome the limitations of computational offloading in practice by analyzing how a particular smartphone app behaves in a community of devices [7]. Computational crowdsourcing strategies are viable solutions to understand potential problems in software applications with high accuracy (bugs, leaks, etc.). The main advantage of relying on a community is to capture the diversity of cases in which an applications works. Our fine-grained framework at code level is inspired by the coarse-grained solution Carat [13], which attempts to

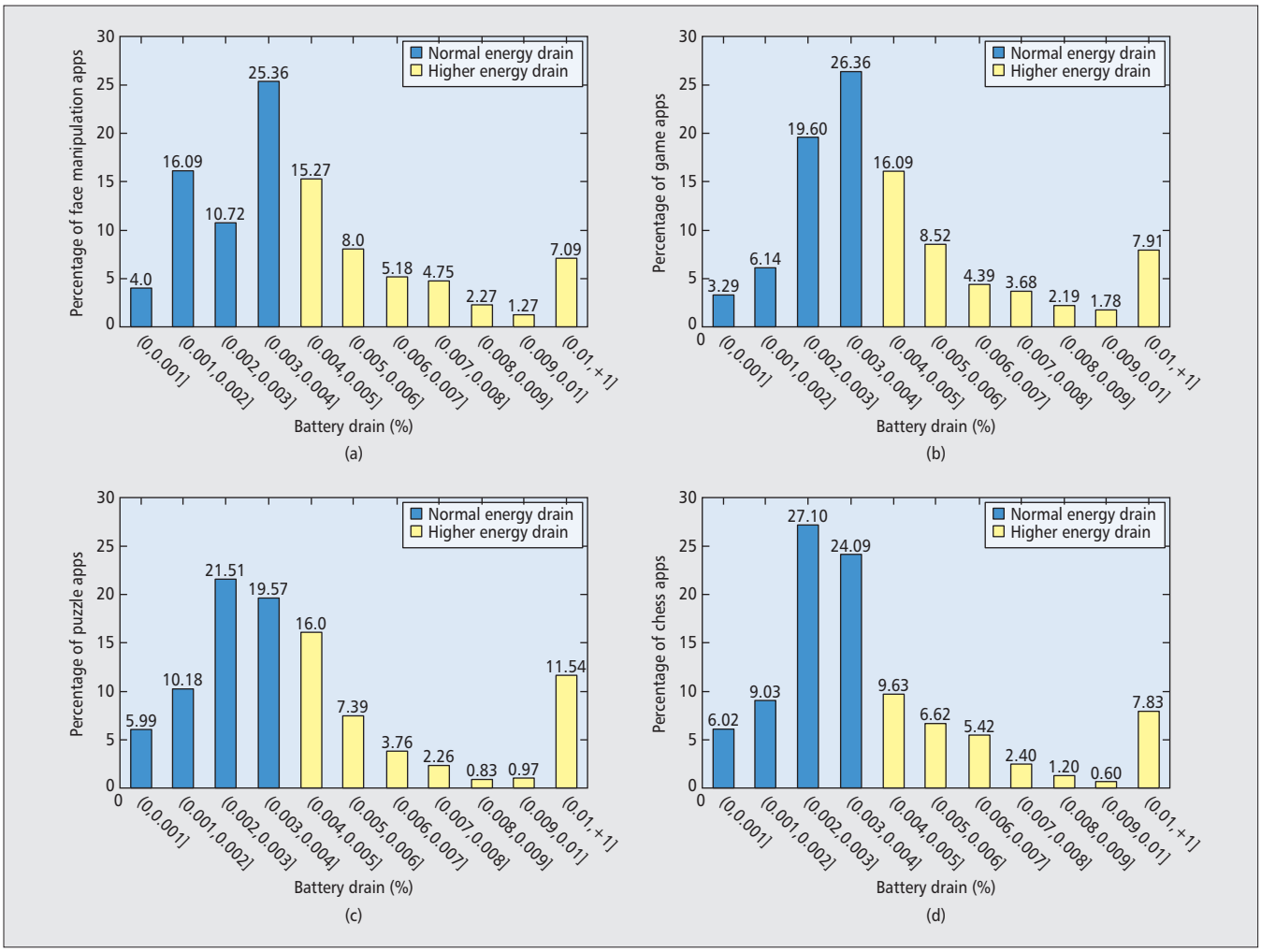

Figure 3. Smartphone apps that depict higher energy drain.

find energy bugs and hogs in mobile apps. Carat analyzes big repositories of data that depict the runtime behavior of a mobile app in order to find anomalies that can turn into customized recommendations for preferable configurations, (e.g., apps to kill) the mobile user can apply to make the battery life of his/her device longer

We believe that in the same way Carat detects energy anomalies, it is possible to determine the conditions or configurations for offloading smartphone applications. For instance, by applying the Carat method [13] over a subset of data (≈328,000 apps), we can get an idea about what is a resource-intensive app. Based on this data, which is collected in a real life deployment, we develop several case studies to motivate the viability and applicability of our approach. The subset contains for each app the expected percentage of energy drain. The Carat method consists of determining the energy drain distribution of a particular app, and then comparing it with the average energy drain distribution of other apps running in the device. The key insight of the method is to determine the possible overlap between application energy distributions in order to detect anomalies in application energy usage.

By analyzing apps' categories and specific purposes, we develop four case studies, which consist of face manipulation, games, puzzles, and chess. Figure 3 shows the results of the case studies. For each case, we extract a group from the subset of apps, where each app in a group is different from the rest. The four groups consist of about 550 face manipulation apps, about 7805 game apps, about 717 puzzle apps, and about 166 chess apps. Each group is compared to the average energy drain of the subset, which excludes its energy drain. From the results, we can observe that around 43.84 percent, 44.56 percent, 42.75 percent, and 33 percent of apps implement computational operations that require higher energy drain than normal, which is a significant number of apps. As consequence, computational offloading is required (e.g., customized alarm) to overcome the extra overhead introduced by the apps when showing resource-intensive behavior.

Generally speaking, higher granularity data is able to provide insights about the right conditions of these apps to offload to the cloud. However, beyond equipping the apps with computational offloading, this requires the mechanisms to be instrumentalized with the ability to record their own local/remote execution, such that we can capture more specific details about what induces the code to become resource-intensive and the runtime behavior of

the code in the device/surrogate. Thus, it is reasonable that the characterization of the computational offloading process can be modeled through a community of devices, such that by taking advantage of the huge amount of devices that connect to cloud, it can be possible to foster a more effective offloading strategy for smartphones.

Implicit crowdsourcing that does not need incentives, but rather is extrapolated from application usage, can be used to collect history traces of the computational offloading process across the entire system. Traces can be analyzed using cloud analysis features to extract the characterization. The purpose of the characterization is to define the effect of remote code execution in different conditions and configurations, where a condition depicts the interaction aspects of the user with the mobile (e.g., available memory and CPU, input variability), and a configuration represents the state of the components of the systems (bandwidth size, capacity of the cloud-surrogate, performance metrics of the mobile/back-end, etc.). In this manner, we can find out the most accurate configurations (what to offload) for a specific application in a particular device based on multiple criteria, such as type of surrogate (where to offload) and conditions of the system (when to offload). Additionally, it can be possible to determine offloading plans (how to offload) that enable the device to schedule code offloading operations (e.g., computational parallelization of code).

Furthermore, the characterization can also be utilized to identify reusable results. A reusable result is a portion of code that is commonly offloaded by multiple devices. These results can be cached in the cloud to respond to duplicate offloading requests from other devices. Logically, this accelerates the offloading process as the surrogate avoids the invocation time of the request. We envision that as part of the characterization process, pre-cached functionality from the entire mobile application can be requested on demand, as depicted in Fig. 2b. In this manner, our cloud assistance approach delivers a system in which the cloud is the expert, and mobile devices ask the cloud for its expertise.

Evaluation and Results

The binding between computational cloud services (Amazon EC2, Microsoft Azure, etc.) and smartphones is proven to be feasible with the latest technologies [5, 6, 9] (Android, Windows Phone, etc.), mostly because virtualization technologies and their synchronization primitives enable mobiles' transparent migration and remote invocation of code. In this section, we evaluate our ideas about equipping the offloading architectures with cloud assistance (analytics and stratified features) and support on demand (multi-tenancy). We conduct experiments using our own code offloading framework [10]. Our goal is to demonstrate the most significant insights about how code offloading is enhanced with data captured by a community of devices. In the evaluation, as clients we used a Samsung Galaxy S3 (i9300), Samsung Galaxy S2 (i9100), and Samsung Nexus (i9250), and as a surrogate; we used a nearby local computer (64-bit, 2.3 GHz Intel® Core™, 8 GB of memory), general-purpose servers (instances m1.small, m1.medium, m1.large, m1.xlarge), and a compute optimize instance (c3.2xlarge) from Amazon EC2.[1] To measure the energy consumed by the mobile in our offloading experiments we relied on the Mobile Device Power Monitor.[2]

Code Offloading Framework

Our framework enables the mobile applications to offload code at the method level using Java reflection. The framework follows a client-server model, where the client is located at the mobile and the server is located in the cloud. One mobile subscribes to one server, and a server can handle multiple devices. The framework is equipped with the mechanisms to record the offloading process at different levels of the architecture, such that offloading traces can be gathered from the devices that connect to the cloud. The mobile implements an automated mechanism that profiles code based on the information defined in a JSON schema. The schema is created at the cloud by analyzing the traces of the offloading process and defines the code to be offloaded from different points of view (e.g., *what*, *when*, *where*, and *how*). Each point describes the attributes a smartphone app must meet in order to offload; for instance, *what* defines the name of the candidate methods, *when* describes the informational thresholds the device must detect (e.g., latency), *where* defines the type of server in which the code has to be offloaded (e.g., server of type m1.small), and *how* introduces an execution plan for the code (e.g., parallelize the code into *n* processes). Additionally, other points of view can be introduced (e.g., user's location). However, this requires exploitation of code execution patterns from the community (e.g., pre-cache results based on location).

The characterization is sent to the mobile asynchronously using push notification technologies. Evidently, the use of crowdsourcing also implies solving the scalability problem of the cloud surrogate. Since its only purpose is to execute code, we think that the virtualization of the entire mobile platform in a server is unnecessary as it wastes CPU resources, and is counterproductive as it slows down performance. Thus, in our system, we avoid such virtualization approach, and instead rely on a low-level compiler, which was extracted from the mobile platform (Android) and built straight in the host operating system of the server. The compiler is built by downloading and compiling the source code of Android Open Source Project (AOSP)[3] over the server to target an x86 server architecture. In this manner, the framework takes advantage of the server at full capacity, which means that the surrogate is released from the limitations imposed by the mobile operating system, such as activating multiple instances from the same , and executing multiple applications concurrently, among others.

Furthermore, our framework implements an auto scaling mechanism that allows the server to scale horizontally in the cloud. Figure 4 depicts how the scaling process occurs. The framework

We envision that as a part of the characterization process, pre-cached functionality from the entire mobile application can be requested on demand. In this manner, our cloud assistance approach delivers a system where the cloud is the expert and mobile devices ask the cloud for its expertise.

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

[2] http://www.msoon.com/LabEquipment/PowerMonitor/

[3] http://source.android.com/

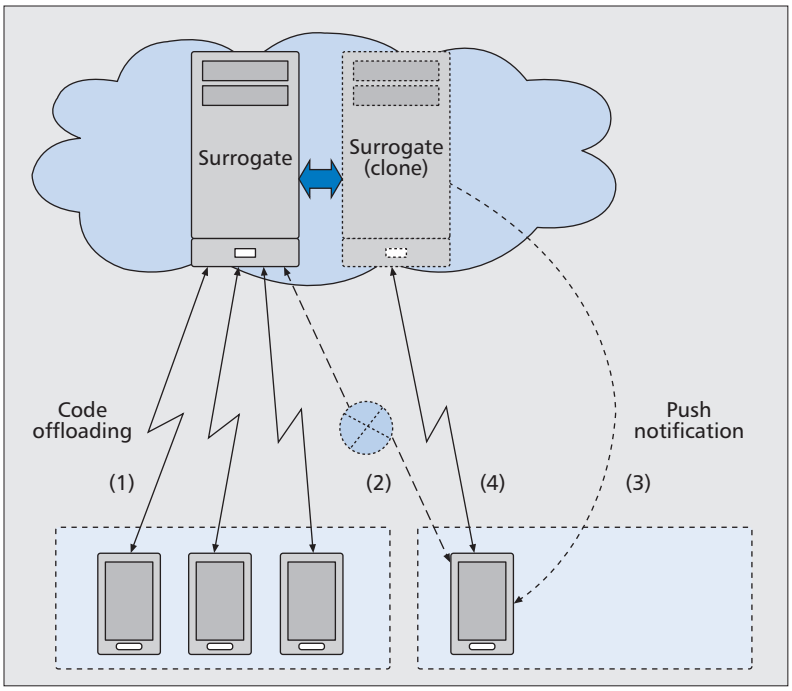

Figure 4. Representation of a cloud-based surrogate supporting multi-tenancy for code offloading.

implements a performance-based mechanism that monitors the utilization of CPU of the server, such that when a server is handling multiple offloading requests (1) and the CPU utilization is too high for method invocation (2), another server is created (3). In this process, the load of the subscribers is split between the available servers (4). The servers are in charge of updating the information of their subscribers via push notification.

Results

To demonstrate that the analytic properties of the cloud beyond service provisioning enhance the offloading process, we develop a use case based on the results of puzzle apps presented in a previous section and analyze the effects of offloading an NQueens algorithm from the smartphone to cloud. The aim of the algorithm is to calculate how to place n queens on an $n \times n$ chessboard. It uses backtracking with pruning to calculate all the solutions. The motivation of using this algorithm lies in its inherent characteristics to turn into a resource-intensive operation based on a normal user's interaction (play time). Definitely, we could rely on other uses cases, such as image or video processing (e.g., face recognition), to strongly emphasize the need for going cloud-aware. However, we believe that the applicability of code offloading beyond obvious use cases is possible, but it requires a priori knowledge about the possible executions of a mobile application. Of course, the knowledge should cover the effect of local and remote execution to assess multiple trade-offs.

Figure 5a shows the execution of NQueens with $n = 13$ in various cases, locally and remotely. Under the assumption that the same code can be offloaded multiple times by a smartphone app running in a particular or different devices, a code offloading request can be generic enough, such that its result can be reused in future invocations by other devices. In other words, the cloud can allocate an extended cache to store reusable results. We assumed that the NQueens is offloaded multiple times by a crowd of i9300, such that our framework identifies the request as a common offloading operation. A computational task offloaded to the cloud is serializable. Thus, it is possible to calculate for each request a MessageDigest key based on a standard checksum (e.g., a SHA-1 checksum) in order to uniquely identify each request. By collecting and clustering all the checksums of the requests using DBSCAN, it is able to find all the generic requests offloaded by a community of devices. By this approach, the result of a previous remote method invocation is pre-cached in the cloud. The diagram also shows the total offloading time of the pre-cached result. Based on the diagram, we can observe that the offloading process obtains further acceleration as the invocation time in the server is avoided. Arguably, the offloading result could be stored temporally in the cache of the device. However, the limited space of the cache in the device is unsuitable for long-term reuse. Clearly, the cache can be increased, but a cache that is too large can cause out-of-memory exceptions and leave the mobile application with little memory to work. In parallel, Fig. 5b shows the energy consumed by the mobile device in each of the previous cases. Since the acceleration of the offloading process influences the effort required by the mobile to execute the code, the faster the time to execute the code, the less computational effort is required to maintain an active communication channel to remote resources. Of course, this is certain when the code invocation requires a long execution time.

Finally, Fig. 5a also compares how the algorithm is processed by different smartphones and multiple servers of the cloud ecosystem in terms of response time. The diagram includes the communication latency, invocation, and synchronization time of the code. From the diagram, we can see that the latest mobile technologies are computationally comparable with some servers in Amazon's cloud such as m1.large and m1.xlarge. Moreover, we can visualize that the utility computing features of the cloud are critical in the decision of *where to offload*. While the latency in communication cannot be controlled, the total time of the invocation can decrease by adjusting the trade-off between utilization price and computational capabilities of the server. As a consequence, the characterization of an offloading operation is associated with multiple levels of enhancement. In this case, i9300 should offload smartly to m1.xlarge or c3.2xlarge to obtain maximum benefits. Clearly, the characterization is not obvious for the devices or the architecture in general. The characterization is a process computationally exhaustive and long-term for a single device, and to an even greater extent if we consider specific mobile applications. Thus, crowdsourcing can be utilized to capture information that can be transformed into knowledge to assist in the offloading process.

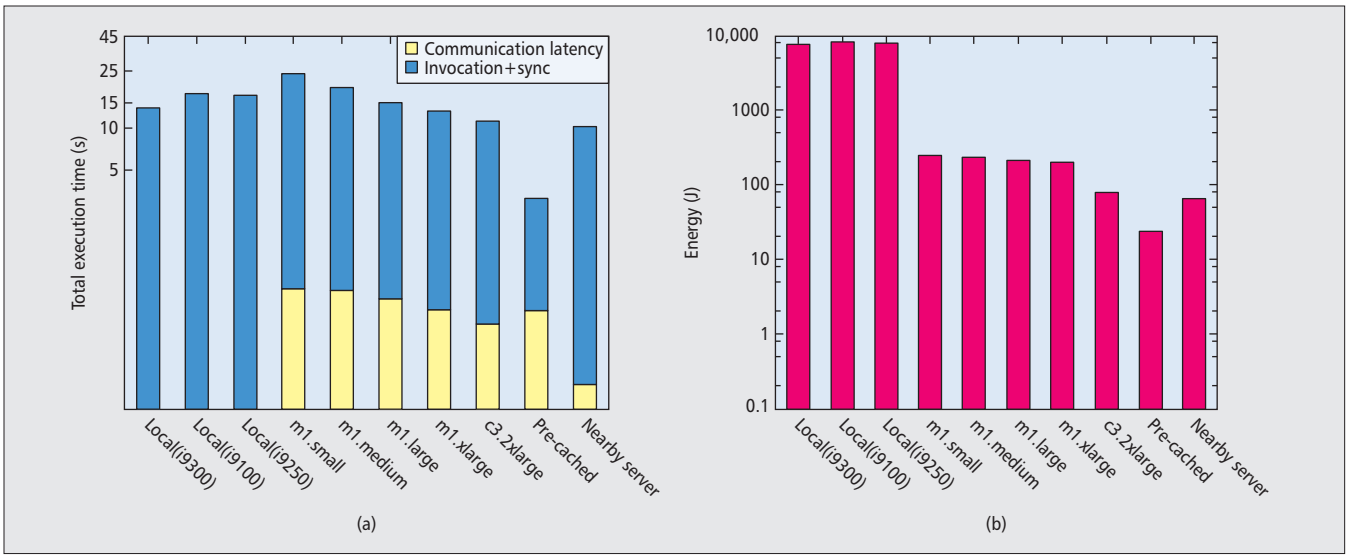

Figure 5. a) Acceleration gains that can be achieved by caching offloading results in terms of response time; b) energy consumed in each offloading case (local, remote, and pre-cached).

Discussion

As with any other crowdsourcing solution, our strategy becomes gradually more effective as data is being collected to understand the offloading process for a particular mobile application. However, initially, the advantages of our framework lie in the ability to consider many parameters from different perspectives in the inference process that determines when to offload. Once cloud assistance is available, the advantages of our framework take place, such that a computational request can speed up the performance of a smartphone app up to 10×, and up to 30× using pre-cached functionality. This acceleration is variable as it is proportional to the actual time required to execute the computational task. We believe that the assistance in the offloading process can be improved further by analyzing different aspects of the architecture at multiple levels of granularity [13]. The benchmarking (support for multi-tenancy) of the framework is left as future work since it cannot be shown with simple insights, but instead requires large-scale detailed analysis.

Conclusions and Future Directions

The bridging of the mobile and the cloud has led to rethinking the binding mechanisms that enable the cloud to be exploited for the benefit of the handset. Code offloading is a key technique in augmenting the computational capabilities available for mobile applications with elastic cloud resources. However, the sustainability of the technique in practice is an open issue. In this article, we explore the challenges for code offloading from a systemic point of view and identify the key limitations that prevent the adoption of code offloading. We evaluate a number of proposed design strategies to overcome these limitations by using our own code offloading framework. The source code used in the experiments is available as open source in GitHub.[4] Our study highlights new directions for the design of future mobile architectures supported by cloud computing. Essentially, our work proposes the use of data analytics over offloading history captured by a community of devices in order to enrich the context of the device to offload to the cloud without inducing a counterproductive outcome for the mobile. Moreover, data analytics can also empower the offloading architectures with adaptive features that respond to the behavior of code during runtime, such as quality of experience based on code acceleration in different cloud servers, pre-cache functionality of the apps, auto scaling according to code execution, and so on.

Acknowledgment

We want to thank to Eemil Lagerspetz for his valuable assistance and suggestions. The authors also thank the anonymous reviewers for their insightful comments. This research was supported by European Social Fund's Doctoral Studies and Internationalization Programme DoRa, which is carried out by Foundation Archimedes.

References

[1] M. Satyanarayanan *et al.*, "The Case for Vm-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, 2009, , pp. 14–23.
[2] M. V. Barbera *et al.*, "Mobile Offloading in the Wild: Findings and Lessons Learned through a Real-Life Experiment with a New Cloud-Aware System," *Proc. IEEE INFOCOM 2014*, Toronto, Canada, Apr. 27–May 2, 2014.
[3] P. Bahl *et al.*, "Advancing the State of Mobile Cloud Computing," *Proc. ACM MobiSys Workshop 2012*, Low Wood Bay, Lake District, U.K., June 25–29, 2012.
[4] K. Kumar and Y.-H. Lu, "Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?," *Comp. Mag.* vol. 43, no. 4, 2010, pp. 51–56.
[5] E. Cuervo *et al.*, "Maui: Making Smartphones Last Longer with Code Offload," *Proc. ACM MobiSys 2010*, San Francisco, CA, June 15–18, 2010.
[6] B.-G. Chun *et al.*, "Clonecloud: Elastic Execution between Mobile Device and Cloud," *Proc. ACM EuroSys 2011*, Salzburg, Austria, Apr. 10–13, 2011.
[7] H. Flores and S. Srirama, "Mobile Code Offloading: Should It Be a Local Decision or Global Inference?," *Proc. ACM MobiSys 2013*, Taipei, Taiwan, June 25–28, 2013.

[4] https://github.com/huberflores/NQueensCodeOffloading

[8] S. Kosta *et al.*, "Thinkair: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading," *Proc. IEEE INFOCOM*, Orlando, FL, Mar. 25–30, 2012.
[9] M. S. Gordon *et al.*, "Comet: Code Offload by Migrating Execution Transparently," *Proc. USENIX 2012*, Boston, MA, June 13–15, 2012.
[10] H. Flores and S. Srirama, "Adaptive Code Offloading for Mobile Cloud Applications: Exploiting Fuzzy Sets and Evidence-Based Learning," *Proc. ACM MobiSys Wksp. 2013*, Taipei, Taiwan, June 25–28, 2013.
[11] C. Shi *et al.*, "Cosmos: Computation Offloading as a Service for Mobile Devices," *Proc. ACM MobiHoc 2014*, Philadelphia, PA, Aug. 11–14, 2014.
[12] H. Flores and S. N. Srirama, "Mobile cloud Middleware," *J. Systems and Software*, vol. 92, 2014, pp. 82–94.
[13] A. J. Oliner *et al.*, "Carat: Collaborative Energy Diagnosis for Mobile Devices," *Proc. ACM Sensys 2013*, Rome, Italy, Nov. 11–14, 2013.

BIOGRAPHIES

HUBER FLORES [S] (huber@ut.ee) is currently a Ph.D student at the Faculty of Mathematics and Computer Science, University of Tartu. He received his B.Eng. in computer science from the University of San Carlos of Guatemala and his M.Sc. in software engineering from a combined program between the University of Tartu and Tallinn University of Technology, Estonia. He is also a Student Member of ACM (SIGMOBILE). His major research interests include mobile offloading, mobile middleware architectures, and mobile cloud computing.

PAN HUI (panhui@cse.ust.hk) received his Ph.D. degree from the Computer Laboratory, University of Cambridge, and earned both his M.Phil. and B.Eng. from the Department of Electrical and Electronic Engineering, University of Hong Kong. He is currently a faculty member of the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology, where he directs the HKUST-DT System and Media Lab. He also serves as a Distinguished Scientist of Telekom Innovation Laboratories (T-labs) Germany and an adjunct professor of social computing and networking at Aalto University, Finland. Before returning to Hong Kong, he spent several years in T-labs and Intel Research Cambridge. He has published around 150 research papers, and has some granted and pending European patents. He has founded and chaired several IEEE/ACM conferences/workshops, and has served on the Organizing and Technical Program Committees of numerous international conferences and workshops including ACM SIGCOMM, IEEE INFOCOM, ICNP, SECON, MASS, GLOBECOM, WCNC, ITC, ICWSM, and WWW. He is an Associate Editor for *IEEE Transactions on Mobile Computing* and *IEEE Transactions on Cloud Computing*.

SASU TARKOMA [M'06, SM'12] (sasu.tarkoma@helsinki.fi) received an M.Sc. degree in 2001 and a Ph.D. degree in 2006 in computer science from the University of Helsinki. Since 2009 he has been a full professor of computer science at the University of Helsinki. He has led and participated in national and international research projects at the University of Helsinki, Aalto University, and Helsinki Institute for Information Technology (HIIT). He has worked in the IT industry as a consultant and chief system architect as well as principal researcher and laboratory expert at Nokia Research Center. He has over 140 scientific publications, four books, and four U.S. patents.

YONG LI [M'09] (liyong07@tsinghua.edu.cn) received his B.S. degree in electronics and information engineering from Huazhong University of Science and Technology, Wuhan, China, in 2007 and his Ph.D. degree in electronic engineering from Tsinghua University, Beijing, China, in 2012. During July to August 2012 and 2013, he was a visiting research associate with Telekom Innovation Laboratories and the Hong Kong University of Science and Technology, respectively. During December 2013 to March 2014, he was a visiting scientist with the University of Miami, Coral Gables, Florida. He is currently a faculty member in the Department of Electronic Engineering, Tsinghua University. His research interests are in the areas of networking and communications, including mobile opportunistic networks, device-to-device communication, software-defined networks, network virtualization, and future Internet. He is currently an Associate Editor of the *EURASIP Journal on Wireless Communications and Networking*.

SATISH SRIRAMA (srirama@ut.ee) is an associate professor and head of the Mobile & Cloud Lab at the Institute of Computer Science, University of Tartu. He received his Ph.D in computer science and Master's in software systems engineering from RWTH Aachen University, Germany, and his Bachelor's degree (BTech) in computer science and systems engineering from Andhra University, India. His current research focuses on cloud computing, mobile web services, mobile cloud, Internet of Things, and migrating scientific computing and enterprise applications to the cloud.

RAJKUMAR BUYYA (rbuyya@unimelb.edu.au) is a professor of computer science and software engineering and director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is the founding CEO of Manjrasoft, a spin-off company of the university, commercializing its innovations in cloud computing. He has authored over 430 publications and four textbooks. He has also edited several books, including *Cloud Computing: Principles and Paradigms* (Wiley, 2011). He is one of the most highly cited authors in computer science and software engineering worldwide (h-index = 66 and 21,300+ citations). Software technologies for grid and cloud computing developed under his leadership have gained rapid acceptance and are in use at several academic institutions and commercial enterprises in 40 countries around the world. He has led the establishment and development of key community activities, including serving as founding Chair of the IEEE Technical Committee on Scalable Computing and five IEEE/ACM conferences. These contributions and his international research leadership have been recognized through the 2009 IEEE Medal for Excellence in Scalable Computing award. Manjrasoft's Aneka Cloud technology developed under his leadership has received the 2010 Asia Pacific Frost & Sullivan New Product Innovation Award and the 2011 Telstra Innovation Challenge, People's Choice Award. He is currently serving as the first Editor-in-Chief of *IEEE Transactions on Cloud Computing*.