Automated Selection of Offloadable Tasks for Mobile Computation Offloading in Edge Computing

Alessandro Zanni*§, Se-young Yu§†, Paolo Bellavista*, Rami Langar‡ and Stefano Secci§
*Dept. Computer Science and Engineering (DISI), University of Bologna, Italy
§LIP6, Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, France
†International Center for Advanced Internet Research (iCAIR), Northwestern University, USA
‡LIGM, University Paris Est Marne-la-Vallee, UMR 8049, France
Email: {alessandro.zanni3,paolo.bellavista}@unibo.it, young.yu@northwestern.edu, rami.langar@u-pem.fr, stefano.secci@upmc.fr

***Abstract*—Mobile computation offloading has recently attracted much interest and first offloading solutions have been developed. However, the relevant technical challenge of how to automatically determine offloadable sections of Android applications has not been adequately investigated so far. This paper proposes an innovative task selection algorithm that can parse an Android application autonomously and classify all the methods based on their offloadability by adopting a fine-grained and multi-steps analyzer. The reported experimental results show the effectiveness of our solution when applied to the top 25 most downloaded Android apps on the Google Play store, by showing its accuracy in identifying offloadable methods and demonstrating the potential benefits of automated mobile computation offloading.**

I. Introduction

Mobile applications are constantly growing in terms of their functionality and complexity in order to offer a wider set of high-level features that challenge execution time and energy usage. Recent mobile applications such as speech recognition, natural language processing, computer vision and augmented reality applications may require processing power beyond which mobile devices are equipped with or can drain their battery life at an unacceptable pace.

Mobile computation offloading [1] allows dynamically migrating computation-intensive applications from resource-limited devices to external machines; it can reduce execution time and energy consumption at the cost of transferring computation-related information. In particular, it is being recognized that the offloading decision should be context-dependent, i.e., deciding when and which computation task to offload depending on current context data such as execution time, energy consumption, task complexity, expected network latency, etc. In addition, modern mobile computation offloading technologies should dynamically consider the opportunities associated with fog or mobile edge nodes, in addition to the ones targeted by traditional mobile computation offloading, in order to go beyond the sole direct interaction of mobile devices and global cloud datacenters [2]. In fact, fog computing [3] and Mobile Edge Computing (MEC) [4] can provide virtualized computation/storage resources in the proximity of data sources and targeted devices, with the potential ability to overcome cloud limitations, by exploiting geographically distributed heterogeneous resources with mobility support [2]. They allow mobile devices to be served from closer servers to offload part of the mobile applications computation with lower delay. Mobile applications can benefit from computation capability of fog and MEC nodes to remain available and responsive while processing large volumes of data [5].

One of the main, still open, technical challenges related to mobile offloading is how to automatically partition an application code into offloadable and non-offloadable parts, for arbitrary applications by learning application code architecture [6]. Performing this partitioning manually requires significant human effort: portions of Android applications (tasks) may be unsuitable for offloading due to several non-trivial motivations, such as frequent interaction with mobile users, difficulties in replicating device-specific resource instances at cloud/edge nodes, impossibility to serialize used resources, only to mention a few. Moreover, this required human effort slows down the acceptance of mobile offloading techniques in industrial scenarios for obvious cost motivations.

Therefore, to maximize the benefits of offloading, a sophisticated and autonomous task selection algorithm is required. Developing a general-purpose task selection algorithm is not trivial because it requires detecting offloadable parts of tasks from the entire Android application code without prior knowledge of the analyzed application. This paper specifically presents the design, implementation, and evaluation of a novel fine-grained task selection algorithm that can autonomously scan a generic Android application without any application-specific prior knowledge. Our algorithm analyzes and dynamically partitions the application into offloadable and non-offloadable methods. The granularity of our offloading suitability check is made at the method level and achieved with an analysis of the application structure of classes/methods to ensure that the selected methods will be executed on the edge/cloud side consistently. Finally, our solution has been experimentally evaluated on the top 25 most downloaded Android applications from the online Google marketplace to prove the efficiency of our automated task selection algorithm in terms of the amount of detected methods and associated performance. To the best of our knowledge, this is the first proposal of an automated task selection algorithm for mobile computation offloading.

II. Related Work

MAUI [7] proposes code offloading using a profiler that measures energy consumption and a solver which decides to offload a method based on those measurements. ThinkAir [8] performs on-demand resource allocation, executing methods in parallel by dynamically creating, resuming, and destroying multiple VMs in the cloud. COMET [9] uses thread offloading among distributed devices to augment mobile devices with machines available in the network. Cuckoo [10] implements a communication library between mobile devices and the Ibis communication middleware, by offloading bundles that contain compiled code. The ULOOF framework [11], [12] introduces an improved offloading decision mechanism that exploits the assessment of the available bandwidth as well as energy consumption, thus providing realistic execution time and energy consumption estimations.

The above proposals have tried to address computation offloading in different ways and at different granularity levels, i.e., method, thread, or component levels. They focus on offloading execution rather than selecting which tasks to offload; they generally require developers to add annotations to indicate which portion of an application to offload, that is to modify application code manually. In fact, as already stated, code selection decision is not well explored yet in the related literature. Authors in [6] underline the task selection algorithm as the main issue in offloading; moreover, they highlight the granularity selection and the dynamic application partition as the main technical challenges to face in order to offload to edge/cloud nodes. In this perspective, CloneCloud [13] uses a static code analyzer to automatically mark possible migration points and to partition the binary of an application with a set of execution point. CloneCloud uses a thread-granularity for the offloading execution selection and each execution point decides between where the application migrates the thread towards the cloud or the local execution on the device.

Our solution, differently from the above work, provides a fully autonomous code selection mechanism with method-level granularity. It can dynamically analyze every kind of Android application and detect the list of methods suitable to be offloaded. Only the CloneCloud implements a similar autonomous code selection mechanism without the need of hard-coding into the specific application. However, it specifically targets a powerful remote platform, i.e., the cloud computing, to execute the offloaded code, since it needs to re-create a VM with the same hardware and OS used locally. On the contrary, we focus on a decoupled and lightweight solution that has no platform constraints and can be used in many remote servers independently from the available resources. In fact, our solution aims to offload code at the method level that can run on every server equipped with a Java Virtual Machine without the same hardware/OS of the mobile device, thus suitable also for less-powerful platforms, such as in the case of envisioned MEC nodes.

III. Architecture and Implementation Insights of our Task Selection Solution

Our autonomous selection algorithm takes any Android application package as an input and detects the included methods that are suitable to be dynamically offloaded. The proposed solution is designed to be general-purpose and independent from application type, domain, size, and internal structure. In particular, our algorithm starts by scanning the whole application structure and apply incremental checks on each single class/method and dependency constraints between classes/methods.

To minimize the needed computation while achieving an overall accurate result, we consider the following criteria when we scan the package. Table III lists the criteria that our algorithm considers for classes and methods.

Type of Checks	Classes	Methods
Internal Usage	✓	✓
Class Offloadability	✓	✓
Internal Objects Calls	✗	✓
Internal Methods Calls	✗	✓

TABLE I
Checks automatically performed by the algorithm

It is worth noting that we classify a device-dependent class that includes objects impossible to be available/migrated at/to remote cloud/edge nodes as non-offloadable, e.g., native Android libraries or non-serializable Java objects (i.e., Threads). All methods that are included into a non-offloadable class are classified as non-offloadable. Vice versa, an offloadable class is a class that successively passes all the class checks and whose methods will be considered as possible offloading candidates (after method-level checks). In fact, only after a method passes all the incremental checks in the table, we mark it as offloadable.

Figure 1 shows the components of our task selection algorithm and, in the following sections, we give implementation insights about each step of the method selection algorithm.

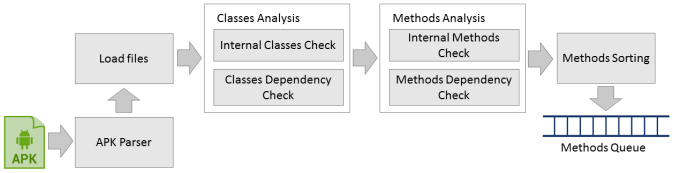

Fig. 1. Methods Selection Algorithm Architecture

A. APK Parser and Configuration File Loading

The APK parser retrieves an Android manifest file from a provided APK to find a relevant information about the internal application structure: it lists app packages and the permissions to access protected parts of the API; it also contains information about app components, which include activities, services, broadcast receivers, and content providers; for each component, it indicates the class that implements the component and its capabilities (e.g., how it can be launched).

Successively, our solution retrieves, parses, and manipulates class file methods, by leveraging on the Soot framework [14]. Soot provides a set of Java APIs to modify bytecode in APK packages; in particular, Jimple [15] is an intermediate representation of the Java bytecode that we use to implement and optimize modifications of existing bytecode.

After that, we apply a list of keywords to filter out components that cannot be offloaded based on the nature of the component, such as GUI and hardware sensors. The complete list of keywords and its rationale are not reported here for the sake of briefness.

B. Class Analysis

Our class analyzer uses a set of tests to detect which classes are not suitable to be offloaded, as follows.

- **Internal Class Test**: We found it is impossible to offload anonymous inner classes because they are not associated with the class name they are defined in the Java bytecode and the Java compiler cannot locate the class with the specific class name in the app code. However, our algorithm allows offloading non-anonymous inner classes that are identified with the name "class_name+$+inner_class_name".
- **Android Class Test**: We inspect classes that may contain device-specific information by looking at their path of the package the classes are belong to. This step is not mandatory from a functional point of view because all Android classes can be detected by the following method analysis, but it allows to detect non-offloadable classes more efficiently and, thus, to improve the performance of our algorithm.
- **Superclass and Dependency Tests**: For each non-internal class, we check if it extends an already defined non-offloadable class. A subclass of an non-offloadable superclass is likely to call non-offloadable methods or access variables that should not be accessed from the remote machine, therefore we classify them as non-offloadable.

C. Method Analysis

The algorithm proposed to check method offloadability mainly consists of two parts: (i) the main control that checks the suitability of each method to be offloaded and (ii) a dependency check that analyzes dependencies among methods. The main control phase performs the following tests:

- **Internal Method test**. We check the suitability of each method to be offloaded by using two criteria. (i) If the method represents a static initializer (<clinit> method name) used to initialize the class object itself, it is not offloaded because a static method is always added by the Java compiler and called by JVM after class loading. (ii) If the method contains the MainActivity among its input parameters, it is used internally by the Android compiler to manage the MainActivity class or to use the context of the MainActivity for the application startup.
- **Class test**. If a method belongs to a class marked as non-offloadable, it is classified as non-offloadable as well because it is likely to access non-offloadable objects, methods or variables of the class.
- **Objects Calls test**. If a method contains platform-dependent calls towards objects that were marked as non-offloadable during class analysis, it is classified as non-offloadable.
- **Keywords test**. We check if the methods include platform-dependent calls towards methods that cannot be offloaded. These methods are in the keyword-based blacklist loaded from configuration files.

In addition to the previous checks, we also have dependency check phase where we check which methods are invoked while each inspected method executes. For each method, we can determine its dependencies and discard the ones having at least one method marked as non-offloadable. To this purpose, we check the body of each method to see if there are any calls to methods that are already classified as non-offloadable.

In particular, to scan dependencies of methods with minimum overhead, we build a directed graph where each node is a method and each edge is a method call directed from the caller method (parent node) towards the callee method (child node). Each node contains the following information: (i) method signature; (ii) offloadable status, to indicate if the method is offloadable, non-offloadable or temporally unknown; (iii) visited status, which indicates if the dependency check has already been executed for that node; (iv) parents list that contains the list of the caller methods. The algorithm used to create, modify, and parse the graph is described below:

- **Step 1.** We parse each method found in the offloadable classes to build a directed graph. In each method invocation from the method body, we create an edge from child to parent node.
- **Step 2.** Once the graph is created, we parse it to detect which methods depend on non-offloadable methods (at least one child node is non-offloadable). Starting from the non-offloadable methods found during method checks, we iterate through their parents until we reach to the root and set the scanned nodes as non-offloadable. In this way, we can recursively set as non-offloadable all the interested branches without scanning all nodes.
- **Step 3.** We retrieve all the methods that are marked as offloadable from the graph.

It is worth noting that the same dependency check can be performed through an iterative approach where we check the internal offloadable methods calls, with a complexity of $\Theta(n^2 * \log n)$, where n is the total number of methods. Since the target application may have a large number of methods, the iterative approach has demonstrated to be too slow and therefore the graph approach is required. In this case, the time complexity for the dependency checks using our graph-based approach is $\Theta(n)$, where n is limited to the number of non-offloadable methods.

IV. Experimental Evaluation

We have thoroughly evaluated the performance of our proposed task selection algorithm by applying it to the top-25 most popular Android applications from the Google Play marketplace (according to the ranking of Feb. 1st, 2017). Those applications compose an heterogeneous set of applications that are different in terms of goals, domains, and internal structure. In our experiments, besides the ratio and type of offloadable and non-offloadable methods, we analyze also our algorithm performance in terms of time to analyze apps and to retrieve the list of offloadable methods.

Let us note that we fully tested our task selection algorithm combined with an external computation offloading mechanism, i.e., ULOOF [16], [12]. We set-up a complete offloading platform that autonomously creates new Android applications with the ability to scan dynamically their associated codes, modify them, and offload some methods on an edge/cloud node, if necessary, without any human intervention or specific configuration. Additional details about how we use the ULOOF framework are available at [17].

A. Method Selection Assessment

The first significant figures relate to the number of methods included in the top-25 most popular Android apps, how many of them are offloadable, and the motivations why they are considered non-offloadable, as listed in Section III-C.

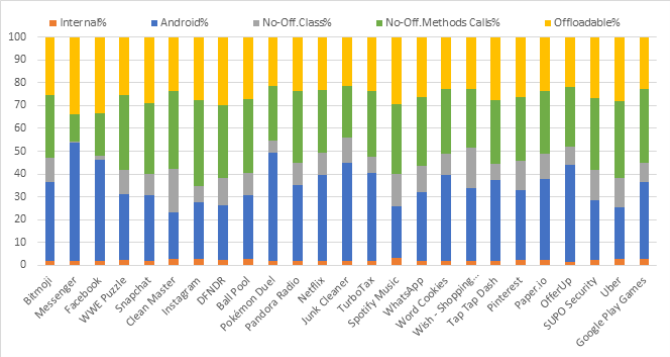

Fig. 2. Methods analysis

As depicted in Figure 2, the average and median ratio of methods suitable to be offloaded is 24.0%, with a standard deviation of 3.0%. Some popular apps, such as Messenger, Facebook, and Uber, show even a higher ratio, almost 30%.

B. Task Selection Performance

To measure the performance of our task selection algorithm, for each app, we compare the total number of methods found and the time needed by our algorithm to scan the app and to apply the proposed heuristics (Figure 3).

The total time needed to scan an app is the sum of the times needed for classes and methods. Due to the limited number of classes usually present into an app if compared with the number of methods, the latency associated with classes has demonstrated to be almost negligible (always less than 5% of

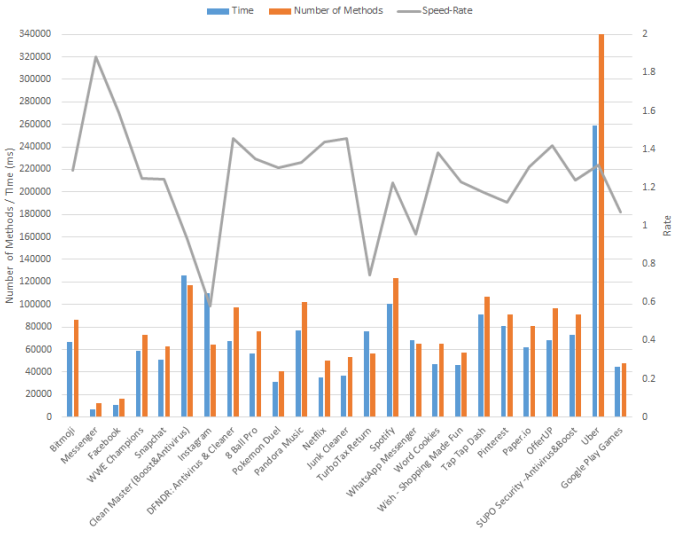

Fig. 3. Task Selection Performance

total time) and the overall performance is mainly determined by the time needed to scan methods. Note that our algorithm exhibits very good performance and applicability to real-world scenarios, with its ability to scan the vast majority of existing top-25 apps in less than 1 minute.

For each app, we also show the speed-rate, defined as the number of methods evaluated per ms. As shown in Figure 3, it ranges from 0.6 to 1.9, with an average rate of 1.3 methods scanned per ms.

V. Conclusive Remarks

This paper proposes an innovative task selection algorithm that autonomously parses a mobile Android app and retrieves the list of methods that are suitable to be offloaded to a cloud/edge node. The reported experimental results show that our algorithm can identify up to almost 30% of methods that are suitable candidates for offloading when applied to most popular top-25 apps on the Google Play marketplace. In addition, our algorithm has demonstrated to be able to perform a complete scan on large-scale real-world apps within tens of seconds. These encouraging results are stimulating our ongoing research work in the field: among the others, we are extending our experimentation to measure latency and energy savings when applying the proposed solution, integrated with ULOOF, to the same set of top-25 Android apps.

Acknowledgment

This work was partially supported by the FUI PODIUM project (Grant no. 15016552) and the ANR ABCD project (contract nb ANR-INFRA-13-005).

References

[1] K. Kumar and Yung-Hsiang Lu, “Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?” *Computer*, vol. 43, no. 4, pp. 51–56, Apr. 2010.

[2] P. Bellavista, A. Corradi, A. Zanni, “Integrating Mobile Internet of Things and Cloud Computing towards Scalability: Lessons Learned from Existing Fog Computing Architectures and Solutions,” ser. 3rd International IBM Cloud Academy Conference (ICA CON) ’15, 2015.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things.” ACM Press, 2012, p. 13.

[4] Beck, M. T., Werner, M., Feld, S., and Schimper, T. (2014). Mobile edge computing: A taxonomy, in *Proc. of AFIN 2014*.

[5] A. V. Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, “MEC-ConPaaS: An experimental single-board based mobile edge cloud,” Apr. 2017.

[6] S. Yi, C. Li, and Q. Li, “A Survey of Fog Computing: Concepts, Applications and Issues,” in *Proc. of ACM Mobidata 2015*.

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: Making Smartphones Last Longer with Code Offload,” in *Proc. of ACM MobiSys 2010*.

[8] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, Thinkair: Dynamic resource allo- cation and parallel execution in the cloud for mobile code offloading, in INFOCOM, 2012 Proceedings IEEE, pp. 945953, IEEE.

[9] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, “COMET: Code Offload by Migrating Execution Transparently,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 93–106.

[10] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, “Cuckoo: A Computation Offloading Framework for Smartphones,” in *Mobile Computing, Applications, and Services*. Springer, Berlin, Heidelberg, Oct. 2010, pp. 59–79.

[11] J. L. D. Neto, D. F. Macedo, and J. M. S. Nogueira, “Location aware decision engine to offload mobile computation to the cloud,” in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, Apr. 2016, pp. 543–549.

[12] J. L. D. Neto, S. Yu, D. Macedo, J. M. S. Nogueira, R. Langar, S. Secci, “ULOOF: a User Level Online Offloading Framework for Mobile Edge Computing,”. 2017. HAL research report nb. hal-01547036.

[13] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “CloneCloud: Elastic Execution Between Mobile Device and Cloud,” in *Proc. of ACM EuroSys 2011*.

[14] “A framework for analyzing and transforming Java and Android Applications.” [Online]. Available: https://sable.github.io/soot/

[15] R. Vallee-Rai and L. J. Hendren, *Jimple: Simplifying Java Bytecode for Analyses and Transformations*, 1998.

[16] A. Zanni, S. Yu, S. Secci, R. Langar, P. Bellavista, D.F. Macedo, “Automated Offloading of Android Applications for Computation/Energy Optimizations,” in *Proc. of IEEE INFOCOM 2017*.

[17] ULOOF project (website): https://uloof.lip6.fr.

[18] “dex2jar.” [Online]. Available: https://github.com/pxb1988/dex2jar