Automated Offloading of Android Applications for Computation/Energy Optimizations

Alessandro Zanni*, Se-young Yu†, Stefano Secci†, Rami Langar‡, Paolo Bellavista* and Daniel F. Macedo§
*Dept. Computer Science and Engineering (DISI), University of Bologna, Italy
Email: {alessandro.zanni3,paolo.bellavista}@unibo.it
†LIP6,Sorbonne Universits, UPMC Univ Paris 06, UMR 7606, France
Email: {young.yu, stefano.secci}@lip6.fr
‡LIGM,University Paris Est Marne-la-Vallee, UMR 8049, France
Email: rami.langar@u-pem.fr
§Computer Science Department, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
Email: damacedo@dcc.ufmg.br

***Abstract*—Our work presents a methodology and a tool to optimize Android applications using mobile computation offloading techniques. Our demo provides a study about the compatibility of being offloaded of Android methods and a methodology to autonomously classify Android methods based on their functions.**

I. Introduction

Computation offloading is widely accepted as a powerful concept that can overcome the resource constraints of mobile devices and low-power Internet-of-Things devices, by dynamically sending computation-intensive tasks to remote servers depending on context. In different applications and deployments, remote servers may vary from globally available cloud resources to fog/edge gateway nodes in the vicinity. Many proposed a mobile computation to offload computation from a resource limited device to another device [1]. This can help mobile devices to reduce execution time and energy consumption of the computation at the cost of transferring (parts of) the computation and related information to the offload target and of receiving results from there.

This demo presents a tool that, given a compiled APK (in an Android environment), can create a novel APK file that contains additional code to automate the runtime offloading of a program. The methods enabled to be offloaded at runtime are chosen depending on computing time and energy usage, which are predicted in our framework. This is done, while take into consideration the complexity of the arguments and previous execution data gathered [2].

We present the implementation and evaluation of an offloading tool that can autonomously scan a generic Android mobile application, without any prior and application-specific knowledge, to autonomously create a new version of the application, which is functionally equivalent, but with the capability to offload computations to a remote server. The granularity of the offloading decision is the application method. It is achieved this by a careful automated analysis to detect which methods can be executed on the server consistently and by marginally modifying those methods code to run them remotely in a completely transparent way. The result is then integrated into a new Android application that runs some methods locally on the mobile device and others remotely on the server.

To achieve this goal, we have improved and extended an existing prototype from [2] in two main directions:

- Autonomous method selection. The tool includes an algorithm to autonomously select the methods suitable to be executed both locally and on the remote server, through a complete scan of the: i) APK file; ii) the classes developed by the programmers; iii) the methods for each class. The selected methods are then sorted in terms of the number of offloadable method calls.
- Translate methods and execution on a remote server. Our extension has added the ability to offload every suitable method whatever: i) its input parameters, by extending the methods translation code generation to support complex objects including lists and arraylists; ii) its state, being either static or non-static method; iii) the static or no-static variables used internally in its body with the related object management.

II. Autonomous Method Selection

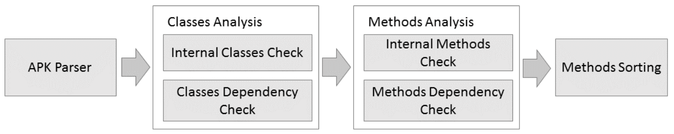

Fig. 1. methods checking analysis inside the method selection algorithm

Figure 1 illustrate the procedure we used in our analysis.

A. APK Parser

Firstly, the parser retrieves the information from the application, e.g., the application structure and the files created by the developers. The parser analyzes the APK file by scanning and parsing the Android Manifest file in the APK to find the MainActivity class.

B. Classes and Methods Analysis

The classes analysis phase aims to minimize the number of offloadable classes from the classes found from APK Parser, using the Soot framework [3] which allows to modify java and android application without their source. Initially, our internal class check detects classes that cannot be offloaded since they extend device-dependent classes, i.e., Android library, Threads, etc. After that, it checks the dependencies of each class whether they contain any of the non-offloadable components iteratively, through multiple loops until no new non-offloadable classes are found.

For each possible offloadable class, our tool parses the body of each method in a fine-grained way. The tool first determines whether the methods are internal or device-dependent using the following criteria:

1) Internal check: if the method is a default constructer generated from the compiler or contains the MainActivity among its parameters, the tool will discard it from the offloadable classes because they are likely have a small computation which makes the offloading inefficient.
2) Class check: if the method belongs to a class marked as not-offloadable, the method cannot be offloaded because it is likely to call not-offloadable objects/methods/parameters.
3) Native keywords check: the tool pre-loads five configuration files with blacklist keywords indicating elements that it is convenient not to contain into an offlodable method, i.e., application lifecycle management, application GUI management, application events management, application in/out management; native Android libraries.

C. Methods Sorting

Finally, the tool uses a sorting algorithm to determine the order the methods. We introduce a priority queue where the methods are added with a priority coefficient (Wm) related to the maximum weight among the internal offloadable methods ones (W_i), as explained in the equation 1.

$$Wm = \begin{cases} 1, & if\ no\ internal\ method \\ max(W_i) + 1, & otherwise \end{cases} \quad (1)$$

then using the following algorithm:

- Step 1 (priority 1): Search the methods that do not contain any calls to methods declared offloadable in their body, and put the method in the priority queue.
- Step N (priority N): Search the methods that contains calls to offloadable method with at maximum =(N-1) in their body and insert in the priority queue.

This results a queue with methods sorted in a decreasing order in terms of their weight, where the weight represents the total number of offloading methods being called by that method and by the methods being called from that method.

III. Method Translation and Optimization

Our post-compiler can optimize every method detected from the previous method selection algorithm into a Jimple pseudocode. The optimized methods can be executed on a remote server at runtime regardless of the type of parameters passed as input arguments and whether the method is static or non-static. The framework from [2] was able to only offload methods with primitive parameters. Our extension allows methods with complex parameters to be optimized for offloading by serializing the complex objects and send them along with the offloading request to the server.

The method translation algorithm in the post-compiler and the remote execution platform is further extended to synchronize the object instances between the devices involved in the offloading. While the previous method translation algorithm could offload only static methods that can only access static variables, our optimization fully exploits object management, allowing to offload non-static methods that can access any variable both primitives and complex. When the method is executed remotely, the application: i) sends the variables to the remote server; ii) updates the values of those variables in the server-side; iii) invokes and executes offloaded method; iv) sends the values of the variables back to the client; v) update the client variables.

IV. Demo Overview

The demo setup includes an Android-based smartphone client and a laptop-based server under the same wireless network. After the post-compiler creates a new modified APK file, the client installs and runs the application on the mobile device. The application, through the method selection algorithm, indicates which methods can be offloaded, and the offloading framework, by historically monitoring data about latency and energy usage, decides whether to offload a method in order to optimize the application execution. The application then sends the selected optimized methods smoothly to the server, at runtime, returning the results of the computation to the device in a complete transparent way for the final users. The application can be downloaded from [4], also showing a video demo.

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] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 49–62.

[2] 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.

[3] R. Valle-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–.

[4] "ULOOF Framework." [Online]. Available: http://uloof.lip6.fr