Annotation Based Build Process Automation for Cyber Foraging Frameworks

P Vekneswaran [1,#], NR Dissanayake[1]

[1]Informatics Institute of technology, Colombo 6, Sri Lanka
[#]prathieshna@hotmail.com

Abstract— Cyber Foraging is a technique introduced to utilize the computing resources in the vicinity to improve the performance and the standby of the portable mobile devices. There have been various attempts to enable Cyber Foraging in smartphones, and they require lots of developer effort to offload the work from the applications in the mobile device, where the developers are required to do lots of code modifications or additions. We introduce an annotation based approach to automate the work offloading in a Cyber Foraging system to a greater extent, which requires minimal developer workload, through flexible technique. This can be refined in the future and be automated using machine learning algorithms, reducing developer effort furthermore.

Keywords— **Annotation, Automation, Offload, Cyber Foraging**

I. INTRODUCTION

This section gives a brief background to the cyber foraging, specifying the problem we are focusing on and the motivation towards the proposed annotation based approach. Then the methodology used in the research is specified.

A. Background

As the portable devices become smaller and smaller the computing capabilities and the standby time of them have become a questionable. Having a bigger battery and having a very powerful processor is no longer an option due to the requirement of smaller size factor. Cyber Foraging is a technique introduced by Satyanarayanan, which can help portable devices to take advantage of the unused computing resources in the vicinity. This exploited computer infrastructure is known as a Surrogate machine.

Cyber Foraging leads one step closer to M. Weiser's vison of ubiquitous computing, where the environment is saturated with technology, working together towards improving the experience of the user. Due to the higher developer effort required to create a cyber foraging system, or to enable cyber foraging application, the adoption of Cyber Foraging techniques into mobile applications is low.

There are so many attempts and approaches made towards achieving Cyber foraging, each having its own techniques, which will be reviewed in the section II. In section III, we will discuss the solution we propose and its limitations. Section IV discusses the testing and evaluation of the proposed approach, and section V concludes the paper, also specifying the future work.

B. Methodology

Literature survey was done to analyse existing work and gain the background knowledge. 57 preliminary researches had been conducted in the domain of enabling Cyber foraging (Lewis & Lago, 2015), and the survey published by Lewis & Lago was helpful in revisiting the approaches, and also recent approaches had been surveyed and considered within the scope of this research.

Empirical evidence was gained based on observing and experimenting on incorporating aspect orientation in development phase in the direction of reducing developer efforts, rather than going for older methods such as developing and deploying the surrogate and mobile components separately. Furthermore, experiments were done in order to identify support for most types of the commonly used method signatures and return types, therefore to understand correct boilerplate methods to be generated in the compilation time of the mobile application.

We worked on designing and developing the proposed solution on top of the Intellij Idea development platform for Mobile Application Development, which is used by Google under the name of Android Studio. Google has officially stopped the support for the eclipse; however, it was made sure that supports legacy systems as well.

II. EXSISTING WORK

In this section we discuss and review the approaches used in the existing cyber foraging work to offload the work from mobile device to the surrogate machine, in order to understand the gap, we intend to fill using the approach we propose in this paper.

One of the major reasons why cyber foraging adaptation rate is low is because of the developers' burden on

(re)writing the code requires separate mobile and surrogate components to be written. There are three common ways to achieve cyber foraging. Web Server Architecture, Mobile Agents and Virtualisation.

Web Server Client Architecture requires many services to be written along with server components for each application. Mobile agents approach requires mobile optimized surrogate components to be written as it uses mobile devices in the surrounding to offload. Virtualisation is the easiest way to enable cyber foraging, where the developer does not need to modify the application at all. However, the overhead of the Virtualization approach is comparatively high (Sharifi, et al., 2012). Some researches/solutions we identified in our literature survey are reviewed below, specifying the approaches they utilize, other techniques they have incorporated, and their pros and cons.

A. *Cuckoo Development Framework*

Cuckoo's development model, which provides the following to reduce developer effort in enabling Cyber Foraging in applications. Developer has to create interface of the intensive task and after Cuckoo generates the dummy stub, developer will have to overwrite the dummies. This give more flexibility than only having an annotation based solution. (Roelof, et al., 2012) A very simple programming model that is prepared for connectivity drops, supports local and remote execution and bundles all code in a single package. Integrating with existing development tools that are familiar to developers. Automating large parts of the development process. A simple way for the application user to collect remote resources, including laptops, home servers and other cloud resources.

B. *AMCO*

AMCO is an annotation based framework, which provides conversion of java code to cyber foraging enabled application. The programmer marks the components suspected of being energy hotspots. In AMCO, components can be defined at any level of program granularity, with the smallest being individual methods and the largest a collection of packages. To mark hotspot components, AMCO provides a Java annotation *@OffloadingCandidate*; this information can also be specified through an XML configuration file. Based on this input, an analysis engine first checks whether the specified component can be offloaded as well as any of its subcomponents (i.e. successors in the call graph). The engine also calculates the program state, to be transferred between the remote and local partitions that would need to be transferred to offload the execution of both the entire component or of any of its subcomponents. A bytecode enhancer then generates the checkpoints that save and restore the calculated state for the entire hotspot components as well as for each of its subcomponents. (Kwon & Tilevich, 2013)

C. *Scavenger*

Just like previously described AMCO, The Scavenger developer model is also based on. The Scavenger library has two operating modes: manual and fully automatic. The automated mode works by the use of Python function decorators. Using this built-in language feature adding cyber foraging can be as simple as adding a decorator to a function that may benefit from remote execution. (Kristensen & Bouvin, 2010) The limitations are the function must be self-contained, i.e., it must not call other functions or methods defined elsewhere in the application. Modules used from within the function must be imported within the function itself, so that these modules may also be imported and used at the surrogate.

D. *Analysis of Development Approaches*

Even though it is easier to develop on top of frameworks such as Scavenger and AMCO because it is just a matter of annotating the offload candidates in the program and the rest of the application build process is handled by the frameworks, the developer has less freedom to improvise the outcome of the generated code and the application.

These approaches are designed to reduce developer burden even though the frameworks are rigid when it comes to exceptional scenarios. Cuckoo framework provides freedom to the programmer to handle exceptional scenarios even though converting an app through Cuckoo is a time consuming for developers; and it gets trickier if the developer is not aware of the application he/she is converting however as noted by Flinn et al., a little application specific knowledge can go a long way when preparing an application for distribution. However, there is no validation for this approach to avoid using device local hardware inputs from sensors. These approaches have strong functional limitations when it comes to handling states, task mitigation and migration.

III. PROPOSED APPROACH

In order to enable less developer effort in cyber foraging enabled mobile applications development, we think that the assistance provided by the development framework to the developers is an important fact. In the direction of this

fact, the work offloading components – in the framework of our ongoing research – were supposed to be designed and developed initially, in order to identify a proper technique to offload work with minimal developer workload, before experimenting with other components like decision making engine, resource monitors, or surrogate service, to integrate them accordingly. Aspects reduce the build process immensely, without needing to write separate code for the surrogate components, which will be discussed in the following section.

A. *Aspect based Annotation*

Candidate methods for offloading can be scattered all over the application, thus the application has to invoke the decision making engine every time before such method is invoked. Knowledge of those offload method calls and log messages is simply irrelevant to the business logic in the class. In such scenario, The Aspect Orientation can be used in mobile application code, to identify the compute intensive tasks all around the application, and while using it won't interfere with the business logic or the object orientated design and architecture of the application.

An aspect is a common feature, which is typically scattered across methods, classes, object hierarchies, or even entire object models (JBoss, 2016). Aspect has two components. The advice is the code that will be injected to the class, joint point is the point of interception of the class (Cejas, 2016). The boilerplate code is the section of the code that needs to be included in many places throughout the application, i.e. the decision making logic in cyber foraging. Generation of boilerplate code is done by AspectJ (Java runtime for Aspect Orientation) by interpreting the advice and the joint points during the build process, which will reduce the developer effort drastically.

The annotation is the key of our proposed approach, which helps the developer to specify the identified intense tasks in the application. In this approach, there are already predefined advices, which need to be injected during the compilation phase itself. Here the advice is the code that is injected into the class file; typically, which needs to be inserted before, after, or instead of the target method. When the method is marked using the "offloadMethod" annotation, AspectJ will generate the boilerplates that is necessary for the annotated method. Refer the figure 1 for this scenario.

In our approach, we suggest to annotate the potential offloading candidate methods in classes, using the "offloadMethod" using Aspect orientation. Just using one annotation, identifying different methods with different behaviours is not possible. Therefore, it is necessary to be able to annotate a single method using multiple annotations as a part of the framework we propose; so the developer can create his custom defined aspects through creating new annotation interfaces using default templates provided.

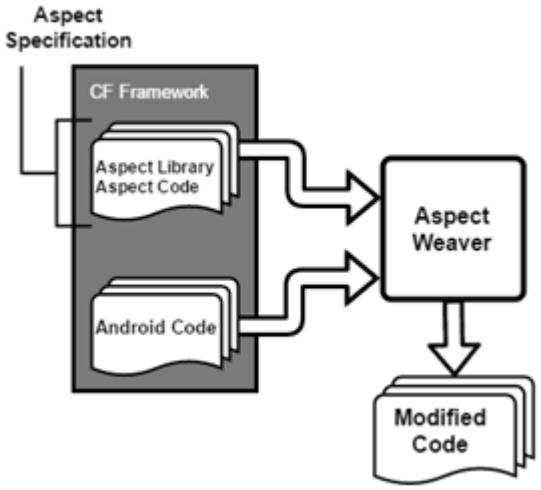

Figure1. Annotation Process

B. *Development Process and Limitations*

When using the annotations, the developer should identify the methods that consists of the compute intensive components that has be considered by the decision making engine during the runtime, weather it needs to be offloaded or not. This requires the developer to have a basic knowledge of the application source and the flow. The next step the developer should make sure the following. Identified intensive methods cannot reside inside Android Activities. These methods should not attempt to read hardware specific sensor data inside their scope, which will cause errors during the runtime. Alternative is to read the sensory data and pass it as a parameter. The intensive components should be added to a separate java class for it to be executed correctly in the surrogate environment. If the source is already separated this step can be ignored.

These candidate methods should be marked by the developer by adding annotations. During the build time boilerplate codes will be added by the AspectJ runtime, which is included in the framework. There are predefined aspects, which the developer can use, or if the developer is looking for some distinct characteristics, he can modify the aspects accordingly. Afterwards developer can build the

application and produce the APK, which is ready to be installed in the mobile devices.

The surrogate service should be running at the time when the application is about to execute the compute intensive tasks. According to configuration, the mobile application will find the surrogate device and attempt to connect to the service and offload. The surrogate service will then check if the mobile application is already in its repository of packages. If not, it will pull the source from the mobile device. As the components are sent automatically to the surrogate, there is no need for the developer to manually deploy the components explicitly. This is a onetime process, and afterwards any mobile device running the similar application can offload to that particular surrogate device, without pulling the source to the surrogate. Once the source is loaded, the surrogate will execute the task and return the result back to the mobile device. If it is not beneficial to offload, the mobile device will execute the task normally without offloading to the surrogate.

IV. EVALUATION

The proposed solution will have the following advantages over the existing development approaches.

It eliminates the need for writing two separate codes to do the same task in the mobile and surrogate, to enable cyber foraging in new applications. Also no need to modify the existing source code, just need to annotate the offload candidate methods. This will cut down the time required by the developer to enable cyber foraging in mobile applications. The Aspect based approach will allow the developer to plug and play his own logic in the decision making engine, communication protocol etc. according to his requirements giving him/her full freedom to customize the outcome. When building the application, the AspectJ runtime will handle the generation of the necessary boilerplate codes, which will further cut down the developer cost. In comparison to the existing approaches that is discussed in Section 2, the proposed solution in this paper has majority of the development process automated and easily configurable through the build scripts. This approach will also give developers, enough customisation options as well.

V. CONCLUSION AND FUTURE WORK

Even though cyber foraging can be easily achieved through virtualization, there are other factors such as functional overhead, which have adverse impact on the energy consumption and performance. We can conclude that the offload technique we propose is more effective in a preliminary level, reducing the time consumption for the development by automating most of the build process with the help of AspectJ and Android Development Environment.

We expect to further improve the proposed approach and introduce a framework, allowing more common features like integration to the available IDEs, which will reduce the developer effort furthermore, supporting rapid development. Future of this research will extend to use Machine Learning algorithms to identify the intense segments of code during runtime and offload without requiring the developer to annotate each offloading candidates.

REFERENCES

Balan, et al., 2007. *Simplifying cyber foraging for mobile devices.* s.l., ACM, pp. 272-285.

Balan, R. et al., 2002. *The Case for Cyber Foraging.* s.l., ACM, pp. 87-92.

Barbera, M. V., Kosta, S., Mei, A. & Stefa, J., 2013. To Offload or Not to Offload? The Bandwidth and Energy Costs of Mobile Cloud Computing. s.l., IEEE.

Cejas, F., 2016. *Aspect Oriented Programming in Android.* [Online] Available at: http://fernandocejas.com/2014/08/03/aspect-oriented-programming-in-android/
[Accessed 1 Apirl 2016].

Chun, B. G. et al., 2011. *Clonecloud: elastic execution between mobile device and cloud.* s.l., Proceedings of the sixth conference on Computer systems, pp. 301-314.

Cuervo, et al., 2010. *MAUI: making smartphones last longer with code offload..* s.l., Proceedings of the 8th international conference on Mobile systems, applications, and services, pp. 49-62.

Flinn, Jason, Park, S. Y. & Satyanarayanan, M., 2002. *Balancing performance, energy, and quality in pervasive computing.* s.l., IEEE, pp. 217-226.

Flores, H. et al., 2015. Mobile code offloading: from concept to practice and beyond. s.l., IEEE, pp. 80-88.

Flores, H. & Srirama., S., 2013. *Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning..* s.l., Proceeding of the fourth ACM workshop on Mobile cloud computing and services, pp. 9-16.

Gordon, M. S. et al., 2012. COMET: Code Offload by Migrating Execution Transparently. s.l., OSDI, pp. 93-106.

Goyal, S. & John, C., 2004. A lightweight secure cyber foraging infrastructure for resource-constrained devices. s.l., IEEE.

JBoss, 2016. *Chapter 1. What Is Aspect-Oriented Programming?.* [Online]
Available at: http://docs.jboss.org/aop/1.0/aspect-framework/userguide/en/html/what.html
[Accessed 30 March 2016].

Kosta, et al., 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. s.l., INFOCOM, 2012 Proceedings IEEE, pp. 945-953.

Kristensen, M. D., 2007. *Enabling cyber foraging for mobile devices.* s.l., Proceedings of the 5th MiNEMA Workshop: Middleware for Network Eccentric and Mobile Applications, pp. 32-36.

Kristensen, M. D., 2008. *Scavenger—mobile remote execution,* s.l.: s.n.

Kristensen, M. D. & Bouvin, N. O., 2010. Scheduling and development support in the Scavenger cyber foraging. *Pervasive and Mobile Computing,* VI(6), p. 677–692.

Kwon, Y.-W. & Tilevich, E., 2013. Reducing the Energy Consumption of Mobile Applications Behind the Scenes. s.l., IEEE.

Lewis, G. A. & Lago, P., 2015. A Catalog of Architectural Tactics for Cyber-Foraging. s.l., ACM.

Noble, B. D. et al., 1997. *Agile application-aware adaptation for mobility.* s.l., ACM, pp. 276-287.

Roelof, K., Palmer, N., Kielmann, T. & Bal, H., 2012. *Cuckoo: a computation offloading framework for smartphones.* s.l., Springer Berlin Heidelberg, pp. 59-79.

Satyanarayanan, M., 2001. *Pervasive computing: Vision and challenges..* s.l., IEEE, pp. 10-17.

Sharifi, M., Kafaie, S. & Kashefi, O., 2012. A survey and taxonomy of cyber foraging of mobile devices. s.l., IEEE.

Su, Y.-Y. & Flinn., J., 2005. Slingshot: deploying stateful services in wireless hotspots. s.l., ACM.

V, P. & R, D. N., 2016. *Cyber Foraging: What has been missing.* Colombo, ICIIT.

Weiser, M., 1991. *The computer for the 21st century.* s.l., Scientific american 265.3, pp. 94-104.