Energy efficient adaptation engines for android applications

Angel Cañete *, Jose-Miguel Horcas, Inmaculada Ayala, Lidia Fuentes

Universidad de Málaga, Andalucía Tech, Spain

Keywords:

Android

Dynamic adaptation

Energy efficiency

Mobile application

User behavior

A B S T R A C T

Context The energy consumption of mobile devices is increasing due to the improvement in their components (e.g., better processors, larger screens). Although the hardware consumes the energy, the software is responsible for managing hardware resources such as the camera software and its functionality, and therefore, affects the energy consumption. Energy consumption not only depends on the installed code, but also on the execution context (environment, devices status) and how the user interacts with the application.

Objective In order to reduce the energy consumption based on user behavior, it is necessary to dynamically adapt the application. However, the adaptation mechanism also consumes a certain amount of energy in itself, which may lead to an important increase in the energy expenditure of the application in comparison with the benefits of the adaptation. Therefore, this footprint must be measured and compared with the benefit obtained.

Method In this paper, we (1) determine the benefits, in terms of energy consumption, of dynamically adapting mobile applications, based on user behavior; and (2) advocate the most energy-efficient adaptation mechanism. We provide four different implementations of a proposed adaptation model and measure their energy consumption.

Results The proposed adaptation engines do not increase the energy consumption when compared to the benefits of the adaptation, which can reduce the energy consumption by up to 20%.

Conclusion The adaptation engines proposed in this paper can decrease the energy consumption of the mobile devices based on user behavior. The overhead introduced by the adaptation engines is negligible in comparison with the benefits obtained by the adaptation.

1. Introduction

The current smartphone market trend is the enhancement of the user experience by means of more powerful processors, larger screens and additional components like dual cameras. However, these components are boosting the energy consumption of smartphones [1]. Energy con- sumption directly affects user experience, forcing users to charge their phones daily, sometimes leading to external batteries being required to keep the device switched on throughout the day. Although the hard- ware consumes the energy, the software is responsible for managing the hardware resources (e.g., the camera software and its functionality), and therefore, also affects the energy consumption [2].

When a mobile application is running, the energy consumption not only depends on the installed code, but also on how the user interacts with the application [3]. Based on the user's interaction, the applica- tion functionality uses the available resources differently. For instance, a messaging application loads and displays all the friends' chats when it starts, even though some of them have not been contacted by the user for a long time. The application could be more energy efficient if it just displayed recent chats or most frequently contacted friends. Ideally, when a specific resource is not being used it should not be consuming energy (e.g., the screen, WiFi, GPS sensor or any other application service) [4,5].

Most mobile applications are deployed using static configurations [6]. These configurations are pre-planned to be energy efficient for a generic user behavior and, normally, they are not prepared for changes that may occur at runtime [7]. Therefore, when the user behavior changes, the energy efficiency of the application decreases [8]. In contrast, self-adaptive applications are able to self-adapt their behavior or structure at runtime in response to user behavior [9]. Existing self-adaptive mobile applications, whose goal is to decrease the energy consumption of the device, only take into account the device status (e.g., battery level, CPU frequency) or the environment (e.g., location, noise level) [6,10,11] to trigger the reconfiguration, and do not consider the user interaction with the application as part of their solution. Taking into account that the energy consumption of the mobile applications also depends on the user's interactions, our question is: *to what extent is it possible to reduce the energy consumption of the applications by adapting them to the user behavior?* That is, what is the quantifiable benefit to energy consumption of adapting the application to the user behavior?

To reduce the energy consumption based on user behavior it is necessary to dynamically adapt the applications. In this paper, we propose an adaptation model based on user interactions for Android applications. In the literature, there are several context-aware dynamic adaptation solutions for mobile applications based on different mechanisms such as polymorphic methods [10], reflection, dynamic components [6], dynamic offloading of code with the OSGi platform [12], evolutionary algorithms [11] and self-adaptive frameworks [13]. Each adaptation solution consumes a certain amount of energy, which may lead to an important increase in the energy expenditure of the application. However, there are no approaches that measure the energy consumption introduced by the adaptation engine itself. Therefore, it would be interesting to quantify the overhead introduced by the adaptation engine to assess its benefit in terms of energy saving during application execution [13]. In addition, existing adaptation technologies for mobile applications depend strongly on the execution environment (e.g., version of the Android system) and they become obsolete very quickly [10]. This is due to the continuous advancement in technologies made by companies like Google, which makes it difficult to find generic adaptation solutions that will not become obsolete in the next version of the system.

In this paper we propose four different adaptation engines for Android applications and evaluate the overhead in terms of energy consumption introduced by those adaptation engines. Our implementations, based on dynamic proxies and the Xposed framework[14], support all current versions of the Android system. We demonstrate that it is possible to reduce the applications' energy consumption by adapting the functionality at runtime to the user behavior, while maintaining the service quality levels required by the user. To summarize, in this paper we aim to answer the following research questions:

RQ1: To what extent is it possible to reduce the energy consumption of the applications by adapting them to the user behavior?

RQ2: What is the impact of the adaptation engine on the energy consumption of the mobile device? Which adaptation engine is more energy-efficient?

RQ3: Are the adaptation engines scalable? What scenarios could increase their energy consumption?

The rest of the paper is organized as follows. Section 2 introduces the background information to dynamic adaptation in Android applications and motivates our approach. Section 3 presents our adaptation model for mobile applications, while Section 4 details our four different implementations of the adaptation model. Section 5 evaluates our adaptation approach and discusses the threats to validity. Section 6 discusses related work and exposes the limitations of existing reconfiguration and energy optimization approaches in Android. Finally, Section 7 concludes the paper and presents future work.

2. Background information and motivation

This section presents the necessary knowledge about the Android architecture, and the existing approaches to dynamically adapt mobile applications based on the user's interactions.

2.1. Compilation and execution models in android

In Android, Java source code is compiled by the Standard Java Compiler to obtain Java bytecode (.class files) and then, this code undergoes a second round of compilation to obtain DEX bytecode (Android-specific). DEX files are signed and packaged to construct the Android Application Package (APK). DEX bytecode is translated into native instructions and executed by the Android runtime environment: *Android Runtime (ART)* or *Dalvik*. Fig. 1 summarizes the compilation and execution processes of an Android application.

ART is the current application runtime environment used by the Android operating system, which was introduced with the release of Android 4.4 (Kitkat). Prior to ART, the runtime environment for Android apps was the *Dalvik Virtual Machine (DVM)*, which was completely replaced by ART in Android 5.0 (Lollipop). The DVM has certain restrictions such as, it does not support custom classloaders, dynamic code generation and bytecode manipulation at runtime [15]. While Dalvik uses a *Just-in-Time (JIT)* compilation model, ART uses an *Ahead-of-Time (AOT)* compiler. More recently, Android 7.0 (Nougat) introduced a hybrid runtime model including a runtime JIT compiler with code profiling to ART.

2.2. Dynamic adaptation in mobile applications

This section presents existing solutions to dynamically adapt the behavior of mobile applications, and in particular, of Android applications. Fig. 1 summarizes these approaches indicating the moment (compilation, loading, execution) where they can be applied. For each approach, we describe how it works and discuss its limitations. Note that these approaches can be combined to benefit from their individual strengths.

2.2.1. OSGi platform

The **OSGi platform**[1] is a service and component-based platform which offers a class-loading engine to dynamically load and unload modules. OSGI is embedded within an Android APK, and the OSGi bundles can be called from within Android activities and services. There are some initiatives that propose OSGi for hosting applications in mobile devices, such as the OSGi Mobile Specification (JSR-232)[2] or Androsgi [16].

The main limitation of the OSGI platform is that it implies an additional application size, memory consumption, and start-up time, but this can be minimized if the OSGI platform is started inside the device and supports the needs of different application bundles [17,18]. Regarding energy consumption, the OSGi bundles does not affect the device's energy autonomy significantly [17].

2.2.2. Virtual-method hooking

Virtual-Method Hooking is a technique for managing an array of pointers to virtual functions that instances of the class may call (i.e., a virtual method table). These functions can be intercepted (hooked) by replacing the pointers to them within any virtual method table in which they appear [19]. An example of a framework using virtual-method hooking in Android is Xposed [14]. It extends the Android application launcher (`App launcher` in Fig. 1) to load external modules (JAR files) on startup without modifying the original APK.

However, this technique is not completely dynamic, since frameworks require the new functionality to be introduced at load-time, and thus, changes need to be predefined before the execution of the application.

2.2.3. Reflection

The use of **reflection** gives applications the ability to examine, introspect, and modify their own structure and behavior at runtime.

In Android, reflection is a powerful technique because it uses base Java code without needing external modules. However, it requires more effort from the application developers. Reflection also poses serious security problems in Android applications despite its widespread use [20].

2.2.4. Dynamic proxies

Dynamic proxies are wrappers that pass function invocation through their own facilities and can potentially add new functionality

[1] http://www.osgi.org.

[2] http://jcp.org/en/jsr/detail?id=232.

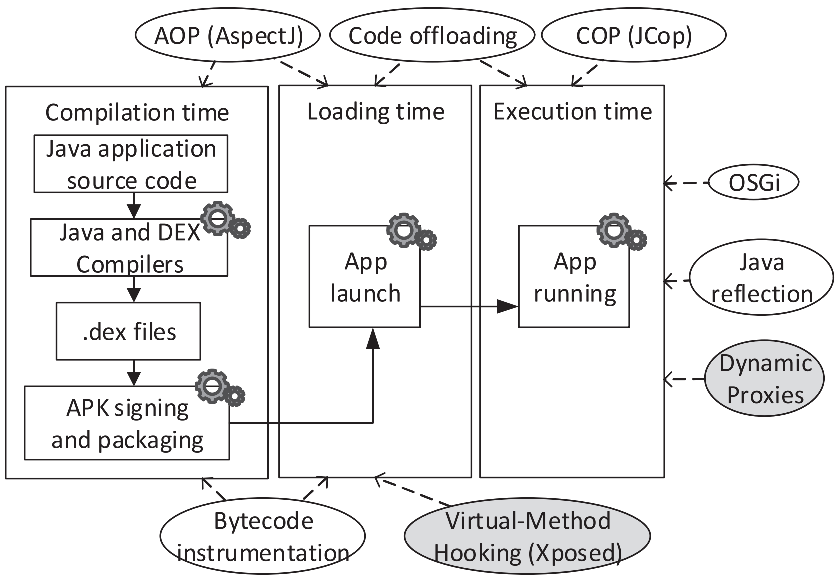

Fig. 1. Adaptation approaches in Android compilation and execution processes.

or modify the existing one. Dynamic proxies allow adapting the application's behavior by extending the original implementation of the application's functionality [21].

Dynamic proxies require a considerable planning effort on the part of application developers, but these techniques are easy to adopt and the overhead introduced in the system at runtime is very low [22]. Moreover, dynamic proxies, in contrast to reflection, do not represent a security vulnerability because developers can control the classes that can be adapted or not [23].

2.2.5. Bytecode instrumentation

Bytecode instrumentation is commonly used to modify the behavior of Java applications [24]. It consists in modifying the binary code generated by the compiler (e.g.,.class files). However, in Android, an additional compilation step is performed to generate.dex bytecode. As a consequence, bytecode modifications have to be applied to.dex files, or to.class files before DEX compilation. Whatever the case, bytecode instrumentation in Android is limited to compile-time because the final APK file must be built and signed to verify its content [24].

2.2.6. Dynamic offloading

Dynamic offloading is a technique that extends the capabilities of mobile devices, as well as their battery life, by migrating hard-computation tasks to resource-rich devices such as remote servers or cloud platforms (e.g., Google Awareness) [25,26]. Consequently, code offloading solutions require a set of surrogate devices on to which the client can offload tasks, as well as connectivity to the internet. Indeed, the rate increase of the mobile data traffic can become a cost issue when no WiFi (free) connection is available [27].

2.2.7. Aspect-oriented programming

Aspect-Oriented Programming (AOP) can be used to construct dynamically reconfigurable systems by providing an instrumentation engine that allows for the change of execution flows [28]. Aspects can intercept or change the behaviors of target components, without modifying their source code, by replacing the implementation of the components at runtime. However, components need to be introduced (woven) at compile or load-time. AspectJ is certainly the most successful language and compiler for implementing AOP, and can be used in Android for instrumenting mobile applications [29].

The main issue with AOP is that developers need to know the application's code to implement the appropriate aspects, and moreover, AOP requires having to learn a new programming paradigm. It also introduces an additional step, to test whether or not aspects will work without problems with the application code, complicating the development of the mobile applications.

2.2.8. Context-oriented programming

In **Context-Oriented Programming (COP)** [30], context-dependent behaviors are represented as partial method definitions and encapsulated inside layers [31]. Depending on the context, a method call can be dynamically redirected to a partial method by layer activation. An example of a COP language is JCop [15], a Java extension that can be applied to Android.

2.2.9. Ad-hoc solutions

Finally, there are **ad-hoc solutions** that build their own adaptation solution based on custom self-adaptive frameworks [10,13], function mapping [32] or evolutionary algorithms [11]. In general, these ad-hoc solutions are difficult for software developers to reuse, and require the specific solution to be studied in detail to adapt it to the developers' needs.

3. Our approach

This section presents our approach to adapt mobile applications based on user interactions. Adapting applications according to user behavior adjusts the number of tasks performed by the applications to the ones that are relevant to the user in a certain moment. Among the benefits that our approach provides compared with other adaptation engines are: (1) the QoS perceived by the user is not affected because adaptations are transparent to him/her; (2) our adaptation engine can be applied irrespective of the device's architecture in which applications are running; (3) our approach can be easily combined with other kinds of adaptation engines (e.g., screen off after a minute of inactivity); (4) user profiles and preferences that can vary a lot and change very often, offer many more opportunities to save energy compared with the device context, which is highly limited to signal strength, supported graphic mode, level of battery, or the status of any other device's resource, etc.

First, we propose a generic adaptation schema inspired by the MAPE-K loop [33]. Then, we present four different implementations following this adaptation schema. As described by Elmalaki et al. [10], a context-aware system can be divided into three parts: (1) a set of *replaceable*

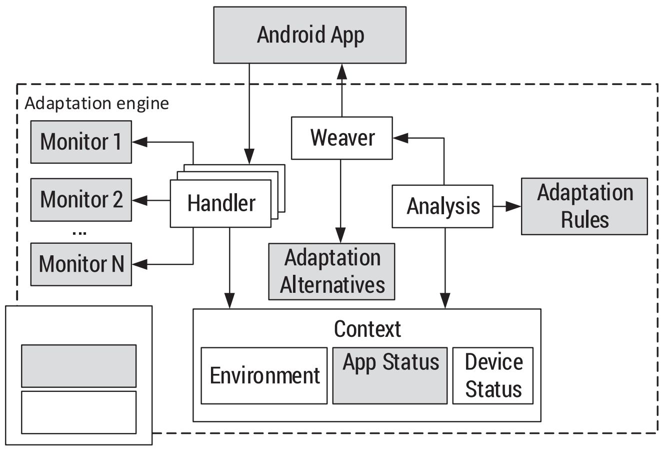

Fig. 2. Our dynamic adaptation approach.

polymorphic methods; (2) a *context monitoring system*; and (3) an *adaptation engine* that switches between the different methods based on the monitored context. Fig. 2 details our approach with these three parts in which (1) the replaceable polymorphic methods are the alternative implementations (`Adaptation Alternatives`); (2) the context monitoring system is managed through `Handlers` with access to several `Monitors` that track the application for changes; and (3) the adaptation engine responsible for making adaptation decisions (`Analysis`) driven by a set of `Adaptation Rules`, and performing the changes between the available adaptation alternatives (`Weaver`).

The proposed adaptation approach aims to be as unintrusive as possible within the application, therefore the application's base functionality (`Android App`) is modified by external modules. Our approach contains two kinds of modules: generic and specific. Generic modules (the white modules in Fig. 2) are integral parts of our adaptation engine and can be directly reused with different applications without modification. Specific modules (the shaded modules in Fig. 2) contain information about a particular application, and usually, this information is different for each application. Note that, in this paper we are interested in reconfiguring the base functionality of the applications, and not just the reusable functionality such as crosscutting-concerns (e.g., authentication methods, encryption algorithms). Using this approach, our adaptation engine enables different implementation paradigms to be used such as AOP, COP, Java reflections or the integration of the OSGi platform, just by modifying the weaver and handler modules and implementing the adaptation alternatives in the chosen approach. Our approach works as follows: **Handlers** The application objects to be modified are managed by handlers. The handlers intercept the interactions (calls) to the objects and execute the appropriate behavior (method) configured by the `Weaver` module.

Monitors Monitors obtain context information from the application. Each monitor is in charge of controlling a specific set of variables in the application. The information collected by the monitors depends on the adaptation trigger. For instance, in a chat application that shows the top five people who the user chats with, a monitor will keep an eye on the number of input/output chat messages of each chat. This information is used by the `Analysis` module and helps us to create the user profile.

Context This module maintains the information about the current application context. The contextual information is divided into three submodules [3]: (1) `Environment` represents the external information of the application (e.g.,location); (2) `App Status` maintains the application state (user behavior) between executions by using the information provided by `Monitors`; and (3) `Device Status` gets the state about the mobile itself (e.g., battery level).

Analysis This module makes decisions about the most appropriate functionality based on the application context, and especially on the user profile. The analysis module uses a set of adaptation rules that evaluate the information provided by the `Context` module to decide which of the alternative implementation classes and methods must be executed. Another alternative could be to delegate the analysis to a server, which allows the collection of users' profile information, which can then be exploited by techniques like machine learning to predict future user behavior based on the current one. In this paper we focus on decision-making based on adaptation rules.

Adaptation rules This module contains a set of expressions (adaptation rules) that define how the adaptation mechanism should act based on the current context. The adaptation rules can be included in the application using if/else statements or can be external and provided as a microservice. There are rules that can be applied to any application (e.g., decreasing the screen brightness when battery level is low or reducing sensor usage as far as possible). Other rules are specific to each application, for example, the rules that depend on the user profile. Here, it is possible to define different adaptation profiles making the mechanism more or less aggressive according to the energy policy. A high energy-saving policy can save more energy than a low energy-saving one, but it could compromise the user experience.

Weaver The `Weaver` is the module responsible for directly modifying application code. It is executed when the `Analysis` module requests an adaptation. How the code is modified depends on the `Weaver` module's implementation and will be different for each proposed solution (e.g., dynamic proxies).

4. Adaptation engines

We have implemented four different adaptation engines following the schema presented in the last section. Two of them are based exclusively on dynamic proxies while the other two are based on the Xposed framework, and a combination of the Xposed framework with dynamic proxies.

We have chosen these approaches for several reasons. On the one hand, dynamic proxies represent a generic approach independent of the platform and runtime environment, so it is completely compatible with any version of Android irrespective of its compilation and execution model. Dynamic proxies do not introduce a high overhead in comparison with other solutions such as the introduction of a middleware or framework. Additionally, we have chosen the Xposed framework because (1) it is a complementary solution to dynamic proxies (dynamic proxies carry out the adaptation at runtime and Xposed acts at load-time); (2) it does not need to be embedded within the application being reconfigured; (3) it does not introduce overhead at runtime; and (4) it can act over any component of the application functionality and even over aspects of the system such as the mobile screen. Finally, Xposed is currently maintained and up to date for the latest Android version.

The adaptation process (see Fig. 3) is similar for all adaptation engines. The application functionality consists of a set of instantiated classes *C* divided into two subsets: a set of fixed classes (`Non adaptable classes`) and a set of classes with functionality that can be changed (`Adaptable classes`, shaded in Figure 3). On the other hand, there is a repository of classes *A* (`Adaptation Alternatives`) that provide alternative implementations to the adaptable classes. To make possible the use of the Adaptation Alternatives, classes *A* (e.g., class `C3'`) implement the same interfaces as classes from *C* (e.g., class `C3`). Finally, the set of adaptation rules *R* indicate which functionality or methods (e.g., method `m2` of class `C3`) should be changed by the new functionality or methods (e.g., method `m1` of class `C3`"), under a certain application context (e.g., `context1`).

The way the adaptable and the alternative adaptation classes are managed determines whether the adaptation engine is based on dynamic proxies or on Xposed. The four proposed adaptation engines are **AE1: Internal Proxy, AE2: External Proxy, AE3: Xposed Proxy** and **AE4: Xposed**.

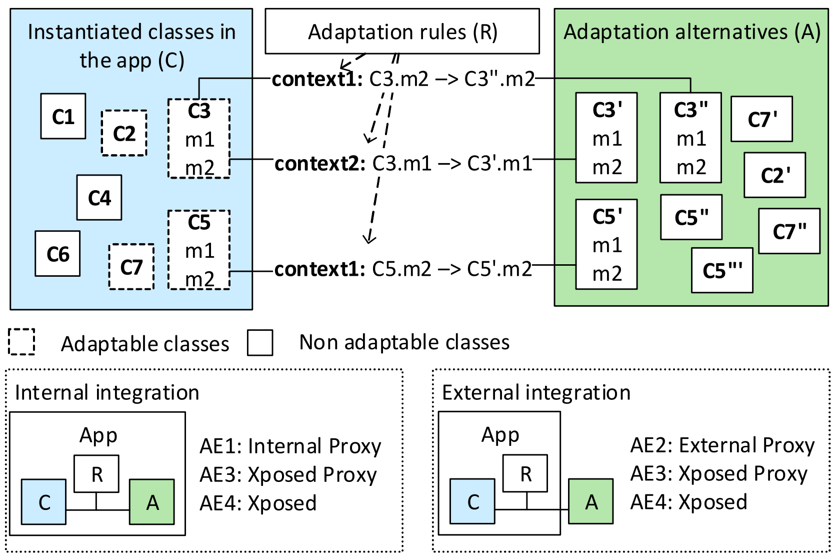

Fig. 3. Dynamic adaptation process.

4.1. Adaptation engines based on dynamic proxies

When adaptable classes are implemented as dynamic proxies, the proxy works as a class handler, controlling the access to the class functionality. In addition, the proxy facilitates the creation of the user profile by controlling the usage of each class and method.

When the context changes, the weaver modifies the class functionality with the most energy-efficient alternative. Alternative classes can be located in the application code or in external files. Based on the location of the adaptation alternatives, two different implementations of dynamic proxies are possible, the **AE1: Internal Proxy** and **AE2: External Proxy**. The code adaptation is produced at runtime, in both, and is transparent to the user.

Using the **AE1: Internal Proxy**, adaptation alternative classes are available with the application code. The weaver will instantiate the classes when needed. If a class is not being used during the application execution, it will not be instantiated, and therefore, it will not consume resources.

In contrast, when **AE2: External Proxy** is used, the application only contains the basic functionality of its APK file. Alternative classes are contained in external files, separate from the application code (*.dex* files, in Fig. 1). These files are located in the external memory of the mobile device and it can contain multiple alternative classes for any adaptable class. The weaver will load these files and instantiate the classes on demand. If a class is not required during the application execution, its code will not be loaded nor instantiated in the application.

4.2. Adaptation engines based on the Xposed framework

Using the Xposed framework, the application can be modified just before it starts (i.e., at loading time). At the beginning of the application execution, depending on the context and attending to the adaptation rules, the functionality of the adaptable classes is changed by the most energy-efficient adaptation alternative. In this case, adaptation alternatives are available when the application is launched. These alternatives are contained in modules, which can be external to the APK. We can enable and disable the modules using the Xposed Installer tool. There are two different types of adaptation mechanisms based on how the system creates the user profile, **AE3: Xposed Proxy** and **AE4: Xposed**.

When the **AE3: Xposed Proxy** is applied, we use dynamic proxies to manage and monitor the access to the adaptable classes. Nevertheless, the difference between this solution and AE1 and AE2 is that we use the Xposed framework to adapt the application. In this case, adaptation rules are evaluated when the application is launched, so the adaptation is performed at loading time. Note that in this solution, dynamic proxies are only used to monitor the user behavior and not to perform the adaptation.

The **AE4: Xposed** solution realizes the functionality adaptation like AE3, but the management and monitoring of the adaptable classes are done by specific calls by means of the interception system of Xposed.

In both cases, the advantage of using Xposed is that the changes between adaptable classes are managed by the framework, alleviating the developer of the task of implementing reconfigurable interfaces for the adaptable classes. This allows the focus to be on monitoring the user behavior to keep track of the application context information. Nevertheless, the functionality adaptation is done at launch-time (loading time).

5. Evaluation

In this section we evaluate our proposal. We measure the energy consumption of our adaptation solution in different scenarios to answer our research questions (RQs).

5.1. Experimentation setup

The evaluation has been carried out over two mobile devices and different adaptation scenarios. The energy consumption has been estimated using two different tools: GreenScaler [34,35] and Trepn Power Profiler[3]. The results of these evaluations and the scripts to replicate them are publicly available[4].

5.1.1. Adaptation scenarios

Applications in which our adaptation engines can be applied are very diverse, which makes it impossible to evaluate all of them. Nevertheless, it is possible to evaluate the factors that can affect the amount of energy saving when using our energy-efficient adaptation engine. For this reason, we have developed a lightweight benchmark application that enables the applicable scenario features to be configured.

In addition, we have also developed two possible scenarios in which adapting applications according to the user behavior make sense. First, by adapting the number of data/functionality requests and amount of information displayed on the screen according to the user profile. Examples of applications with this behavior are instant messaging apps (e.g., WhatsApp, Telegram), newspapers and weather apps, sport reports apps

[3] https://developer.qualcomm.com/software/trepn-power-profiler.
[4] http://caosd.lcc.uma.es/research/rsc/android-adaptation-results.zip.

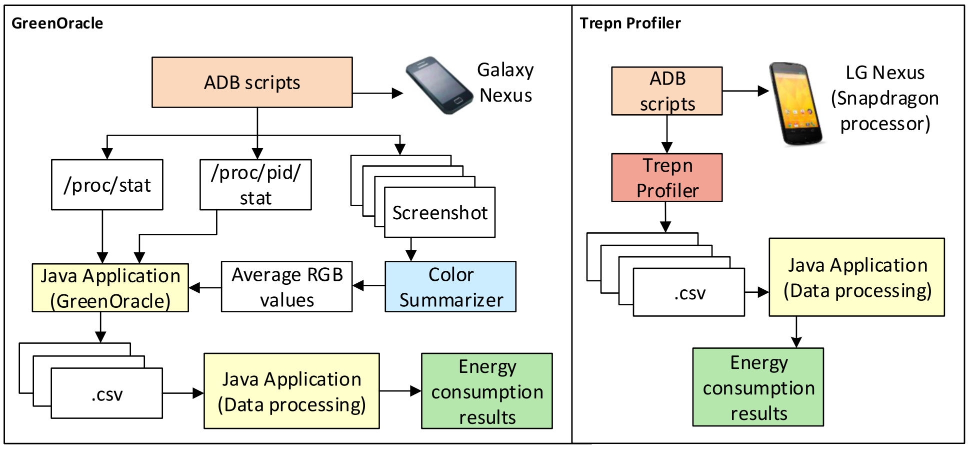
GreenOracle
ADB scripts
Galaxy Nexus
/proc/stat
/proc/pid/stat
Screenshot
Java Application (GreenOracle)
Average RGB values
Color Summarizer
.csv
Java Application (Data processing)
Energy consumption results
Trepn Profiler
ADB scripts
LG Nexus (Snapdragon processor)
Trepn Profiler
.csv
Java Application (Data processing)
Energy consumption results

Fig. 4. GreenScaler and Trepn Profiler energy profiles.

(e.g., LiveScore, DAZN), and market places (e.g., Play Store). The energy consumption of these kinds of applications can be reduced significantly if they only request and show the information which is relevant to the user. For instance, in the case of WhatsApp, it loads all the friends' chats when it starts. Since the user usually only texts his/her most popular contacts, if it only displays and requests the information of these contacts, it can save energy without affecting the user experience.

The second scenario for adapting user behavior consists in introducing a new functionality (or adapting an existing one) according to the known user profile (e.g., data compression before sending) [36]. Examples of applications with this behavior are social apps (e.g., Facebook, Instagram), e-mail clients (e.g., Gmail, Microsoft Outlook), videoconference apps (e.g., Skype, Hangouts), or Google Photos. Normally, these applications offer an interface where the user can select a video or photo from the gallery and choose between sending it as it is or compressing it. The adaptation in this case consists of selecting the compression algorithm taking into account the device context. Specifically, we consider the battery level, if the mobile phone is charging or not, and the available networks. The aim is to find the balance between the computational cost of the compression and the cost of sending the media file using the network on hand.

5.1.2. *Energy profile*

To estimate the energy consumption we use two different software-based tools, GreenScaler and Trepn Profiler. We have chosen these tools because they allow the energy consumption of mobile applications to be estimated, with an upper error bound of less than 10% [34,37] and they have been used in other studies on energy consumption [38–40]. Although hardware-based tools provide more precise results, it is not easy to identify the part of the software responsible for this consumption using them. In order to apply these tools, we use the framework depicted in Fig. 4, which was presented in [38].

GreenScaler is an energy model generated by the processing of hundreds of energy measurements obtained by the GreenMiner model [41] using 472 real world Android applications. The general idea of this tool is to estimate power consumption using information provided by the operating system (e.g., number of CPU jiffies) and information that can be extracted during the normal functioning of the app (e.g., screen color). This information is the input of the GreenScaler energy model, which provides an estimation of the energy consumed by the app at a given time. In order to collect this information, we have developed several scripts for Android ADB (Fig. 4) and a Java application that automatically process the information and apply the energy model. GreenScaler considers RGB values in its energy model, we obtain these values using the API of Color Summarizer[5].

Trepn Profiler is a commercial measuring tool able to provide energy consumption information compatible with the majority of Android devices (Android 4.0 and higher), but particularly intended for devices with a Qualcomm Snapdragon processor. Trepn Profiler collects the power readings from the power management integrated circuit and the battery fuel gauge software. Using ADB commands, we set the measure tool profile, launch the application to be measured, start the Trepn Profiler service, and, finally, download the energy measurements information (Fig. 4).

In order to compare our four adaptation engines[6], we generate 10 random user behaviors and then we replicate them in each experiment. This includes service requests and the application closures and launches. For each experiment, we perform 20 executions, obtaining the average and standard deviation of the energy consumption.

5.1.3. *Mobile devices*

We have used two mobile devices for the experiments: (1) Samsung Galaxy Nexus with Android 4.3 and Dalvik VM, and (2) LG Nexus 5 with Android 6.0 and ART.

All the energy measure tests have been carried out with the mobiles in the same conditions: flight mode activated, WiFi turned on in order to send commands through the ADB tool to execute each test, applications' updates are deactivated and only the corresponding application is executing in the foreground. To ensure the same operating system behavior, we turn off the Android energy-saving mode.

5.2. *Results*

This section shows the experimentation results to answer our research questions (RQs). For each RQ we explain the motivation, the experiments performed, the analysis of the experimental data, and the findings.

5.2.1. *RQ1. Benefits of our adaptation engines*

Research Question 1 To what extent is it possible to reduce the energy consumption of the applications by adapting them to the user behavior?

Motivation Self-adaptive applications are able to self-adapt their functionality at runtime, in response to changes in the context. Existing self-adaptive mobile applications whose goal is to decrease the device's energy consumption do not take into account the user behavior with the application as part of the context, as they only consider the device status as the reconfiguration trigger. This research question explores the influence of the reconfiguration of the application functionality in the

[5] http://mkweb.bcgsc.ca/color-summarizer/?home.

[6] Source code available at https://github.com/angelcvx/Adaptation-Engines.

global energy consumption of a mobile application based on the user behavior.

Experiments To answer RQ1 we measure the energy consumption of the base applications running with their default behavior (i.e., without any adaptation or adaptation engine integrated). Then, for each adaptation engine (see Section 4), we repeat the experiments but adapting the application to the user's interactions.

In the first scenario, the application has been adapted to reduce the number of requests to the server and to show only the information the user is interested in. We exemplify this scenario with a sport reports consulting application. By default, this applications shows information about 25 different football leagues. The adaptation engine collects the number of user requests for each league and saves it as part of the context. After five requests, the reconfiguration rule defines that only the most consulted leagues by the user should be shown. So, the adaptation engine modifies the functionality by changing the appropriate classes and methods.

In the second scenario, we use an application that compresses and sends a video selected by the user. The application has been adapted to change the compression algorithm with different compression ratios before sending the videos through the network. The user interface of the application has also been adapted to reduce its energy consumption while the compression takes place. By default, the application uses light colors in its interface and an algorithm with a low compression ratio (around 20% of video compression) in order to maintain the video quality. The adaptation rules specify that the application reconfigure its functionality in order to use an algorithm with a higher compression ratio (around 75% of video compression) in two cases: (1) the battery level of the mobile is lower than 25%, the device is not charging and the user shares a video; (2) the user shares a video while the device is connected to a low-band network. In addition, the user interface is adapted by providing darker colors [8] when the device battery level is lower than 25% and the device is not connected to power.

Analysis A Wilcoxon signed-rank test has been applied to evaluate if there is a statistical difference between adapting the applications or not (see Fig. 5). The null hypothesis is that the energy consumption is the same in both cases: the base application $ec_{app_{base}}$ without adaptation and the application adapted $ec_{app}{}^{adap}$:

$$H_0 : ec_{app_{base}} = ec_{app_{adap}} \qquad H_1 : ec_{app_{base}} \neq ec_{app_{adap}} \tag{1}$$

To do so, first, we determine whether or not the measurements are normally distributed by applying a Shapiro-Wilk normality test [42]. This test shows that not all measurements are normally distributed. Hence, we apply a non-parametric test. Specifically, we use a Wilcoxon Rank Sum test [43]. Using a confidence interval of 95%, $p \leq 0.05$, results for the Wilcoxon test show that there are statistical differences between the energy consumption of the non-adapted application and the adapted-application. We repeat this test comparing the application without any adaptation engine and the adapted version using each adaptation engine. The results show a statistical difference in all cases, with a p-value around 0.001. Additionally, we run a statistical power analysis to check that the size of the sample is enough to discard Type II error (i.e., failure to reject a false null hypothesis or false negatives) with a level of significance lower than 0.05. The observed power is 1 for both case studies. So, we can conclude that the number of random behaviours and repetitions of the experiments are enough to answer this research question.

Findings A benefit of up to 20% can be observed using the adaptation solution based on Internal Proxy (AE1) and External Proxy (AE2) in the first application (34 Joules). Fig. 6 clearly shows the difference in the energy consumption over time. We observe a benefit up to 20% (from 168 Joules to 134 Joules) comparing it with the base application without adaptation. This benefit corresponds to the adaptation engines based on dynamic proxies, being smaller (10-12%) for the adaptation engine based on the Xposed framework. The differences between the

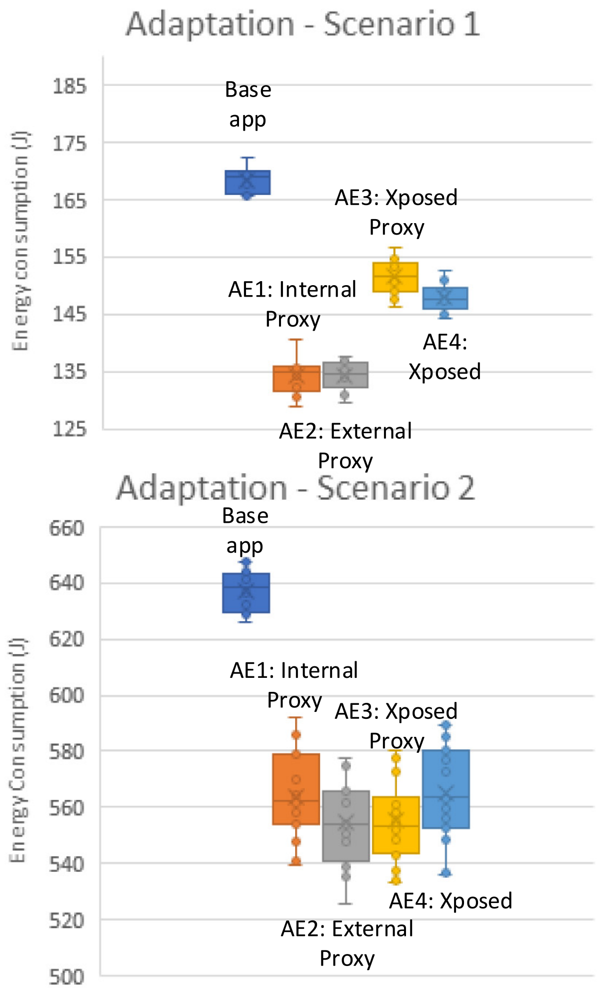

Fig. 5. Box plot of the energy consumption before and after the adaptation.

solutions correspond to the time when adaptation is performed (runtime for dynamic proxies based solution vs load-time for Xposed based solutions).

In the second application, we obtain a benefit of 13% (from 638 Joules to 558 Joules), a decrease in energy consumption in more than 80 Joules (see Fig. 6). However, in this case, there is not a major difference between the four adaptation engines because the user only interacts with one functionality and then closes the application. The execution time is smaller with no difference appreciated in adapting the application at runtime or at load-time. Therefore, depending on the expected use of our application, we can select one adaptation solution or another

5.2.2. RQ2. Energy consumption of the adaptation engines.

Research Question 2 What is the impact of the adaptation mechanism in the energy consumption of the mobile device? Which adaptation mechanism is more energy-efficient?

Motivation The energy benefit obtained by using the proposed adaptation engines will depend on the functionality changes carried out in the application. In turn, the latter will depend on the application. All adaptation engines have an associated computational cost. Adaptation engines monitor the context, analyze the information, evaluate the adaptation rules and change the functionality of the application. The objective of RQ2 is to evaluate the overhead in energy consumption introduced by each adaptation engine. This is sensible as (1) it allows the viability of using our approach to be predicted according to the number of functionalities fixed to the user behavior; and (2) they can be used not

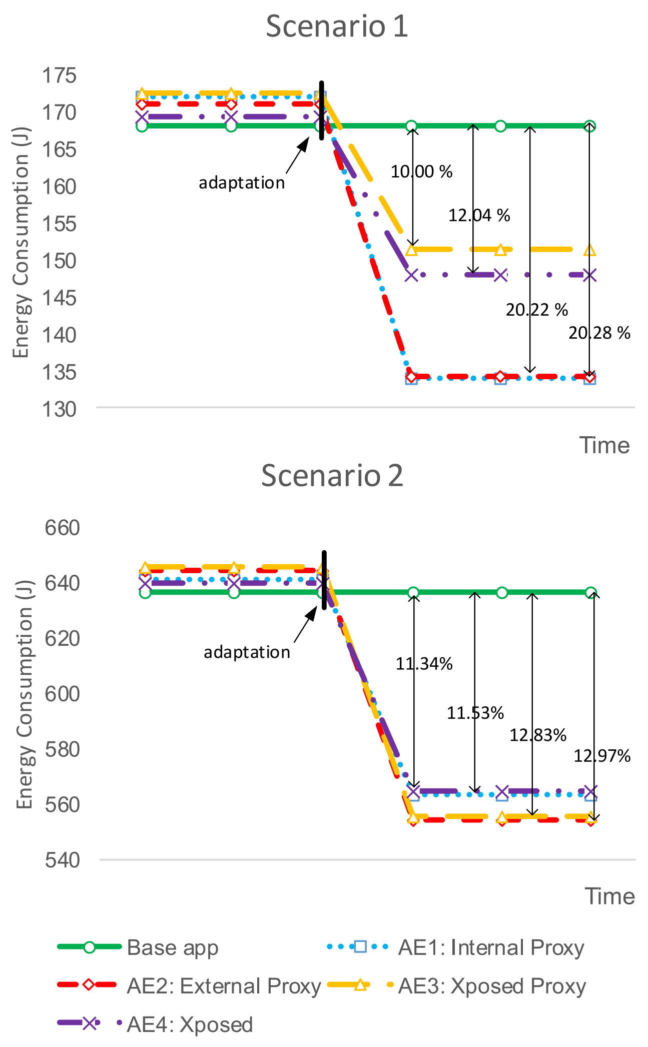

Fig. 6. Energy consumption before and after the adaptation.

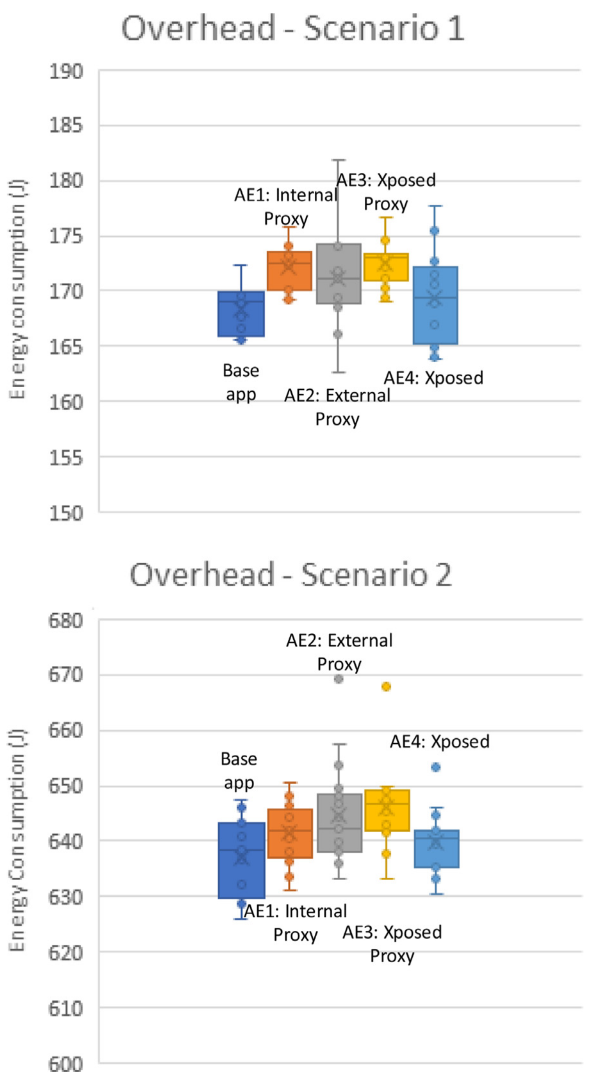

Fig. 7. Box plot of energy consumption overhead introduced by the adaptation engines.

only with the aim of reducing energy consumption, but also with other objectives (e.g., improving the user experience, reducing the execution time).

Experiments To evaluate just the energy consumed by the adaptation engines, we integrate each adaptation engine with the base applications and check the normal operation of the application. Adaptation engines monitor the application's context, manage adaptable classes, evaluate the adaptation rules, and load the external implementation alternatives if needed. However, for these experiments, the behavior of the application is unchanged. The energy consumption of the application is measured without modifying its behavior to observe the energy cost of the monitors, handlers, analysis, and weaving components of the adaptation engines.

Analysis As for RQ1, we use a Shapiro-Wilk normality test to verify that the energy measurements are not normally distributed. Thus, an hypothesis contrast (Wilcoxon signed-rank test) determines whether there is a statistical difference between the energy consumption of the base application ($ec_{app_{base}}$) and the energy consumption of the application with each of the adaptation engines integrated ($ec_{app_{adapt}}$). The null hypothesis is that the energy consumption is the same in both cases:

$$\begin{aligned} H_0 &: ec_{app_{base}} = ec_{app_{adapt}} \\ H_1 &: ec_{app_{base}} \neq ec_{app_{adapt}} \end{aligned} \quad (2)$$

We select a confidence interval of 95%. A $p - value \leqslant 0.05$ means that there is a statistical difference between the base application energy consumption (no adaptation engines included) and each application with a different adaptation solution (but without context-adaptation). As in the previous research question, we have performed a statistical power analysis for both case studies. The result is 1 for both for a significance value lower than 0.05. This shows that the sample size (133 for case study 1 and 120 for case study 2) is enough to discard a Type II error. The hypothesis contrast results show there is no statistical difference between the usage of a specific adaptation engine in 50% of the tests (Fig. 7). We observe a 2.51% increment in the energy consumption in the worst case. However, these values are negligible in comparison with the benefits obtained from the adaptation (Section 5.2.1).

Findings Results show that the overhead introduced by the adaptation engine is minimal (see Fig. 8). The overhead represents between the 0.58% and 2.51% (an increase in 1 to 4 Joules) of the energy consumption of the application in the first scenario, while the overhead in the second scenario represents between 0.43% and 1.39% (an increment between 3 and 9 Joules) of the energy consumption of the application. This means an increase of just 8.87 Joules in the worst case.

The Xposed engine (AE4) shows a minor energy cost overhead. These results can be explained due to the creation and management of the proxy object for each adaptable class in those adaptation engines based

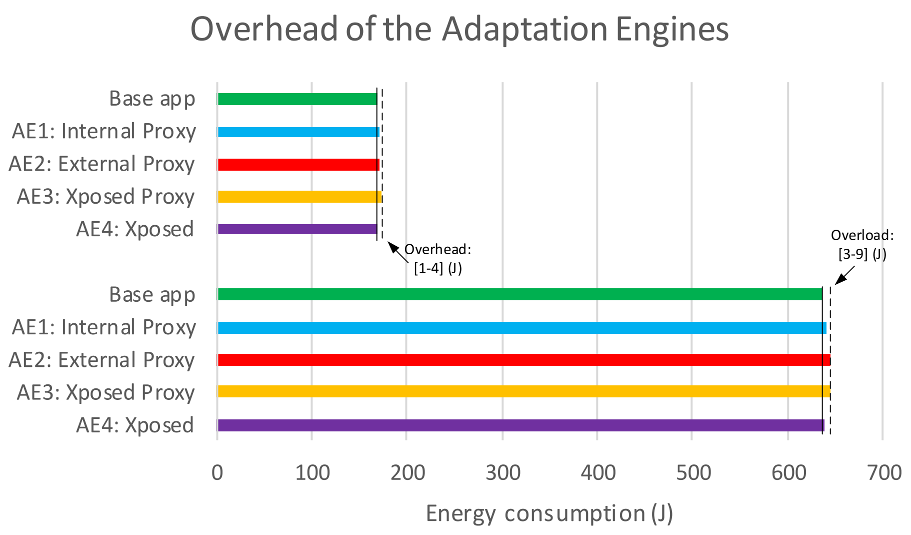

Fig. 8. Energy consumption overhead introduced by the adaptation engines.

on dynamic proxies (AE1, AE2 and AE3), but not AE4. Nevertheless, comparing the adaptation system's averages of the measures, no significant difference is found between the adaptation engines.

5.2.3. RQ3. Proposed engines scalability.

Research Question 3 Are the adaptation engines scalable? What scenarios could increase their energy consumption?

Motivation Applicable scenarios of our reconfiguration engines are very diverse, so it is impossible to evaluate all of them using single apps. There are some critical points, like the number of adaptable classes, adaptation alternatives or adaptation rules that can increase the energy consumption of the adaptation engines. In order to study these aspects, we evaluate the viability of using our solutions in different scenarios.

Experiments Each adaptation system has different critical points that could increase its energy cost (e.g., number of adaptable classes, adaptation alternatives, adaptation rules, methods, etc.), reducing the benefit obtained in applications' energy consumption. To evaluate the scalability of our proposal, we evaluate the energy consumption of our adaptation engines when the number of elements managed by adaptation engine vary. To achieve that, we provide a set of benchmark applications[7] (according to the AE being evaluated) that provide a user interface to configure the number of elements managed by the adaptation mechanisms. For solutions based on dynamic proxies (AE1 and AE2), the benchmark application allows the configuration of the number of adaptable classes, adaptation rules and adaptation alternatives because the adaptation is performed at runtime. For Xposed-based solutions (AE3 and AE4), the benchmark application configures the number of functionalities (methods) adapted by the framework because adaptations are done at load-time and no additional energy cost is required at runtime to perform the adaptation. All of them allow the generation of random user behaviors and are based on our first scenario, simplifying it in order to isolate the overhead caused by our adaptation engines, as much as possible. In any case, the experiments depend on the mobile devices' capabilities because of the hardware differences and limitations between devices.

Analysis A comparison between the averages of the energy consumption measurements of each configuration gives us information about the scalability and behavior of the adaptation engines. We use a graphical representation of the energy consumption averages to show the results.

Findings *Number of adaptable classes* Fig. 9 shows the result of increasing the number of adaptable and instantiated classes managed by

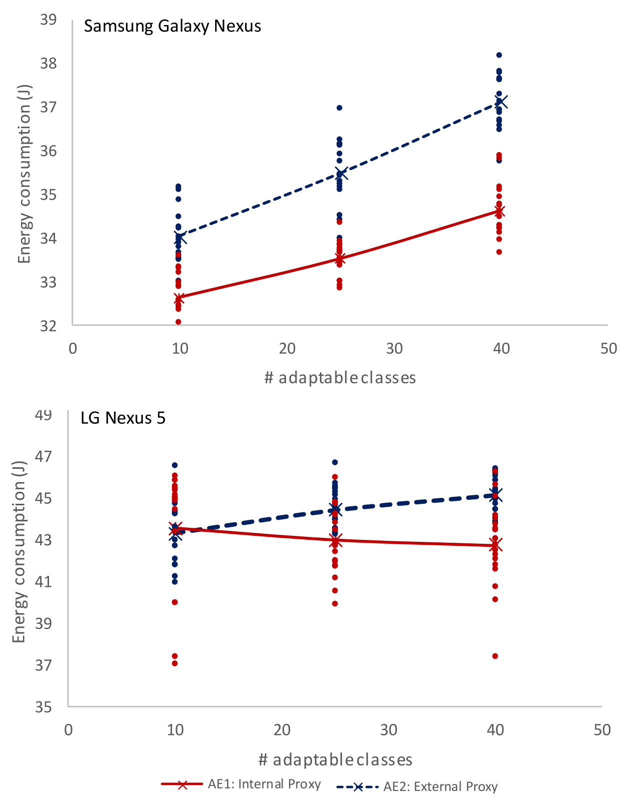

Fig. 9. Energy consumption in relation to the number of adaptable classes.

the application up to 40 (more detailed information at Table 1). To do this, the number of monitors and adaptable classes increases in a 1 to 1 relation. In the case of 40 adaptable classes managed by the application, there are 40 monitors instantiated to control the user behavior with the application. It is observed that the application energy consumption is barely increased. In our case, there is no increase in the energy consumption in the LG Nexus 5 device for the AE1 solution. In the case of adaptation engine AE2 (External Proxy), loading 15 external classes supposes around 1 Joule of overhead in the energy consumption of the application. In the Samsung Galaxy Nexus, 15 adaptable classes suppose an increment of around 1 Joule for the AE1 (Internal Proxy) and 1.5 Joules for the AE2 (External Proxy).

[7] Benchmark applications available at https://github.com/angelcvx/Benchmarks-AE.

Table 1
Energy consumption in relation to the number of adaptable classes.

		AE1				AE2			
	N° of adaptable classes	Mean (J)	Std	Median (J)	Duration (s)	Mean (J)	Std	Median (J)	Duration (s)
Samsung Galaxy Nexus	10	32.6	0.6	32.6	19.4	34.0	0.6	33.9	20.0
	25	33.5	0.4	33.6	19.7	35.5	0.8	35.5	20.5
	40	34.6	0.6	34.5	20.2	37.1	0.6	37.0	21.1
LG Nexus 5	10	43.6	2.9	45.0	18.8	43.3	1.5	43.6	19.3
	25	43.0	1.6	42.9	18.8	44.4	1.1	44.6	20.0
	40	42.8	2.0	43.1	18.8	45.1	1.0	45.3	20.7

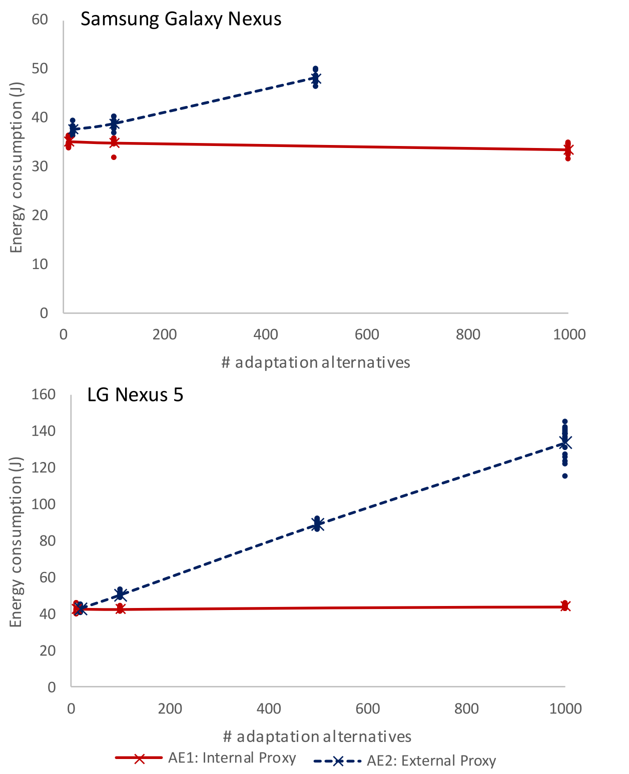

Fig. 10. Energy consumption in relation to the number of adaptation alternatives.

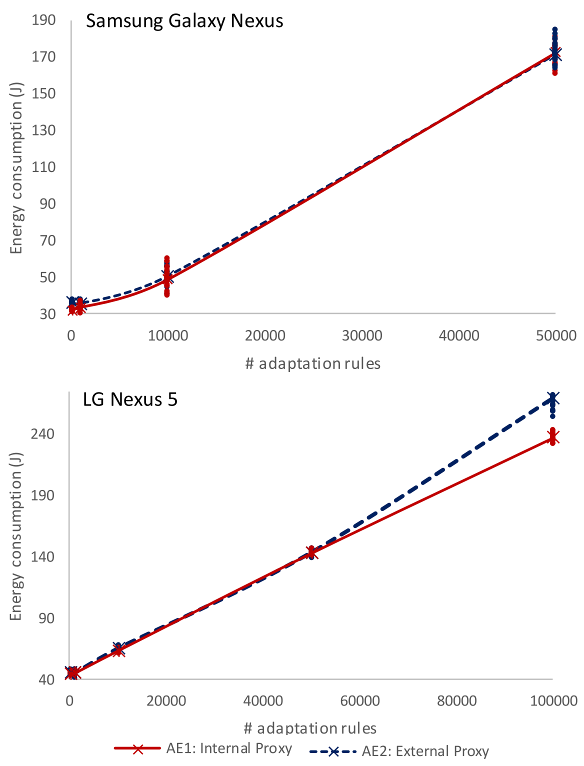

Fig. 11. Energy consumption in relation to the number of adaptation rules.

Number of adaptation alternatives The number of adaptation alternatives may affect the application energy consumption. This number will depend on the specific application size. This experiment is intended to show whether or not is possible to include our adaptation engine in an application with a high degree of variability.

The results show that in the case of the External Proxy solution (AE2), loading and managing a higher number of adaptation alternatives does not increase the energy consumption to a high degree (see Fig. 10 and Table 2). The number of classes to be loaded externally is limited by the device performance. In our case, this number is limited to 500 for Galaxy Nexus and to 1000 for the LG Nexus 5 device. In the case of the Internal Proxy solution (AE1), we even reach 1000 classes without a significant increase in the energy consumption. The results also show that the number of adaptation alternatives to be managed by the solutions is not a handicap. Nevertheless, the Internal Proxy solution allows the management of a higher number of adaptation alternatives without a negative impact on the applications' energy consumption.

Number of adaptation rules A critical point of a context-aware engine is the number of adaptation rules. This number is conditioned by the number of different configurations that an application can have. Fig. 11 and Table 3 show the energy consumption of the solutions based on Internal Proxy (AE1) and External Proxy (AE2) in relation to the number of adaptation rules they contain. It is observed that with more than 1000 adaptation rules, the system hardly increases its energy consumption. This number means that an application is checking 1000 different aspects to adapt itself to the context.

Number of methods changed in the same class with Xposed This test evaluates the number of methods modified in the same class by one Xposed module class. This analysis gives us information about the Xposed framework energy consumption associated with the number of functionalities (methods) changed. The number of adapted methods from the same Xposed module class was increased up to 32 without resulting in an increase in the application's energy consumption (Fig. 12 and Table 4).

5.3. Threats to validity

This section presents the threats to validity of the evaluation. The four threats are conclusion, internal, construct, and external validity [44].

Table 2
Energy consumption in relation to the number of adaptation alternatives.

	N° of adaptation alternatives	AE1 Mean (J)	AE1 Std	AE1 Median (J)	AE1 Duration (s)	AE2 Mean (J)	AE2 Std	AE2 Median (J)	AE2 Duration (s)
Samsung Galaxy Nexus	10	35.2	0.9	35.0	19.3	37.6	0.8	37.6	20.6
	100	34.9	0.9	35.0	19.3	38.8	0.9	39.0	22.3
	500	-	-	-	-	48.2	0.9	48.1	29.5
	1000	33.4	0.9	33.3	19.7	-	-	-	-
LG Nexus 5	10	42.9	2.3	42.4	18.7	43.1	1.2	43.3	18.9
	100	42.7	0.9	42.5	18.5	50.5	1.0	50.4	22.2
	500	-	-	-	-	89.2	1.6	89.4	35.3
	1000	44.0	0.8	44.0	18.6	133.9	7.8	136.6	60.6

Table 3
Energy consumption in relation to the number of adaptation rules.

	N° of adaptation rules	AE1 Mean (J)	AE1 Std	AE1 Median (J)	AE1 Duration (s)	AE2 Mean (J)	AE2 Std	AE2 Median (J)	AE2 Duration (s)
Samsung Galaxy Nexus	100	32.0	0.7	32.1	19.7	36.0	0.8	36.1	19.7
	1000	33.4	1.6	33.3	19.6	35.8	0.8	36.0	19.6
	10,000	48.7	6.0	49.0	26.2	50.8	4.8	53.0	26.2
	50,000	172.9	5.9	172.3	97.7	171.5	5.6	172.1	97.2
LG Nexus 5	100	44.6	0.9	44.6	18.7	45.6	1.2	45.6	18.9
	1000	45.2	1.2	45.4	18.9	45.4	2.6	46.0	18.9
	10,000	63.6	1.2	63.7	25.6	65.4	1.0	65.5	26.0
	50,000	143.3	1.6	143.0	63.9	143.5	2.0	143.5	63.9
	100,000	237.7	3.6	238.5	128.0	269.0	8.3	270.0	156.3

Table 4
Energy consumption in relation to the number of adapted methods by Xposed.

	N° of Xposed methods	Mean (J)	Std	Median (J)	Duration (s)
Samsung Galaxy Nexus	4	35.0	0.8	34.8	19.3
	8	34.2	1.1	34.1	19.0
	16	34.6	1.2	34.6	19.6
	32	35.0	1.6	34.6	19.3
LG Nexus 5	4	43.3	0.9	43.4	18.9
	8	42.9	1.3	42.7	18.8
	16	41.9	2.0	42.2	18.6
	32	42.4	0.6	42.5	18.5

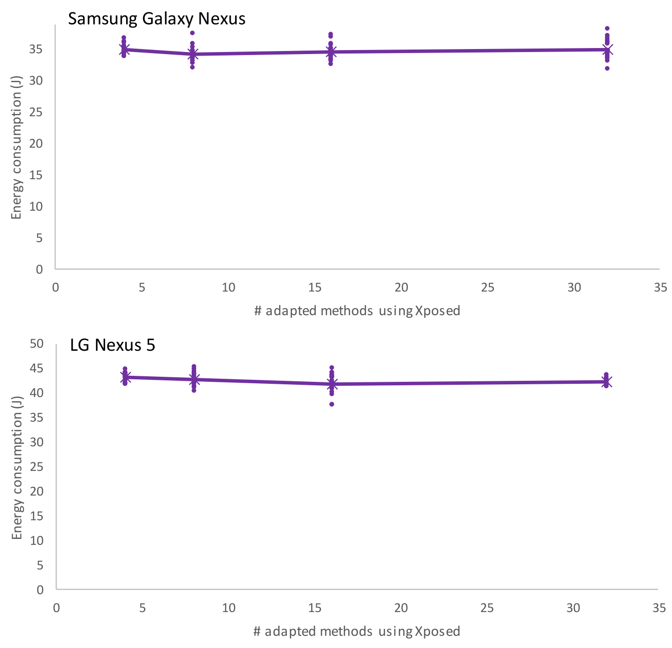

Fig. 12. Energy consumption in relation to the number methods changed by Xposed.

5.3.1. *Conclusion validity*

To evaluate the solutions presented, we have used two measurement tools: GreenScaler and Trepn Profiler (see Section 5.1.2). GreenScaler has been specifically designed for Samsung Galaxy devices. Trepn Profiler has been designed by Qualcomm, and therefore works better on devices with Qualcomm Snapdragon processors. The mobiles used here have been specifically chosen to provide reliable results with these measurement tools, and are accepted by the community as devices that report accurate energy consumption results. In any case, the tools only give us estimations of energy consumption, and to obtain more precise measurements we need to use hardware-based tools. Notwithstanding, although we performed only 20 measurements for each experiment and a greater number of measurements would improve the accuracy of the results, we are more concerned with comparing the results from different solutions than in the exact values of energy consumption.

5.3.2. *Internal validity*

In mobile devices, replication of experiments under the same conditions is not straightforward. Several system processes and applications can be running in the background, making use of different resources. So, the energy consumption of the application to be tested cannot be totally isolated from the rest of the system processes. To mitigate the uncontrolled factors, we undertook the measurements with the devices in the same conditions as explained in Section 5.1.3. Only the application to be tested was in the foreground during the experiment. In addition, the flight mode was enabled, and application updates were disabled.

5.3.3. *Construct validity*

In order to compare the solutions presented with each other, we defined and replicated the same user behavior for all experiments. Normally, a user's interactions with an application cannot be predicted. A set of random interactions with the application, emulating different users' interactions could also be applied.

5.3.4. *External validity*

We used two mobile phones with different software and hardware architectures to undertake the measurements. Although we observe the same tendency in both mobiles, the results cannot be generalized from these devices to others. The mobiles chosen confirm that our proposal works in Dalvik and ART Android virtual machines.

We have used two representative scenarios which are followed by hundreds of applications. However, there are other types of applications, which could be interesting to evaluate and dynamically adapt.

6. Related work

This section discusses the related work on dynamic reconfiguration and energy optimization in mobile devices, and concretely in Android.

In mobile applications, despite the fact that adaptations are frequently driven by non-functional requirements such as improving the application's performance [10,12,15], increasing the failure tolerance [15], or improving the user experience [3,32,48], among others; reducing the energy consumption is the most important objective [3,10,45,46,49,53], as the most recent energy optimization approaches focus on [10,50,53,54].

A dynamic adaptation system is characterized by different dimensions or characteristics such as the primary reconfiguration goal (e.g., improving performance, energy saving), the type of adaptation engine (application-oriented vs system-oriented) [10,13], or the context that is monitored (e.g., device status, environment, user's interactions) [3]. Here we have classified publications by their adaptation technique according to the different approaches introduced in Section 2. Table 5 summarizes and compares the main characteristics of existing papers.

First, some approaches have investigated the reconfiguration based on code offloading [12,45,46,48,53], which delegates complex decision making and reconfiguration management to a server. For instance, MAsCOT [12] uses static and dynamic Decision Networks (DNs) to determine the best configuration. Code offloading is usually applied in combination with other approaches such as dynamic proxies [46] or the use of the OSGi platform [12,45,48]. All these approaches [12,45,46,48,53] have in common that they monitor the device status (e.g., battery level, CPU frequency) and the environment (e.g., location, noise level, lighting), but they do not consider the user's interactions with the application. For instance, Dynamix [45] uses OSGi where context information is provided using modules created by the community and accessible to the user, who decides which of them he/she wants to use. Dynamix provides a configurable context firewall that allows managing the contextual information available to each application. Another interesting approach that runs on top of the OSGi platform is the MUSIC middleware of Rouvoy et al. [48]. In MUSIC, a utility function, which refers to the user's overall satisfaction, is defined to decide which is the most appropriate configuration for the current execution context (e.g., increase the sound level if the user is in a noisy place). MUSIC takes into account a predefined user profile with the user's preferences, but it does not consider the user's behavior data as part of the monitored context.

There are other adaptation frameworks specific to Android, such as the Xposed framework used in this paper, morphone.OS [3], ARTDroid [19], or CRITiCAL [32]. Nacci et al. define morphone.OS [3], a framework for monitoring the context of an Android smartphone. They define three types of context aspects: environment (ambient light, noise, etc), user behavior and the mobile (battery status, CPU frequency, etc). In [3], the authors distinguish between configuration changes and application changes. However, they do not determine how these changes are carried out. Our approach takes into account the environment, device and user context, although in this paper we have focused on the user's interactions. ARTDroid is a framework for hooking virtual-method calls like Xposed, but ARTDroid is oriented towards directly modifying the app's virtual-memory tampering with ART internal representation of Java classes and methods, instead of code modifications of both applications and system. However, as Xposed, ARTDroid requires the new functionality to be introduced at load-time, so changes need to be predefined before the execution of the application. CRITiCAL [32] is based on a model-driven middleware that includes a domain specific language for modeling the contextual information and adaptation rules, transforming them into executable code. The middleware provides the code required to deal with sensors and to react and execute functionality according to contextual rules.

Specific programming techniques have been used to reconfigure mobile applications, as for example reflection [6], polymorphic methods [10], dynamic proxies [46], Aspect-Oriented Programming [47], and Context-Oriented Programming [15]. For example, Casquina et al. propose Cosmapek [6], an adaptive deployment infrastructure to adapt the applications, using reflection, in response to errors that may occur in the context in which the application is located. CAreDroid is a framework [10] in which context-aware methods are defined in application source code, the mapping of methods to context is defined in configuration files and context-monitoring and method replacement are performed at runtime. WeaveDroid [47] and JCop [15] are extensions of Java that apply, respectively, Aspect-Oriented Programming (AOP) and Context-Oriented Programming (COP) to the Android environment. A disadvantage of these approaches [6,10,15,46,47] is that they have to be integrated as part of the Dalvik VM, since they are relatively old approaches (prior to the ART VM release in 2015–2016).

Multiple ad-hoc solutions have also been proposed for reconfiguring Android applications [3,10,49,53]. For instance, Pascual et al. [49] define MODAGAME, a multi-objective reconfiguration approach that uses genetic algorithms. In this case, the algorithms modify the configuration of the device, such as the network connectivity, Bluetooth status or sound quality, but there is no code change in the applications themselves.GEMMA [53] also uses genetic algorithms in combination with power models and color theory to optimize the colors used by Android applications, and thereby, reduce the energy consumption of the displays. As shown in Table 5, our approach uses different adaptation methods: virtual-method hooking, and dynamic proxies, to modify the application functionality at loadtime and runtime, respectively. Moreover, our solution can also be used to modify the operating system's resources by encoding the operating system calls in polymorphic methods.

A general disadvantage of adaptation engines for Android is that they are highly dependent on the Android version. Most of the existing approaches are tested only in one of Android's runtime environments such as Dalvik VM [3,6,10,15,32,45,46,48], and the adaptation engine needs to be modified to be applied over a more recent execution environment (e.g., ART). Our adaptation engines have been tested in both Dalvik and ART VMs. In fact, one of our solutions, based on dynamic proxies, does not depend on the Android environment, and thus, it can be applied to other mobile operating systems.

Recently, current approaches [40,50–52,54] have tried to optimize the energy consumption of the mobile applications following a refactoring approach, without an explicit dynamic reconfiguration of the application. In contrast to most of the classical reconfiguration approaches that adapt the application at runtime, these approaches refactor the code of the application at compile time.For example, Banerjee et al. [50] they use a refactoring technique that relies on a set of energy-efficiency guidelines to encode the optimal usage of energy-intensive hardware resources in an Android application, reducing the energy consumption of the applications by between 3% to 29%.

Refactoring can be applied to different level of abstraction: from low level implementation details [51], to the design level [52,54] or to the architectural level [40]. For instance, Sahin et al. [51] explore the

Table 5
Comparison of dynamic reconfiguration and energy optimization approaches in Android mobile devices.

Approach	Year	Adaptation approach	Adaptation type[a]	Monitoring context[b]	Reconfiguration goal	Runtime environment	Dynamicity
Dynamix [45]	2011	OSGi platform	APP	DEV,ENV	Energy saving	Dalvik	runtime
JCop [15]	2011	Context-Oriented Programming	APP	DEV	Performance, failure tolerance	Dalvik	compilation
Artail et al. [46]	2012	Dynamic proxies, code offloading	APP	-	Energy saving	Dalvik	runtime
WeaveDroid [47]	2012	Aspect-Oriented Programming	APP	-	-	Dalvik	compilation
MUSIC [48]	2013	OSGi platform	SYS	DEV,ENV	User experience	Dalvik	runtime
morphone.OS [3]	2013	Ad-hoc solution (custom framework)	SYS,APP	DEV	Energy saving, user experience	Dalvik	compilation
CRITiCAL [32]	2015	Virtual-method hooking	APP	DEV,ENV	User experience	Dalvik	runtime
MODAGAME [49]	2015	Ad-hoc solution (genetic algorithms)	APP	DEV	Energy saving	Dalvik, ART	runtime
MAsCOT [12]	2016	OSGi platform, code offloading	APP	DEV	Performance	Dalvik, ART	runtime
ARTDroid [19]	2016	Virtual-method hooking	APP	USER	Malware analysis, policy enforcement	ART	loading
Cosmapek [6]	2016	Java reflection	APP	USER	Failure tolerance	Dalvik	runtime
CAreDroid [10]	2016	Ad-hoc solution (polymorphic methods)	SYS,APP	DEV,ENV, USER	Reducing SLOC, execution time, energy saving	Dalvik	runtime
Banerjee et al. [50]	2016	Ad-hoc solution (refactoring framework)	APP	DEV,USER	Energy saving	-	compilation
Sahin et al. [51]	2016	Refactoring (performance tips)	APP	-	Energy saving	Dalvik, ART	compilation
Hasan et al. [52]	2016	Refactoring (Java collections)	APP	-	Energy saving	Dalvik	compilation
GEMMA [53]	2017	Ad-hoc solution (genetic algorithms, code offloading)	APP	DEV	Energy saving, increasing contrast, improving colors	-	loading
EARMO [54]	2018	Ad-hoc solution (genetic algorithms)	APP	DEV,USER	Energy saving, refactoring recommendations	Dalvik, ART	compilation
GreenBundle [40]	2019	Patterns refactoring (bundling and dropping events)	APP	-	Energy saving	Dalvik, ART	compilation
Cruz et al. [55]	2019	Refactoring	SYS,APP	DEV,ENV, USER	Energy saving	-	**compilation, runtime**
Our approach	**2019**	**Dynamic proxies, Virtual-method hooking**	**SYS,APP**	**DEV,ENV, USER**	**Energy saving**	**Dalvik, ART**	**loading, runtime**

[a] SYS: system-oriented (hardware resources: display, network, sensors,...). APP: application-oriented (software resources: functionality, code) [10,13].
[b] DEV: device status (battery level, CPU frequency). ENV: environment (location, noise level, lighting,...). USER: user behavior (interactions, user data,...) [3].

energy impacts of performance tips, and they demonstrate that those changes in the code are unlikely to impact the energy usage in a statistically significant manner. However, Hasan et al. [52] measure the energy consumption of the Java collections classes by creating energy profiles, and demonstrate that choosing an energy efficient collection can improve energy consumption of Android application by as much as 38%. Similar analysis of energy impact have been performed over other specific functionalities such as logging [39], compression [56], the HTTP protocol [57], testing frameworks [58], or the operationalization of quality attributes [7]. Raising the abstraction level of the refactoring, Morales et al. [54] analyze the impact of anti-patterns on the design of Android applications, and propose EARMO, an anti-pattern correction approach based on evolutionary multi-objective techniques that accounts for energy consumption when refactoring mobile anti-patterns. Chowdhury et al. presents GreenBundle [40], an empirical study on the energy impact of bundling and dropping strategies applied over architectural patterns. They reduce the energy consumption of applications by 30% when refactoring classical Model-View-Controller (MVC) architectures into bundled Model-View-Presenter (MVP) architectures. One advantage of the refactoring approaches is that they can be applied independently of the execution environment or Android version because they do not introduce new technology. Despite the great improvement in energy efficiency shown by the refactoring techniques that can improve the energy consumption up to 30% [40,50], a disadvantage of these approaches is that they cannot consider the dynamic behaviour since they are applied at compile time, and thus they waste energy saving opportunities derived by a certain user behaviour with the application, something that our approach does consider.

Finally, Cruz et al. [55] identify a catalog of design practices to improve the energy efficiency of mobile applications where some of the proposed refactoring can be applied at runtime and affect both the application and the system. For example, they propose to increasing the retry interval delay when the connection to a resource fails a number of times; activate the power save mode; or using WiFi over Cellular, among other solutions. However, authors have not yet implemented an automated refactoring tool for these patterns.

7. Conclusions and future work

We have proposed four adaptation engines for dynamically adapting Android applications with the goal of decreasing the energy consumption based on users' interactions with the application.

Through our results, we have demonstrated that the proposed adaptation engines do not increase the energy consumption in relation to the benefits of the adaptation, which can reduce the energy consumption by up to 20% in applications that follow our scenarios. Additionally, our approach is compatible with all current Android versions.

As for future work, we will consider the evaluation and comparison of other existing adaptation engines such as the OSGi platform, Aspect-Oriented Programming, or Context-Oriented Programming, as the overhead in energy consumption introduced by them is unknown.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the projects Magic P12-TIC1814, HADAS TIN2015-64841-R (co-financed by FEDER funds), TASOVA MCIU-AEI TIN2017-90644-REDT, MEDEA RTI2018-099213-B-I00 (co-funded by FEDER funds) and LEIA UMA18-FEDERJA-157 (co-funded by FEDER funds); and the post-doctoral plan of the University of Málaga.

References

[1] L. He, G. Meng, Y. Gu, C. Liu, J. Sun, T. Zhu, Y. Liu, K.G. Shin, Battery-aware mobile data service, IEEE Trans. Mob. Comput. 16 (6) (2017) 1544–1558, doi:10.1109/TMC.2016.2597842.
[2] D. Li, S. Hao, J. Gui, W.G.J. Halfond, An empirical study of the energy consumption of android applications, in: Software Maintenance and Evolution, 2014, pp. 121–130, doi:10.1109/ICSME.2014.34.
[3] A.A. Nacci, M. Mazzucchelli, M. Maggio, A. Bonetto, D. Sciuto, M.D. Santambrogio, morphone.OS: context-awareness in everyday life, in: Digital System Design (DSD), 2013, pp. 779–786.
[4] A. Rice, S. Hay, Measuring mobile phone energy consumption for 802.11 wireless networking, Pervasive Mob. Comput. 6 (6) (2010) 593–606, doi:10.1016/j.pmcj.2010.07.005.
[5] H. Joe, J. Kim, J. Lee, H. Kim, Output-oriented power saving mode for mobile devices, Futur. Gener. Comput. Syst. 72 (2017) 49–64, doi:10.1016/j.future.2016.05.012.
[6] J.C. Casquina, J.D.A.S. Eleuterio, C.M.F. Rubira, Adaptive deployment infrastructure for Android applications, in: 12th European Dependable Computing Conference (EDCC), 2016, pp. 218–228, doi:10.1109/EDCC.2016.25.
[7] J.-M. Horcas, M. Pinto, L. Fuentes, Variability models for generating efficient configurations of functional quality attributes, Inform. Softw. Technol. 95 (2018) 147–164, doi:10.1016/j.infsof.2017.10.018.
[8] Y. Choi, R. Ha, H. Cha, Fully automated OLED display power modeling for mobile devices, Pervasive Mob. Comput. 50 (2018) 41–55, doi:10.1016/j.pmcj.2018.07.006.
[9] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, D.-K. Baik, Ringa: design and verification of finite state machine for self-adaptive software at runtime, Inform. Softw. Technol. 93 (2018) 200–222, doi:10.1016/j.infsof.2017.09.008.
[10] S. Elmalaki, L.F. Wanner, M.B. Srivastava, CAreDroid: adaptation framework for android context-aware applications, GetMobile 20 (2) (2016) 35–38, doi:10.1145/3009808.3009820.
[11] G.G. Pascual, R.E. Lopez-Herrejon, M. Pinto, L. Fuentes, A. Egyed, Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications, J. Syst. Softw. 103 (2015) 392–411, doi:10.1016/j.jss.2014.12.041.
[12] N.Z. Naqvi, J. Devlieghere, D. Preuveneers, Y. Berbers, Mascot: Self-adaptive opportunistic offloading for cloud-enabled smart mobile applications with probabilistic graphical models at runtime, in: 49th Hawaii International Conference on System Sciences (HICSS), 2016, pp. 5701–5710, doi:10.1109/HICSS.2016.705.
[13] S.K. Datta, C. Bonnet, N. Nikaein, Self-adaptive battery and context aware mobile application development, in: 2014 International Wireless Communications and Mobile Computing Conference (IWCMC), 2014, pp. 761–766, doi:10.1109/IWCMC.2014.6906452.
[14] rovo89, Xposed framework, (http://repo.xposed.info/). Online; accessed 13 May2019.
[15] C. Schuster, M. Appeltauer, R. Hirschfeld, Context-oriented programming for mobile devices: JCop on Android, in: 3rd International Workshop on Context-Oriented Programming, in: COP, 2011, pp. 5:1–5:5, doi:10.1145/2068736.2068741.
[16] S. Bohez, E. De Coninck, T. Verbelen, B. Dhoedt, Androsgi: bringing the power of OSGi to Android, in: Proceedings of the on Eclipse Technology eXchange, 2015, pp. 1–6.
[17] M. Kuna, H. Kolaric, I. Bojic, M. Kusek, G. Jezic, Android/OSGi-based machine-to-machine context-aware system, in: 11th International Conference on Telecommunications, 2011, pp. 95–102.
[18] M.-C. Chen, J.-L. Chen, T.-W. Chang, Android/osgi-based vehicular network management system, Comput. Commun. 34 (2) (2011) 169–183. Special Issue: Open network service technologies and applications doi: 10.1016/j.comcom.2010.03.032.
[19] V. Costamagna, C. Zheng, Artdroid: A virtual-method hooking framework on android art runtime., in: IMPS@ ESSoS, 2016, pp. 20–28.
[20] Y. Zhang, Y. Li, T. Tan, J. Xue, RIPPLE: reflection analysis for android apps in incomplete information environments, Softw. Pract. Exper. 48 (8) (2018) 1419–1437, doi:10.1002/spe.2577.
[21] Y. Hassoun, R. Johnson, S. Counsell, Applications of dynamic proxies in distributed environments, Softw. Pract. Exper. 35 (1) (2004) 75–99, doi:10.1002/spe.629.
[22] T. Van Cutsem, M.S. Miller, Proxies: design principles for robust object-oriented intercession apis, in: ACM Sigplan Notices, 45, ACM, 2010, pp. 59–72.
[23] G. Fourtounis, G. Kastrinis, Y. Smaragdakis, Static analysis of java dynamic proxies, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, in: ISSTA 2018, ACM, New York, NY, USA, 2018, pp. 209–220, doi:10.1145/3213846.3213864.
[24] A. Bartel, J. Klein, Y. Le Traon, M. Monperrus, Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot, in: International Workshop on State of the Art in Java Program Analysis, in: SOAP, 2012, pp. 27–38, doi:10.1145/2259051.2259056.
[25] A.G. de Prado, G. Ortiz, J. Boubeta-Puig, D. Corral-Plaza, Air4people: a smart air quality monitoring and context-aware notification system, J. UCS 24 (7) (2018) 846–863.
[26] J. Gedeon, N. Himmelmann, P. Felka, F. Herrlich, M. Stein, M. Mühlhäuser, vstore: A context-aware framework for mobile micro-storage at the edge, in: Mobile Computing, Applications, and Services, Springer International Publishing, Cham, 2018, pp. 165–182.
[27] K. Lee, J. Lee, Y. Yi, I. Rhee, S. Chong, Mobile data offloading: how much can wifi deliver? IEEE/ACM Trans. Netw. 21 (2) (2013) 536–550, doi:10.1109/TNET.2012.2218122.
[28] D.K. Kim, S. Bohner, Dynamic reconfiguration for java applications using AOP, in: IEEE SoutheastCon, 2008, pp. 210–215, doi:10.1109/SECON.2008.4494287.
[29] S. Arzt, S. Rasthofer, E. Bodden, Instrumenting android and java applications as easy as abc, in: Runtime Verification, 2013, pp. 364–381.
[30] S. González, K. Mens, M. Colacioiu, W. Cazzola, Context traits: Dynamic behaviour adaptation through run-time trait recomposition, in: International Conference on Aspect-oriented Software Development, in: AOSD, 2013, pp. 209–220, doi:10.1145/2451436.2451461.
[31] B. Han, Y. Zhao, C. Zhu, Q. Zeng, Towards runtime adaptation in context-oriented programming, in: International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE), 2013, pp. 201–208, doi:10.1109/ICEEE.2013.6676051.
[32] P.A. d. S. Duarte, F.M. Barreto, F.A. d. A. Gomes, W.V. d. Carvalho, F.A.M. Trinta, CRITiCAL: A configuration tool for context aware and mobile applications, in: Computer Software and Applications Conference, 2, 2015, pp. 159–168, doi:10.1109/COMPSAC.2015.91.
[33] P. Arcaini, E. Riccobene, P. Scandurra, Modeling and analyzing MAPE-K feedback loops for self-adaptation, in: Software Engineering for Adaptive and Self-Managing Systems, in: SEAMS, 2015, pp. 13–23.
[34] S.A. Chowdhury, S. Borle, S. Romansky, A. Hindle, Greenscaler: training software energy models with automatic test generation, Empiri. Softw. Eng. 24 (4) (2019) 1649–1692, doi:10.1007/s10664-018-9640-7.
[35] S.A. Chowdhury, A. Hindle, Greenoracle: Estimating software energy consumption with energy measurement corpora, in: Proceedings of the 13th International Conference on Mining Software Repositories, in: MSR '16, ACM, New York, NY, USA, 2016, pp. 49–60, doi:10.1145/2901739.2901763.
[36] L. Li, J. Gao, M. Hurier, P. Kong, T.F. Bissyandé, A. Bartel, J. Klein, Y.L. Traon, Androzoo + +: collecting millions of android apps and their metadata for the research community, CoRR abs/1709.05281 (2017).
[37] M.A. Hoque, M. Siekkinen, K.N. Khan, Y. Xiao, S. Tarkoma, Modeling, profiling, and debugging the energy consumption of mobile devices, ACM Comput. Surv. 48 (3) (2015) 39:1–39:40, doi:10.1145/2840723.
[38] I. Ayala, M. Amor, L. Fuentes, An energy efficiency study of web-based communication in android phones, Scientific Programming 2019 (2019).
[39] S.A. Chowdhury, S.D. Nardo, A. Hindle, Z.M.J. Jiang, An exploratory study on assessing the energy impact of logging on android applications, Empiri. Softw. Eng. 23 (3) (2018) 1422–1456, doi:10.1007/s10664-017-9545-x.
[40] S.A. Chowdhury, A. Hindle, R. Kazman, T. Shuto, K. Matsui, Y. Kamei, Greenbundle: An empirical study on the energy impact of bundled processing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 1107–1118, doi:10.1109/ICSE.2019.00114.
[41] A. Hindle, A. Wilson, K. Rasmussen, E.J. Barlow, J.C. Campbell, S. Romansky, GreenMiner: A hardware based mining software repositories software energy consumption framework, in: Working Conference on Mining Software Repositories, in: MSR, 2014, pp. 12–21, doi:10.1145/2597073.2597097.
[42] S.S. Shapiro, M.B. Wilk, M.H.J. Chen, A comparative study of various tests for normality, J. Am. Stat. Assoc. 63 (324) (1968) 1343–1372, doi:10.1080/01621459.1968.10480932.
[43] D. Rey, M. Neuhäuser, Wilcoxon-Signed-Rank Test, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1658–1659. 10.1007/978-3-642-04898-2_616.
[44] D.T. Campbell, J.C. Stanley, Experimental and quasi-experimental designs for research, Ravenio Books, 2015.
[45] D. Carlson, A. Schrader, A wide-area context-awareness approach for Android, in: Information Integration and Web-based Applications and Services, in: iiWAS, 2011, pp. 383–386, doi:10.1145/2095536.2095610.
[46] H. Artail, K. Fawaz, A. Ghandour, A proxy-based architecture for dynamic discovery and invocation of web services from mobile devices, IEEE Trans. Serv. Comput. 5 (1) (2012) 99–115, doi:10.1109/TSC.2010.49.
[47] Y. Falcone, S. Currea, Weave droid: Aspect-oriented programming on Android devices: Fully embedded or in the cloud, in: International Conference on Automated Software Engineering, in: ASE, 2012, pp. 350–353, doi:10.1145/2351676.2351744.

[48] J. Floch, C. FrÃ, R. Fricke, K. Geihs, M. Wagner, J. Lorenzo, E. Soladana, S. Mehlhase, N. Paspallis, H. Rahnama, P. Ruiz, U. Scholz, Playing MUSIC building context-aware and self-adaptive mobile applications, Softw. Pract. Exper. 43 (3) (2013) 359–388, doi:10.1002/spe.2116.

[49] G.G. Pascual, M. Pinto, L. Fuentes, Self-adaptation of mobile systems driven by the common variability language, Futur. Gener. Comput. Syst. 47 (2015) 127–144.

[50] A. Banerjee, A. Roychoudhury, Automated re-factoring of android apps to enhance energy-efficiency, in: International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2016, pp. 139–150, doi:10.1109/MobileSoft.2016.038.

[51] C. Sahin, L. Pollock, J. Clause, From benchmarks to real apps: exploring the energy impacts of performance-directed changes, J. Syst. Softw. 117 (2016) 307–316, doi:10.1016/j.jss.2016.03.031.

[52] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, A. Hindle, Energy profiles of java collections classes, in: Proceedings of the 38th International Conference on Software Engineering, in: ICSE '16, ACM, New York, NY, USA, 2016, pp. 225–236, doi:10.1145/2884781.2884869.

[53] M. Linares-Vásquez, C. Bernal-Cárdenas, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, Gemma: Multi-objective optimization of energy consumption of guis in android apps, in: International Conference on Software Engineering Companion (ICSE-C), 2017, pp. 11–14, doi:10.1109/ICSE-C.2017.10.

[54] R. Morales, R. Saborido, F. Khomh, F. Chicano, G. Antoniol, Earmo: an energy-aware refactoring approach for mobile apps, IEEE Transactions on Software Engineering 44 (12) (2018) 1176–1206, doi:10.1109/TSE.2017.2757486.

[55] L. Cruz, R. Abreu, Catalog of energy patterns for mobile applications, Empiri. Softw. Eng. 24 (4) (2019) 2209–2235, doi:10.1007/s10664-019-09682-0.

[56] J.M. Horcas, M. Pinto, L. Fuentes, Context-aware energy-efficient applications for cyber-physical systems, Ad Hoc Netw. 82 (2019) 15–30, doi:10.1016/j.adhoc.2018.08.004.

[57] S.A. Chowdhury, V. Sapra, A. Hindle, Client-side energy efficiency of http/2 for web and mobile app developers, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 1, 2016, pp. 529–540, doi:10.1109/SANER.2016.77.

[58] L. Cruz, R. Abreu, Measuring the energy footprint of mobile testing frameworks, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, in: ICSE '18, ACM, New York, NY, USA, 2018, pp. 400–401, doi:10.1145/3183440.3195027.