Pervasive and Mobile Computing 39 (2017) 214–230

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

An empirical analysis of energy consumption of cross-platform frameworks for mobile development

Matteo Ciman *, Ombretta Gaggi

Department of Mathematics, University of Padua, Padua, Italy

ARTICLE INFO

Article history:
Received 12 July 2016
Received in revised form 4 October 2016
Accepted 21 October 2016
Available online 26 October 2016

Keywords:
Energy consumption
Mobile development
Performance measurement
Web technologies
Cross-platform frameworks

ABSTRACT

The increasing fragmentation of mobile devices market has created the problem of supporting all the possible mobile platforms to reach the highest number of potential users. One possible solution is to use cross-platform frameworks, that let develop only one application that is then deployed to all the supported target platforms. Currently available cross-platform frameworks follow different approaches to deploy the final application, and all of them has *pros* and *cons*. In this paper, we evaluate and compare together the current frameworks for cross-platform mobile development considering one of the most important aspect when dealing with mobile devices: energy consumption. Our analysis shows and measure how the adoption of cross-platform frameworks impacts energy consumption with respect to the native mobile development, identifies which are the most consuming tasks, and tries to define a final rank among all the different approaches. Moreover, we highlight future development necessary to improve performances of cross-platform frameworks to reach native development performances.

1. Introduction

Smartphones are rapidly becoming more and more present in everyday life of users. Thanks to their increasing computing capabilities and an ample set of different sensors, e.g., accelerometer, barometer, environmental thermometer, etc., smartphones play both the role of mobile workstations and of augmented devices, able to sense the environment and monitor user activities. Smartphones can be used for context awareness [1], user activity recognition [2,3], health monitoring [4], etc.

The variety of smartphones available on the market, in terms of, for example, models, vendors, cost, target users and available sensors, has accelerated the diffusion of these devices, since they reach all user needs. All these devices do not share the same operating system [5], so, if market fragmentation introduced by vendors is a positive aspect from the consumer point of view, reducing the final cost for the users, it is a huge problem for developers.

Developers of mobile applications aim for a, as wide as possible, set of target users, i.e., customers. Considering a business application that has to be paid, a wider set of paying customers means higher earnings. Moreover, since several applications have medical purposes, e.g., ambulatory treatment of particular diseases [6–8], excluding a set of possible patients depending on the operating system of their smartphone is a strong limitation.

Since all the operating systems currently used on mobile devices, i.e., Android, Apple, Windows Phone, Blackberry, etc., do not share any API or IDE or programming language, mobile development imposes a critical decision. Given that developers aim for the highest number of users, the choice to support only one particular mobile operating system, discarding the others,

* Corresponding author.
E-mail addresses: Matteo.Ciman@unige.ch (M. Ciman), gaggi@math.unipd.it (O. Gaggi).

http://dx.doi.org/10.1016/j.pmcj.2016.10.004
1574-1192/

can dramatically reduce the target users of the application. On the other hand, supporting all the available operating systems deeply increases development costs, in terms of required time, people working on the application, required programming skills, etc.

To solve this problem, in the last few years, particular attention has been posed to frameworks for cross-platform mobile development. Many differences exist, and will be discussed in Section 3, but the underlying idea is to develop the application only once, using a framework specific language, e.g., Javascript or C++, and to deploy this application to all the operating systems supported by the framework. It is clear that, in this way, the required development time, skills and costs are drastically reduced.

Several authors analyzed these frameworks and highlighted the main differences. They used different criteria, as the supported devices and operating system, native APIs, accessibility of the created application, native user interface, etc. [9–16]. Considering all these analysis, one of the most important aspect when dealing with mobile computing and devices is completely missing, which is the main motivation of this work: the evaluation of *energy consumption* issues.

Energy consumption is one of the most important aspects that has to be considered when dealing with mobile development and applications [17–20]. In fact, if we consider for example the definition of ubiquitous computing by Mike Weiser [21], one of the fundamental aspects is that technology must be invisible. Smartphones can be considered part of the so-called *invisible technology*, since they are always present in individuals' every day life. But this *invisible technology* can become *visible* and even annoying if it requires to recharge the device several times during the day, interfering with people's habits and lifestyle. Moreover, Wilke et al. [22] have shown that mobile applications which drain device's battery are soon rejected by the users.

The aim of this study is to analyze currently available cross-platform frameworks and to measure how their architecture impact energy consumption. The proposal and development of a new cross-platform framework is out of the scope of our research, but combining together experimental results and cross-platform framework architectures, we will highlight critical aspects on which future development should focus to increase framework performances.

The first result of this paper is the testbed applications used in the experiments.[1] We measured the energy consumed by a set of very simple applications which cover all the features available in the most common applications. We created therefore two complete sets of applications (one for the Apple platform and one for the Android platform) which can be used as a benchmark to measure and compare performance of cross-platform frameworks. In particular, we measured energy consumed while retrieving and showing (or not) data from different sensors. Despite the results related to energy consumption, the developed applications can be used to analyze other interesting data about frameworks performance, e.g., response time, memory/disk usage, etc.

We used an external power monitor during the experiments, the Monsoon PowerMonitor, since it avoids problems caused by possible unknown and not measurable overhead, that could be generated by applications running in background on the smartphone and monitoring energy consumption, e.g., PowerScope [11].

As discussed in Section 3, cross-platform frameworks for mobile development are very different in the way they produce the final application. Indeed, cross-platform applications can be very different from native applications. The main contribution of this paper is to investigate how the choice of a cross-platform framework affects energy consumption of the final application, comparing applications developed with different frameworks and native applications developed from scratch. In particular, we investigated the consumption of retrieving data from different sensors, e.g., accelerometer, light sensor, GPS, etc., since they are becoming very important in many mobile applications, and the consumption of updating (or not) the user interface.

The analysis identifies which are the most consuming tasks, and defines a final rank among all the different approaches. Moreover, both Android and Apple devices were considered, and several interesting results showed differences in performances of the same framework in the two different environments.

In particular, the main results showed by this analysis are:

- applications developed using a cross-platform framework lead to an higher energy consumption, even if the framework generates real native applications;
- to update the user interface represents the most expensive task, and it is the main cause of the increased energy consumption;
- the same framework and application can have different performances depending on the platform where the application is deployed, meaning that it is not possible to derive a general ranking of the considered frameworks and approaches among all the mobile platforms;
- the update frequency of data retrieved by sensors and their visualization heavily affects energy consumption, meaning that it is really important to take into consideration this aspect when dealing with sensor data.

However, in this paper we do not assert that the development of a *good* cross-platform framework is not possible, but current development of these frameworks is only at the beginning, and energy consumption improvement is a milestone for future development. This paper makes a step further in this direction, identifying which are the components, according to each particular class of framework, that have the highest impact on energy consumption, and therefore need to be improved.

[1] Test applications are freely available at https://github.com/wizard88mc/EnergyConsumptionCrossPlatformFrameworks.

Preliminary ideas about the analysis of power consumption due to the introduction of frameworks for cross-platform development appeared in [23], but this paper contains a very limited discussion: the paper does not consider the Apple platform but only one Android device, and it tested only Phonegap and Titanium frameworks.

The paper is organized as follows: Section 2 explores the related works and the state of the art, Section 3 describes cross-platform frameworks classification, highlighting *pros* and *cons* of each approach. Section 4 describes experiments setup, whose results are reported in Section 5. Section 6 discusses the results obtained with the experiments and proposes some guideline for the development of more efficient frameworks. We finally conclude in Section 7.

2. Related works

Many research papers [12,13,10,14,24] analyzed the development of mobile applications based on cross-platform frameworks. Heitkotter et al. [10] evaluated frameworks considering licensing, documentation and support, learning success and the possibility to customize user interfaces. Palmieri et al. [14] proposed a set of evaluation criteria, containing the programming environment and the APIs provided by each considered framework.

Charland and Leroux [24] evaluated performances of cross-platform frameworks. They stated that frameworks based on web technologies usually experience worse performances when implementing games, in particular if they need animation effects like fading, scrolling, transition effects, etc. However, this loss of performances usually is unnoticeable when implementing business applications where all the aforementioned effects are not used. Analysis and comparison of different cross-platform frameworks is also reported in [12], where authors discuss positive and negative aspects of each cross-platform framework, considering marketplace deployment, adopted technologies, hardware access, native look and feel and perceived performances. They finally pointed out the main important aspects that should be followed to develop the most promising cross-platform framework. Finally, a survey of several "write once, run everywhere" tools, i.e., Phonegap, Titanium and Sencha Touch, is performed in [13]. They examined performances in terms of CPU, memory usage and power consumption for Android test applications. They concluded that Phonegap is the less expensive framework in terms of memory, CPU and energy consumption.

Although many authors analyzed framework for cross-platform mobile development, to the best of our knowledge authors do not address the problem of energy consumption, considering frameworks only from a static point of view, i.e., from a developer point of view. Considering the user point of view, energy consumption of mobile native applications is extremely important and it has been investigated in [25–27], to find a way to correctly measure it.

Since the usage of an external power monitor is tedious and expensive, [25], several researchers focused on the development of an accurate models of energy consumption. For example WattsOn [25] is an energy emulation tool which allows the developers to estimate energy consumed by their apps using their workstation. Carroll and Heiser [26] performed an analysis of power consumption of a particular smartphone and tried to design a model of energy consumption in different scenarios, or Hao et al. [28] combined program analysis and per-instruction energy modeling to estimate energy consumption of mobile application. [29,30] proposed a model of energy consumption of each single hardware component of a mobile device to identify when its behavior differs from the standard one, to identify malicious behavior of Wifi hardware components. In this way, it is easy to associate an energy consumption signature to each hardware component and understand when a suspect behavior is taking place. However, further analysis showed how the consumption estimation of these approaches could suffer of an error that could reach even 10%. Different is the approach followed by PowerScope [11], which is an Android application that runs in background and analyze energy consumption of applications. However, in this case it is not possible to estimate the overhead introduced by the application itself, thus understanding how it influence energy consumption of all the other tasks and applications running on the smartphone. The use of an external monitor is therefore more accurate than all these approaches.

Thiagarajan et al. in [31] analyzed energy consumption of popular websites like Facebook, Apple, Youtube, etc., when rendered using a mobile browser and an Android device. Using a digital power multimeter, they were able to understand and analyze energy consumption of different elements of a web page, like Javascript code, CSS style rendering, image downloading, etc. The main result of their work is that Javascript code or CSS stylesheet should be page specific, meaning that eliminating unnecessary code helps in saving even more than 30% of battery energy.

Berrocal et al. [32] compared resources consumption of different software architectures and they created a conceptual framework with which it is possible to analyze the consumption of an application in the early development phases. They estimated power consumption of an application by composing the power consumption of a set of primitives of which they already know the cost in terms of battery usage in different software architectures. In particular, they analyzed "server-centric" and "client-centric" architectures.

Friedman et al. [33] analyzed power and throughput performances of Bluetooth and WiFi communication on smartphones, measuring the voltage across a resistor connected to the smartphone. They tested different scenarios and setups, considering both performances and power consumption. The results showed that in several conditions WiFi performances are better than the ones using Bluetooth, contradicting previous researches.

As discussed in Section 1, we analyzed energy consumption of different frameworks in [23], using only an Android device, and Phonegap and Titanium as test frameworks. These very initial results pointed out that the adoption of cross-platform frameworks increases energy consumption. The comparison of data acquisition and display from different sensors showed that Phonegap consumed less energy than Titanium in most cases.

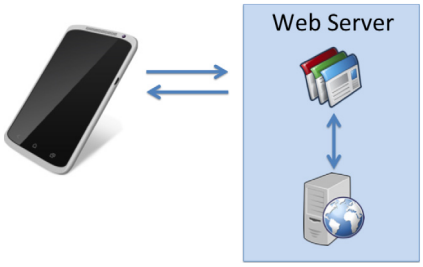

Fig. 1. Architecture of an application using the WA.

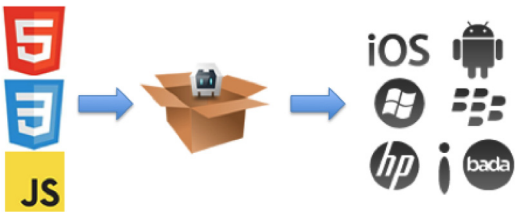

Fig. 2. *Hybrid approach* application architecture.

3. Frameworks classification

Frameworks for cross-platform development can be divided into four classes [15]: the *Web Approach*, the *Hybrid Approach*, the *Interpreted Approach* and the *Cross-Compiled Approach*. This classification is very useful to understand the final results of the experiments, since each framework class deeply influences the performances of the deployed application.

The *Web Approach (WA)* consists of developing a web application, using HTML, CSS and Javascript, which can be accessed through the browser on the smartphone. New features provided by HTML5, as the possibility of acquiring data from different smartphone's sensors, i.e., accelerometer, gyroscope, etc., allows the creation of rich and complex web applications, that can be compared to normal smartphone applications in terms of functionality and acquired data. Moreover, it is possible to have access to user and device specific information, like contacts.

One strong limitation of this approach derives from the different implementation of the features provided by the different mobile browser. Since the current HTML5 features are still under development by the W3C, different browsers can implement or not the different recommendations, or not completely follow the standards, e.g., using different update frequencies. This means that the adoption of recent features must be carefully considered, since it could reduce the set of possible users due to compatibility reasons. Fig. 1 shows this approach. *jQuery Mobile*[2] and *Sencha Touch*[3] are examples of frameworks using this approach.

The *Hybrid Approach (HA)* relies on a WebKit rendering engine to show on the smartphone a web application. On one side, the framework itself provides APIs and access to device hardware and features, while the WebKit engine is responsible to display controls, buttons and animations, and to draw and manage user interface objects. In this case, the application can be distributed and installed on the user device as native mobile application, but the performances are often lower than native solution, since its execution requires to run the browser rendering engine. Fig. 2 shows the architecture of the deployed application using this approach. An example of this class of frameworks is PhoneGap, also known as Apache Cordova.[4]

The third class, called the *Interpreted Approach (InA)*, gives developers the possibility to write the code of the application using a language which is different from languages natively supported by the different platforms, e.g., Javascript. This approach allows developers to use languages they already know and to learn how to use the APIs provided by the framework. The application installed on the device contains also a dedicated interpreter which is used to execute the non-native code. Even if this approach provides access to device features (how many and which features depend on the chosen framework) through an abstraction layer, this additional layer negatively influences the performance. This approach allows the developer to design a final user interface that is identical to the native one without any additional line of code. This kind of frameworks allows an high level of reusable code. Titanium[5] is an example of this class, and Fig. 3 shows the architecture of the framework.

Finally, the *Cross-Compiled Approach (CCA)* is similar to the previous one, since it lets the developer write only one application using a common programming language, e.g., C#, but the final application does not contain an interpreter, but

2 jQuery Mobile: http://jquerymobile.com.
3 Sencha Touch: http://www.sencha.com/products/touch/.
4 Phonegap: http://phonegap.com/.
5 Titanium Appcelerator: http://www.appcelerator.com/titanium/.

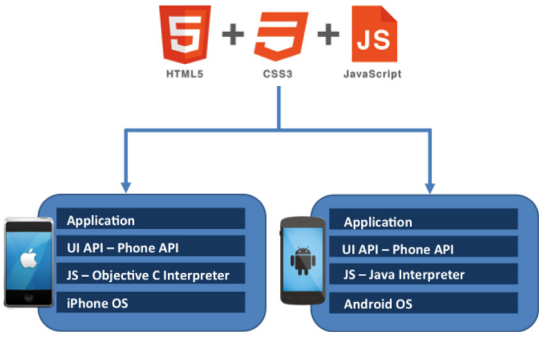

Fig. 3. *Interpreted Approach* architecture.

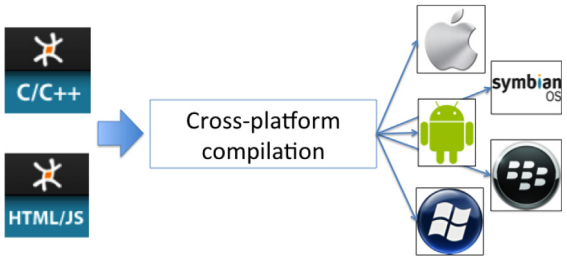

Fig. 4. *Cross-Compiled Approach*.

Table 1
Resume of different cross platform approaches.

Approach	Programming language	Supported platforms	Pros	Cons	Example
Web	HTML, CSS, Javascript	Android, iOS, Windows, BlackBerry[a]	– Easy to update – No installation – Same UI over different devices	– No access to application store – Network delays – Expensive testing – No native UI	jQuery mobile, Sencha Touch
Hybrid	HTML, CSS, Javascript	Android, iOS, Windows, Blackberry,	– Access to application store[b] – Support to most smartphone hardware	– No native look and feel	PhoneGap
Interpreted	Javascript	Android, iOS, Blackberry	– Access to application store – Native look and feel	– Platform branching – Interpretation step	Titanium
Cross-Compiled	C#, C++, Javascript	Android, iOS, Symbian	– Native UI – Real native application	– UI non reusable – High code conversion complexity for complex applications	Mono, MoSync

[a] Support depends on the browser chosen by the user.
[b] Apple store usually tends to refuse applications developed with this approach [34].

the framework generates, after compilation, different native applications for each mobile platform. The final application uses native language, therefore can be considered a native mobile application to all intents and purposes. However, our tests have shown that for complex applications the native solution remains better since the generated code gives worst performances, compared to code written by a developer. An example of this framework are Mono[6] or MoSync,[7] and how it works is depicted in Fig. 4.

Table 1 provides a final resume of this classification, highlighting *pros* and *cons* of each approach.

Considering the final application that each framework produces, and how much this application is similar to a real native one, some issues arise. Starting from the *Web Approach*, it is clear that in this case there is not a native application at all. The *Hybrid Approach* provides a final application that can be installed on the device, but does not use native UI. The *Interpreted*

6 Mono: http://www.mono-project.com/.
7 MoSync: http://www.mosync.com/.

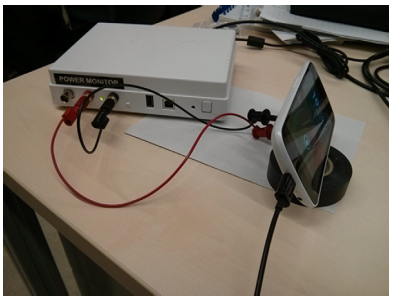

Fig. 5. Hardware setup with one of the Android device.

Approach starts to introduce some native components, i.e., the user interface, while using the interpretation step for the application code. Finally, the *Cross-Compiled Approach* produces a real native application, where no extraneous component is used and only native code is executed.

4. Methodology

In this section we provide details about the hardware and software setup used during the experiments described in this paper. In order to cover the wider set of devices and possible users, applications were deployed both for the Apple and the Android platforms. These two operating systems together cover about 96% of the mobile devices market [5].

4.1. Hardware setup

The test devices, used during the experiments to measure performance in terms of consumed energy, were two Android and two Apple smartphones. Even if the number of considered devices can appear low, different previous works, [35,36, 32,37], have proved that the consumption information provided is accurate enough to create consumption models and estimations. For the Android platform, we used a Samsung Galaxy Nexus i9250 and a Samsung Galaxy S5. The Galaxy Nexus i9250 is equipped with a Dual-core 1.2 GHz CPU, 1 GB RAM and a 720 $\times$ 1280 px display with 16M colors. The Galaxy S5 is equipped with a Quad-core 2.5 GHz CPU, 2 GB RAM and a 1080 $\times$ 1920 px display with 16M colors.

Considering the Apple platform, the test devices were an iPhone 4 and an iPhone 5. The iPhone 4 is equipped with a 1 GHz Cortex CPU, 512 MB RAM and a 640 $\times$ 960 px display with 16M colors. The iPhone 5 is equipped with a Dual-core 1.3 GHz CPU, 1 GB RAM and a 640 $\times$ 1136 px display with 16M colors.

We decided to use these devices because the use an external PowerMonitor requires to have direct access to the battery. For this reason, the possibility of easily opening the devices was a fundamental requirement. Each single component of each device, e.g., brand of each sensor, was not considered, since possible energy consumption differences between different brands can be considered negligible for the final results.

To measure the energy consumed by each application, we did not use software tools installed on the smartphone, but the Monsoon PowerMonitor.[8] The main function of the PowerMonitor is to measure the energy requested by the smartphone (or other devices that use a single lithium battery). The necessary circuit is created isolating the *Voltage* (positive) terminal of the battery and creating a bypass between the PowerMonitor *Vout* to the device. The circuit is finally closed connecting directly to the *Ground* (negative) terminal on the battery. An example of the hardware setup for the Android device is provided in Fig. 5.

Monsoon PowerMonitor does not provide an estimation or creates a model of the energy consumption, [25,26,28,11], but it provides the actual amount of energy requested by the smartphone, i.e., the amount of energy requested to the battery. In fact, during the experiments, smartphones do not use the energy provided by the battery, but the energy provided by the PowerMonitor. This means that, even if the battery is connected to the device, it is not able to provide energy (due to the isolation of the positive terminal), and the PowerMonitor provides (and measures) the requested energy. This permits to measure the precise amount of consumed energy, avoiding errors due to estimations or models. Moreover, even the problem of battery performance deterioration due to usage is removed, since it is isolated during measurements.

We did not choose a software monitor (some examples are discussed in Section 2) since, running on the device, it wastes energy in its turn, and it may influence the measurements, which are then affected by an error very difficult to predict; [25] estimated this error between 4% and 9%, so it affects, in a significant way, the provided results.

The Monsoon PowerMonitor is provided with a software interface which reports several information like consumed energy (measured in μAh), average power (measured in mW), average current (measured in mA), average voltage (measured in V) and expected battery life (h). The most important measure for our purposes is the consumed energy: comparing

[8] Monsoon PowerMonitor, https://www.msoon.com/LabEquipment/PowerMonitor/.

two different tasks along the same time interval and in the same test conditions, the more a task is energy expensive, the more the energy consumption will be higher. This permits a comparison between different frameworks and sensors. Even if interesting, the expected battery life value is a software estimation and cannot supply an objective information about the real energy consumption. It is calculated by the software using the value of the battery capacity with which the smartphone is normally equipped, and the consumed energy of the considered task. It is basically an estimation of how much the battery would last executing constantly the analyzed task/application. Clearly, it is a simple estimation, and do not consider for example battery deterioration due to usage, and cannot provide useful and precise results. For this reason, we do not further consider this measure in the paper.

The use of PowerMonitor is very simple if the smartphone provides access to the lithium battery, so we chose two Android devices with this features (not always present in modern devices). Some difficulties raised with Apple devices. Opening an iPhone 4 and an iPhone 5 was necessary to access the battery, but even in this case, the terminals of the battery were unreachable. So, we created the necessary circuits using the connectors between the battery and the smartphone.

We created an application, and therefore a test, for each considered framework and for each sensor. Each test lasted 2 min and has been repeated three times. The final results for each test have low standard deviation, meaning that the recorded data is very informative and is not affected by underground noise. To have comparable results among all the tests, it was necessary to define a "base" smartphone condition that could be used during all the experiments, avoiding that external factors could affect the measurements and the energy consumption. For this reason, all devices were used in "Airplane mode" (without mobile or WiFi network), with the luminosity of the screen set to the lowest value and, when not tested, with GPS disabled.

4.2. Software setup

The experiments compared together the four different cross-platform approaches: a web application for the *Web Approach*, Phonegap for the *Hybrid Approach*, Titanium for the *Interpreted Approach* and MoSync for the *Cross-Compiled approach*, considering both the C++ and the Javascript implementation. Despite the fact that there are different frameworks for each *approach*, we chose these four different frameworks for the experiments because they actually are the most used frameworks for each category, so we considered them good candidate for the experiments. It is true that differences in the implementations can affect the obtained results, as we will discuss in Section 6 for the MoSync framework, but the architecture followed by these frameworks is the same for all frameworks in the same category, thus results can be considered valid even in presence of small differences in the implementation.

The very important issue for software setup is to be able to reproduce the same test conditions among different frameworks. As an example, in [31] the authors reported that the usage of support libraries when working with HTML5 and Javascript applications, e.g., jQuery, increases the final energy consumption of the application. Therefore, we defined as baseline used to build our test applications the following conditions:

- use only Javascript and not any supporting libraries to manipulate the DOM when developing test application for the *Web* and the *Hybrid Approach*;
- the background of each application is black, the foreground color for the text is white and the font size 15 px.

We used black and white as background and foreground colors because colors representation are standard (e.g., RGB), but how colors are rendered in screen can change according to the device. This means that brightness of a particular color can be different, and so the amount of consumed energy [38]. For this reason, black pixels, which roughly correspond to OFF state of pixels, consume the same energy in all the devices, i.e., around 0 since the pixels are turned off. White pixels are pixels turned on, with the maximum level of brightness according to the brightness of the devices, which is set to minimum in all the devices during the experiments. Therefore, the white pixels consumes the maximum amount of energy for that particular level of brightness, and this is true in all the device. The situation is different for other colors, since the same color code can be represented with different colors and brightness in different devices, and thus requires different amount of energy.

The choice of the font size is arbitrary and could be different, what is important is that the selected conditions are reproducible on all the tested applications. However, this configuration can be considered the lowest expensive one in terms of energy consumption, since most of the screen is black, which is the less expensive configuration.

Since the purpose of this paper is to explore how cross-platform frameworks influence battery consumption when acquiring data from mobile sensors, the conditions mentioned above are sufficient to assure the same testing conditions for all the frameworks when analyzing energy consumption without updating the User Interface (UI), but only acquiring data from a particular sensor.

When updating the User Interface, the situation is more complex, as for example with the light sensor. In this case, it is clear that even the minimum change in the environment, e.g., a movement of a person, influences the data acquired from the sensor and so if the application updates or not the interface. This makes quite impossible to reproduce the same test conditions among all the frameworks, since it is not possible to fully control the environmental conditions, unless closing the smartphone inside an opaque box, but in this case the UI is not updated at all since data acquired never changes. Therefore, for some sensors, i.e., light, proximity sensor and GPS, we decided to analyze only the case when the user interface is not updated, to best guarantee equity between tests. For the audio test, the application records audio from the microphone, and

Table 2
Features supported by the cross-platform frameworks considering the Android platform. By "FC" we mean that the framework lets the developer choose the update frequency.

Sensor	Web App.			Hybrid App.	Interpreted App.	Cross-platform App.	
	Chrome	Firefox	Opera	Phonegap	Titanium	MoSync (C++)	MoSync (Javascript)
Accelerometer	Yes	Yes	Yes	Yes (FC)	Yes	Yes (FC)	Yes
Device orientation	No	Yes	No	No	No	Yes (FC)	No
Compass	No	No	No	Yes (FC)	Yes	No	Yes
Proximity	Yes	No	No	No	No	No	No
Light	Yes	No	No	No	No	No	No
GPS	Yes	Yes	Yes	Yes	Yes	Yes	No
Camera	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Audio record	Yes	Yes	Yes	Yes	No	No	No

no user Interface is updated during this test. For the GPS test, the application constantly requires the position to the GPS sensor, keeping it constantly active and avoiding that the operating system could disable it until a movement is detected.

Despite the native solution, that supports the access to all the native device features and sensors, this is not true for the cross-platform frameworks. The support offered by each framework to a particular device feature depends on the framework itself and, sometimes, on the chosen platform for deployment. This means that, for the most common features like the accelerometer or the GPS, the support is almost complete, while many frameworks do not provide support for other features like light sensor or the device orientation.

Moreover, even the way a framework provides support to a particular feature can change between frameworks, e.g., the update frequency of a sensor value. For example, the updating frequency of the accelerometer value could be the same between the *Web Approach* and the *Interpreted Approach* or, a framework could not allow the developer to change the updating frequency from the default value. The tests followed this policy:

- if the framework allows to change the value of the updating frequency, we test different fixed updating frequencies to understand how energy consumption changes in relation with the update frequency;
- otherwise, we recorded data only at the supported frequency.

Table 2 (for the Android platform) and Table 3 (for the iOS platform) provide a resume of the supported features for each framework. We can note, for example, that the sets of available browsers are different, i.e., Chrome, Firefox and Opera for the Android platform and Chrome and Safari for the iOS platform. Moreover, Titanium support the proximity sensor and audio recording only for iOS devices.

Another frequent task performed with mobile applications is usually send or retrieve data from a server. In this case, energy consumption is affected by the state of the cellular network or of the WiFi conditions, thus not assuring the same test conditions for all the experiments. Moreover, other papers in literature already address this issue, e.g., [31,33].

During the tests, the devices were equipped with Android 5.1.1 (Samsung Galaxy S5) and Android 4.4 (Samsung Galaxy i9250), while for the Apple platform we tested the iPhone 4 with iOS 7.3 and the iPhone 5 with iOS 8. Software used during the tests were Chrome v. 41.0, Firefox v. 31, Opera v. 22.0, MoSync v. 3.8, Titanium v. 3.2.3 and Phonegap v. 3.2. The first idea was to provide the same ambient for the tests in the different devices, i.e., the same operating systems and software. Unfortunately, it was not possible neither to update the iPhone 4 to iOS 8 since Apple does not allow it, nor to downgrade the iPhone 5 to iOS 7.3 for the same reason. The same situation takes place for the Android devices. differently from the iOS platform, the downgrade of Android could be done using other distributions, e.g., cyanogenmod. However, we preferred to use the standard distribution of the platform, to avoid that some differences between the official and the unofficial distribution could change (or influence) the results of the tests. For this reason, tests were performed using the most updated operating system according to the device in use.

5. Experimental results

In this section, we report the results of the experiments. As already discussed, each experiment runs the application for two minutes, and each test was repeated three times. We then calculated the mean value of the "Consumed Energy" of the three runs and the standard deviation, denoted by $\pm n$ where $n \in \mathbb{N}$. The "Consumed Energy" value returned by the PowerMonitor interface is measured in μAh and is the most representative value of power consumption, so it can be used to compare performances of the different frameworks.

Tables 4–11 report details about data recorded during each single experiment.

Tables 4 and 5 present the results obtained with the two Android smartphones when the user interface is updated with sensor values. The results are expressed as additional percentage of consumed energy with respect to the quiet state of the smartphone. The update frequency is 64 ms for frameworks that allow to define the update frequency.

Tables 8 and 9 present the results with the Android device without updating the User Interface, i.e., the application only retrieves data from the considered sensors. Tables 6 and 7 present data obtained with Apple devices, updating the User Interface with sensors' data and Tables 10 and 11 present data recorded with the iPhone devices when the User Interface is not updated.

Table 3
Features supported by the cross-platform frameworks considering the iOS platform. By "FC" we mean that the framework lets the developer choose the update frequency. Other frameworks provide only a value which is a combination of the coordinates.

Sensor	Web App.		Hybrid App.	Interpreted App.	Cross-platform App.	
	Chrome	Safari	Phonegap	Titanium	MoSync (C++)	MoSync (Javascript)
Accelerometer	Yes	Yes	Yes (FC)	Yes	Yes (FC)	Yes
Device orientation	Yes	Yes	No	No	Yes (FC)	No
Compass	No	No	Yes (FC)	Yes	Yes[a]	Yes
Proximity	No	No	No	Yes	Yes	No
Light	No	No	No	No	No	No
GPS	Yes	Yes	Yes	Yes	Yes	No
Camera	No	No	Yes	Yes	Yes	Yes
Audio record	No	No	Yes	Yes	No	No

[a] Framework retrieves 3 values for the coordinates x, y and z.

Table 4
Amount of energy consumed recorded during tests using the Galaxy Nexus, updating the User Interface with sensor data, expressed in term of percentage of increase. $^{(x)}$ indicates the different update frequency.

Sensor	Native	Web App.			Hybrid App.	Interpreted App.	Cross-compiled App.	
		Chrome	Firefox	Opera	PhoneGap	Titanium	MoSync (C++)	MoSync (Javascript)
Only App.	+0.06%	+3.69%	+2.51%	+3.74%	+3.03%	+1.06%	+0.38%	+1.72%
Accelerometer	+36.58%	+115.38%$^{(50)}$	+168.01%	+123.74%$^{(50)}$	+75.01%	+79.29%	+240.88%	+74.07%
Device orient.	+45.84%	–	+180.15%	–	–	–	+250.12%	–
Compass	+30.87%	–	–	–	+47.72%	–	–	+31.69%

Table 5
Percentage of increase in the amount of energy consumed recorded during tests using the Galaxy S5, updating the User Interface with sensor data. $^{(x)}$ indicates the different update frequency.

Sensor	Native	Web App.			Hybrid App.	Interpreted App.	Cross-compiled App.	
		Chrome	Firefox	Opera	PhoneGap	Titanium	MoSync (C++)	MoSync (Javascript)
Only App.	+0.09%	+3.17%	+2.11%	+3.67%	+2.74%	+1.28%	+1.01%	+1.98%
Accelerometer	+115.08%	+675.37%$^{(50)}$	+764.25%	+691.59%$^{(50)}$	+159.62%	+334.59%	+774.04%	+129.87%
Device orient.	+123.17%	–	+485.78%	–	–	–	+686.24%	–
Compass	+120.61%	–	–	–	+162.31%	–	–	+143.47%

Table 6
Amount of consumed energy, recorded during tests using the iPhone 4, updating the User Interface with sensor data, in term of percentage of increase. $^{(x)}$ indicates the different update frequency.

Sensor	Native	Web App.		Hybrid App.	Interpreted App.	Cross-compiled App.	
		Chrome	Safari	PhoneGap	Titanium	MoSync (C++)	MoSync (Javascript)
Only App.	+0.78%	+2.91%	+2.06%	+2.5%	+1.53%	+2.49%	+3.49%
Accelerometer	+8.68%	+94.48%$^{(50)}$	+94.47%$^{(50)}$	+79.10%	+26.38%$^{(100)}$	+190.94%	+25.38%$^{(150)}$
Device orient.	+15.21%	+950.4%$^{(50)}$	49.98%$^{(50)}$	–	–	+180.05%	–
Compass	+51.98%	–	–	+69.51%	–	–	+42.79%$^{(140)}$

Table 7
Percentage of increase in the amount of consumed energy, recorded during tests using the iPhone 5, updating the User Interface with sensor data. $^{(x)}$ indicates the different update frequency.

Sensor	Native	Web App.		Hybrid App.	Interpreted App.	Cross-compiled App.	
		Chrome	Safari	PhoneGap	Titanium	MoSync (C++)	MoSync (Javascript)
Only App.	+1.69%	+2.17%	+3.85%	+3.88%	+3.26%	+4.83%	+5.34%
Accelerometer	+5.34%	+86.78%$^{(50)}$	+144.84%$^{(50)}$	+52.16%	+15.01%$^{(100)}$	+298.28%	+16.52%$^{(150)}$
Device orient.	+14.26%	+86.53%$^{(50)}$	+147.07%$^{(50)}$	–	–	+286.47%	–
Compass	+30.25%	–	–	+48.25%	–	–	+43.97%$^{(140)}$

5.1. Basic energy consumption

To evaluate the effective cost of each framework, in terms of power consumption, a reference value as baseline is required. The ideal baseline value would be the smartphone on, without any other application or background process running, and

Table 8
Amount of consumed energy, recorded during tests using the Galaxy Nexus without updating the User Interface with sensor data expressed in form of percentage.

Sensor	Native	Web App.			Hybrid App.	Interpreted App.	Cross-compiled App.	
		Chrome	Firefox	Opera	PhoneGap	Titanium	MoSync (C++)	MoSync (Javascript)
Only App.	+0.06%	+3.69%	+2.51%	+3.74%	+3.03%	+1.06%	+0.38%	+1.72%
Accelerometer	+3.04%	+18.05%(50)	+32.74%	+27.65%	+7.75%	+4.92%	+239.75%	+6.82%
Device orient.	+10.76%	+20.04%	+26.92%	+26.23%	–	–	+243.35%	–
Compass	+8.61%	–	–	–	+12.24%	–	–	+10.34%
Proximity	+0.90%	–	+2.00%	–	–	–	+200.65%	–
Light	+2.18%	–	+3.79%	–	–	–	–	–
GPS	+42.80%	+62.49%	+46.07%	+60.15%	+43.15%	+43.48%	+43.25%	–
Camera	+233.85%	+200.15%[a]	+208.26%[a]	+258.06%	+252.36%	+250.17%	+244.37%	+254.20%
Audio record	+20.56%	+48.97%	+53.37%	+49.22%	+21.76%	–	–	–

[a] Images from camera are not full screen but only a small section of the web page.

Table 9
Percentage of increase in the amount of consumed energy, recorded during tests using the Galaxy S5 without updating the User Interface with sensor data.

Sensor	Native	Web App.			Hybrid App.	Interpreted App.	Cross-compiled App.	
		Chrome	Firefox	Opera	PhoneGap	Titanium	MoSync (C++)	MoSync (Javascript)
Only App.	+0.09%	+3.17%	+2.11%	+3.67%	+2.74%	+1.28%	+1.01%	+1.98%
Accelerometer	+28.21%	+41.14%(50)	+44.32%	+43.37%	+33.74%	+30.14%	+588.86%	+32.14%
Device orient.	+13.08%	+33.15%	+46.36%	+45.58%	–	–	+542.65%	–
Compass	+12.74%	–	–	–	+24.54%	–	–	+22.63%
Proximity	+1.92%	–	+5.66%	–	–	–	+529.05%	–
Light	+6.59%	–	+53.82%	–	–	–	–	–
GPS	+80.56%	+93.43%	+84.26%	+97.84%	+82.69%	+83.1%	+81.7%	–
Camera	+544.95%	+219.17%[a]	+223.15%[a]	+214.61%	+557.05%	+552.24%	+551.49%	+563.89%
Audio record	+60.46%	+73.49%	+117.90%	+77.78%	+58.24%	–	–	–

[a] Images from camera are not full screen but only a small section of the web page.

Table 10
Percentage of energy consumption increase recorded during tests with the iPhone 4 without updating the UI with sensor data.

Sensor	Native	Web App.		Hybrid App.	Interpreted App.	Cross-compiled App.	
		Chrome	Safari	PhoneGap	Titanium	MoSync (C++)	MoSync (Javascript)
Only App.	+0.78%	+2.91%	+2.06%	+2.50%	+1.53%	+2.49%	+3.49%
Accelerometer	+4.93%	+54.28%(50)	+55.94%(50)	+47.69%	+9.03%(100)	+175.50%	+12.92%
Device orientation	+15.21%	+50.4%(50)	+49.89%(50)	–	–	+180.05%	–
Compass	+36.36%	–	–	+58.48%	–	–	+35.27%(140)
Proximity	+19.55%	–	–	–	+20.21%	+171.97%	–
GPS	+167.81%	+170.19%	+171.54%	+175.43%	+171.85%	+318.42%	–
Camera	+254.39%	–	–	+314.78%	+315.81%	+306.46%	+286.46%
Audio record	+7.51%	–	–	+14.91%	+15.40%	–	–

Table 11
Percentage of energy consumption increase recorded during tests with the iPhone 5 without updating the UI with sensor data.

Sensor	Native	Web App.		Hybrid App.	Interpreted App.	Cross-compiled App.	
		Chrome	Safari	PhoneGap	Titanium	MoSync (C++)	MoSync (Javascript)
Only App.	+1.69%	+2.17%	+3.85%	+3.88%	+3.26%	+4.83%	+5.34%
Accelerometer	+2.56%	+42.53%(50)	+39.98%(50)	+25.18%	+2.47%(100)	+263.94%	+4.91%
Device orientation	+10.19%	+41.81%(50)	+39.92%(50)	–	–	+274.24%	–
Compass	+28.75%	–	–	+44.23%	–	–	+40.25%(140)
Proximity	+0.33%	–	–	–	+1.58%	+271.96%	–
GPS	+168.68%	+171.57%	+171.94%	+176.33%	+172.74%	319.79%	–
Camera	+271.76%	–	–	+321.67%	+311.38%	+286.52%	+296.70%
Audio record	+11.42%	–	–	+15.73%	+21.34%	–	–

where the "computational workload" is the lowest possible. To reach this lowest value as closely as possible, we put the smartphones in "Airplane Mode", without any application running, with the lowest luminosity of the screen and black background of the home screen. Moreover, we minimized the number of graphical elements on the screen, i.e., the icons. For the Android devices, we were able to remove all the graphical elements on the screen (despite some small icons and the time in the top right of the screen), and the measured consumption was $5126_{\pm 11}$ μAh for the Galaxy Nexus and $2213_{\pm 15}$ μAh

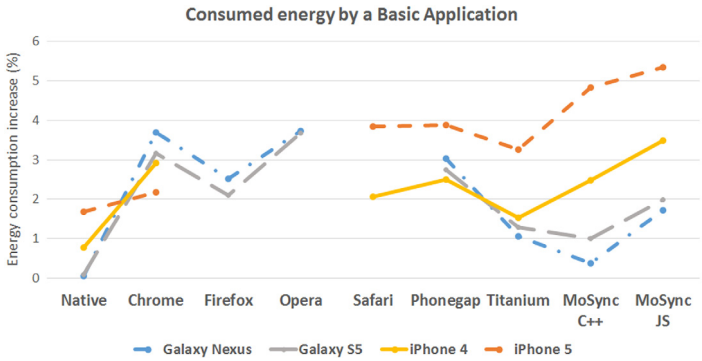

Fig. 6. Consumed energy, expressed in terms of percentage increase, by a basic application developed with the different cross-platform frameworks.

for the Samsung S5. For the Apple devices, we created this ideal "quiet state" by setting a black background of the screen, and choosing a black icon for an application, since it is not possible to remove all the graphical elements from the Home screen (the "quiet state" of the iOS system has at least one icon in the upper part of the screen). The consumption values for the Apple devices were $2777_{\pm 10}$ μAh for the iPhone 4 and $2558_{\pm 6}$ μAh for the iPhone 5.

We must note here that the Apple devices has similar baseline, while the Android devices have not. Even if this is an interesting aspect to notice, this difference in energy consumption does not affect the results and the conclusions derived for the frameworks, since the energy consumption increase is calculated using this reference value that is the same for all the tested frameworks given the same device. All figures in this section report the consumed energy expressed in terms of percentage increase, with respect to the correct baseline. For this reason, even the native solution does not report a zero value since also a basic application has an energy consumption superior, by definition, to the baseline.

5.2. Energy consumption for a simple cross-platform application

The first experiment aims at measuring the energy consumed only by the usage of a particular framework to create an application without any particular behavior. Therefore, for each considered framework, we developed a basic application which does not perform any operation, without any graphical elements or underground task. Then, we compared the energy consumption of all these applications with a native one with the same features. Results are provided in Fig. 6, expressed in terms of percentage increase with respect to the baseline measured for the used device. Since Firefox and Opera cannot be installed on the Apple devices, data is missing. In the same way, since Safari is not available for the Android environment, it is not possible to test this combination of browser and operating system.

As results show, the adoption of an application developed using a cross-platform framework has a cost in terms of energy consumption. Without considering the *Cross-Compiled Approach*, which has very different performances in the two platforms, the more expensive approaches are the *Web Approach* and the *Hybrid Approach*, which runs a browser engine, that is clearly more expensive than a simple application. Instead, the performances of the *Cross-Compiled Approach* are affected by the chosen development language and platform. In both cases, i.e., Android and Apple, performances measured for the application developed using Javascript are worst with respect to the ones collected using C++. But the two platforms performs very differently: while for the Android platform the C++ application is the best one among all other cross-platform frameworks, and the Javascript implementation performs better than the *Web* and the *Hybrid Approaches*, in the Apple platform the performances using Javascript are the worst.

Another important results of this first experiment is related to the generalization of these results. Even if the number of used devices does not cover all the available devices on the market, we can notice that, looking separately at the two platforms, the curves trend is comparable, i.e., the curve trend of data for iPhone 4 is very similar to the curve trend of data for iPhone 5 and the same holds for the experiments with Android. This is very important, because it is possible to assert that these results are representative for both platforms.

5.3. Energy consumption using sensors

After this initial test, we analyzed the performances of each framework, when acquiring data from different sensors. Tests were performed only for that sensors that are supported by at least one framework, and we compared the performances with the native solution. The initial idea was to compare frameworks using the same update frequency, but not all of them let the developer choose it, as already explained in the previous section. Therefore, data presented in the following figures refer to the same update frequency (64 ms), unless the framework does not allow for customization. In this case, the update frequency is reported between rounded brackets in Tables 4–11.

The experiments give an interesting result, in some way predictable: the adoption of a cross-platform framework to develop an application always implies an higher energy consumption with respect to the native solution, and this is true for

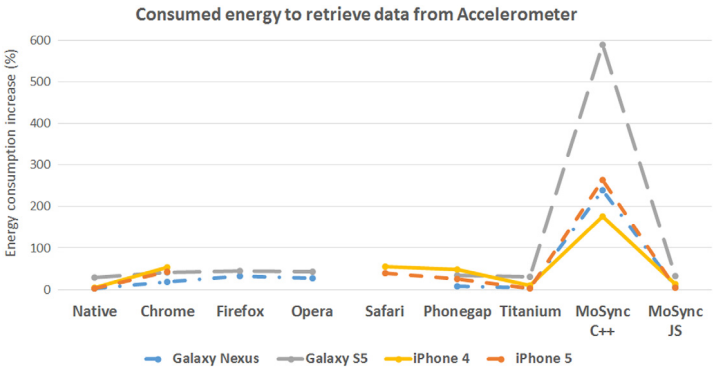

Fig. 7. Consumed energy, expressed in terms of percentage increase, by an application which retrieves data from accelerometer.

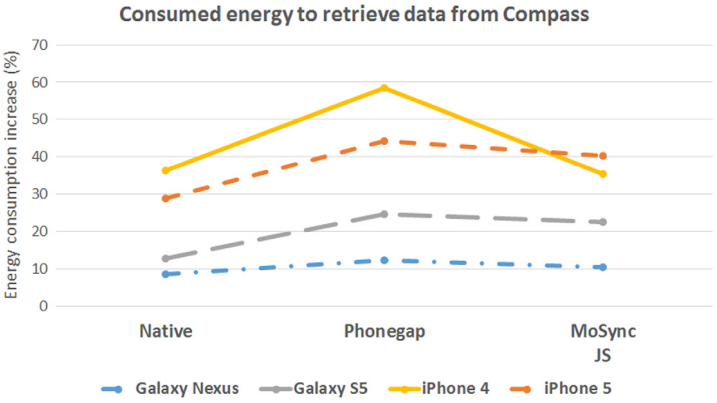

Fig. 8. Consumed energy, expressed in terms of percentage increase, by an application which retrieves data from compass.

all frameworks in both the considered device platforms. Considering for example the accelerometer, Fig. 7 shows that the curve trend is always the same, independently from the used platform. The same behavior appears also for other sensors like compass (see Fig. 8), proximity sensor (see Fig. 9) and to collect data about device orientation (see Fig. 10). Despite the differences between the devices, the charts which illustrate the results show the same curve trends. For this reason, we can argue that the obtained results are generalizable to all devices of both the platform.

Fig. 11 shows that the GPS sensors has the same behavior, except for the MoSync C++ implementation on iPhone 4 and 5,[9] but we also note that the same implementation obtained the worst result also for other sensors like proximity sensor (see Fig. 9), orientation (see Figs. 10 and 14) and accelerometer (see Figs. 7 and 12). Therefore this implementation is actually not efficient to retrieve data from sensors in all platforms.

5.4. Power consumed updating User Interface with data from sensors

The experiments allow also to measure that the increment in energy consumption is higher when the application updates the user interface, showing the data retrieved from sensors, while it is lower, but still present, when the user interface is not updated (see Fig. 7 vs. Fig. 12, Fig. 8 vs. 13 and Fig. 10 vs. Fig. 14). This means that, for all these frameworks, the most expensive task is not to execute or interpret in real-time the code (as in the case of the *Interpreted Approach*), but the more expensive task is to update the graphical elements on the screen.

Consider, for example, the case of data acquisition from the accelerometer. If we compare Figs. 7 and 12 it is possible to notice that the difference in terms of consumed energy is particularly relevant for the *Web Approach* and the *Cross-Compiled Approach* using the C++ implementation of MoSync. Moreover, the difference can be higher: comparing the difference between the native solution and the one using the PhoneGap framework in all the devices, the difference of energy consumption increases required by PhoneGap when updating the user interface with respect to the native solution varies between 38% and 71%, while it lowers down between 2% and 3% when the user interface is not updated (and this is not one of the framework performing worst).

[9] Data about energy consumption for iPhone 4 and 5 for GPS are quite superimposable.

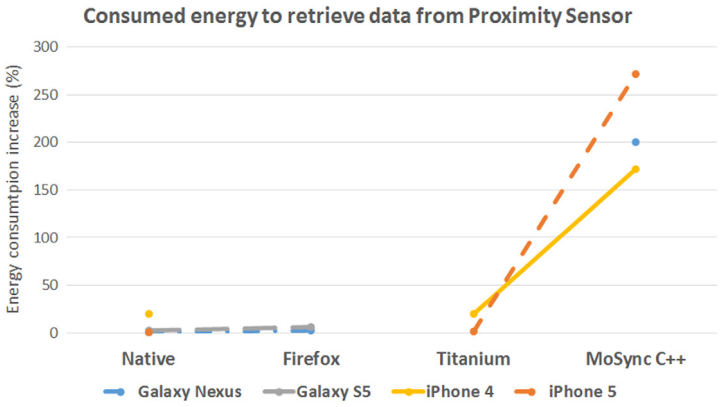

Fig. 9. Consumed energy, expressed in terms of percentage increase, by an application which retrieves data from proximity sensor.

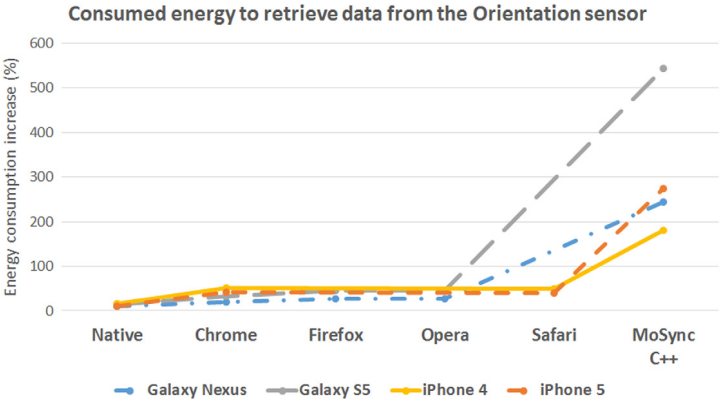

Fig. 10. Consumed energy, expressed in terms of percentage increase, by an application which retrieves data about device orientation.

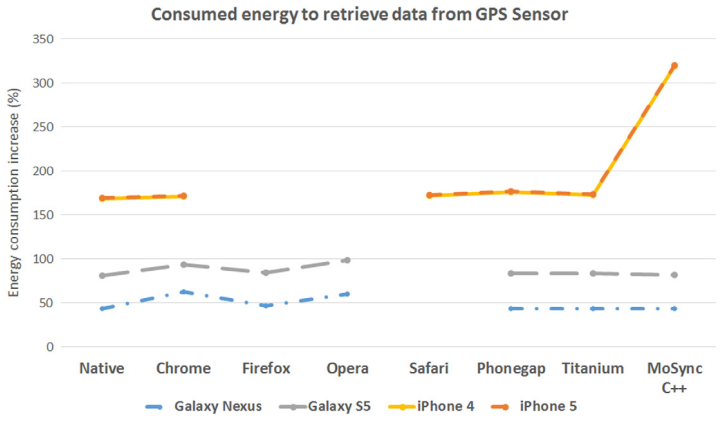

Fig. 11. Consumed energy, expressed in terms of percentage increase, by an application which retrieves data from GPS. Note that data about measurements for iPhone 4 and iPhone 5 are quite superimposable.

Considering the compass sensor and comparing Figs. 8 and 13 we can see that the curves trends are the same, but the curves are moved upper, since the consumed energy is augmented. This behavior deeply affects the measurement for the Galaxy S5, but it is present for all the devices. Even in the case of data about device orientation, analyzing Figs. 10 and 14 the energy consumption increased, in particular for the *Cross-Compiled Approach*.

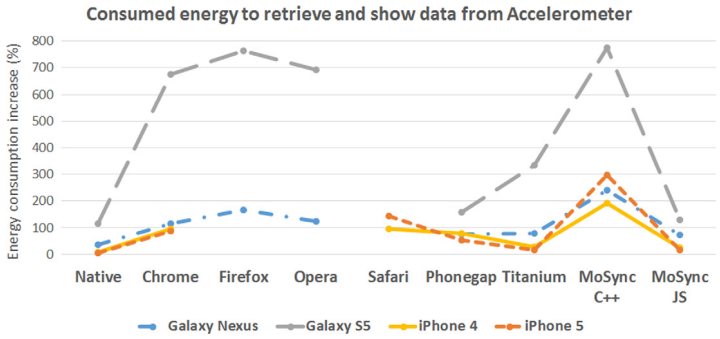

Fig. 12. Consumed energy, expressed in terms of percentage increase, by an application which retrieves and shows data from accelerometer.

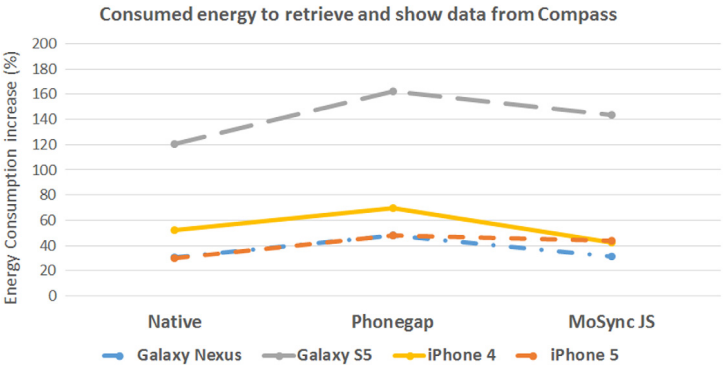

Fig. 13. Consumed energy, expressed in terms of percentage increase, by an application which retrieves and shows data from compass.

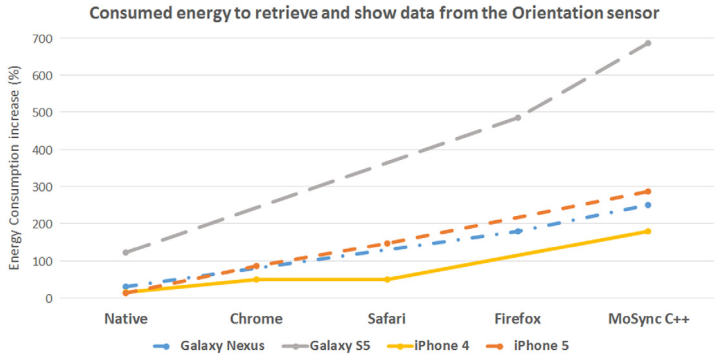

Fig. 14. Consumed energy, expressed in terms of percentage increase, by an application which retrieves and shows data about device orientation.

6. Discussion

6.1. Energy consumption

Data retrieved from the experiments discussed so far, leads to some considerations. An interesting case study is represented by the MoSync framework. As already explained before, this framework follows the *Cross-Compiled Approach*, meaning that it generates a real native application, translating the code into real native code. MoSync supports two programming languages, C++ or Javascript. We tested both the possibilities.

The experiments showed that the C++ implementation, when dealing with data retrieved from sensors, is much more expensive, both with respect to the Javascript implementation and to the native solution. The explanation of this result comes from the nature of C++: since it does not natively handle events (without using an external library), the final result is an application that uses *polling*, i.e., to retrieve data from sensors it implements an infinite cycle that runs for all the application lifecycle till its termination. This behavior is then translated into the final application and, from performances point of view, it is extremely energy consuming. This means that this kind of implementation could not be used "as is" when

developing applications that retrieve and use data from sensors. A possible solution is to use it together with a C++ library which provides support to events.

Without considering the C++ implementation of the MoSync framework, the *Web Approach*, where applications are rendered using a Web browser, is the most expensive one, even if it is used in offline mode, i.e., without being connected to the Internet (one of the new HTML5 feature). There are different reasons to explain this result. The first one is the early stage of development of these features. The possibility to access smartphone sensors are far to be standard and fully supported by all browsers and devices so, at the time of writing, browsers and devices probably are not optimized for an heavy use, yet. Secondly, when a page requires sensor data, this request is made by the Javascript engine, that is interpreted by the browser and translated to a request into the native language of the OS, thus consuming more energy. Finally, the first experiment showed that the execution of a web browser is more expensive than a simple native smartphone application, even without retrieving data from sensors. Therefore, since the *Web Approach* combines together all these aspects, it is clear that in the end the energy consumption of this particular approach is heavy and not suitable for smartphone applications.

Another interesting aspect that we can notice from the analysis of the data, is how the updating of the user interface affects the energy consumption of the frameworks. Let us consider, as an example, Phonegap and Titanium frameworks used to develop an application which retrieves data from the accelerometer, since this is the only sensor supported by both the frameworks. Updating the User Interface (see Fig. 12), Phonegap is less expensive than Titanium in the Android platforms and vice versa in the Apple platform. Without updating the User Interface (see Fig. 7), Titanium is less expensive than Phonegap in both platforms. This means that the mapping between the Javascript code and User Interface elements made by Titanium during compilation and runtime is more expensive than the *Hybrid Approach* of Phonegap in Android devices. Analyzing Phonegap and Titanium frameworks and their performances in both the platforms, it is possible to conclude that Titanium has better performances on the Apple devices, while on the Android platform, Phonegap is sometimes better. This behavior takes two different considerations. The first one is that probably Titanium is much more optimized for the Apple platform instead of the Android one. The second one is that if we must choose between Titanium or Phonegap as developing framework, it is not possible to find the best solution in terms of energy consumption. This means that the choice should consider how users are distributed among the different platforms, and choose the developing framework depending on this distribution (and this is against the idea of cross-platform development). Since cross-platform frameworks can have different behaviors in different platforms, their current state-of-art does not guarantee a real cross-platform development.

6.2. Selection of the best framework

The choice of which framework to use must take into account, first of all, the number and type of supported features, since not all the native features are supported by cross-platform frameworks, e.g., the compass is not supported by browsers (i.e., the *Web Approach*), Titanium and MoSync (using C++).

To choose a winner for the best framework among the considered ones, it is clear that MoSync, using the Javascript implementation, when available, is the one with best performances in terms of energy consumption. The resulting code and final application leads to better results with respect to other approaches like the *Hybrid* or the *Interpreted* one, where code is executed in an encapsulated environment (a web browser) or where the code is interpreted at runtime. It performs better than the C++ implementation of the same framework due to the fact that C++, as already discussed before, does not efficiently handle alone events, meaning that this implementation of MoSync may improve using a library which handles the events. The Javascript implementation is not affected by this problem and so the results are better.

For these reasons, the Javascript implementation of this framework can be a good choice for applications which massively use data from sensors, e.g., applications for runners.

Even if the *Cross-Compiled Approach* produces a native application without any additional software layer that can lower performances, in terms of power consumption performances are deeply affected by the quality of the compilation process which translates the application from the language used for development to the native one. Therefore, this result is not a-priori predictable. This is highlighted by experiments made with the C++ implementation of MoSync.

Simple applications which do not retrieve data from sensors, e.g., home banking, do not require particular attention on the choice of the framework to use, since the increment in terms of energy consumption is always under 6% (the worst case is 5.34%).

6.3. Suggestion for the improvement of the efficiency of the frameworks

The results obtained from the experiments can be read in different ways, some correct and some incorrect. The experiments show that energy consumption performances of cross-platform frameworks are still far from the native approach. Therefore, these frameworks are currently difficultly adoptable when the applications acquire data from sensors (a condition often true especially for ubiquitous and pervasive applications) and update the user interface.

This problem is difficult to solve, and will require a lot of work in the next years, but it is not correct to state that cross-platform frameworks cannot improve in the future. In fact, considering the classification of frameworks described in Section 3, an important result obtained with the experiments is to highlight which are the components that consume more energy, since the experiments identified as the user interface rendering the more consuming task. Therefore, an efficient cross-platform framework must pay particular attention on the efficiency of the UI rendering.

Considering the *Web Approach*, it requires a browser for rendering the application. Therefore, these frameworks can improve their energy consumption only if the future implementations of the browsers become more efficient. The *Hybrid Approach* needs a WebKit engine, a component of a browser, for rendering. Even in this case, improvements in energy consumption do not depend on the frameworks implementation but on the implementation of the WebKit engine.

A different situation is represented by the other two classes of frameworks: in this case they do not rely on external components for rendering, therefore they can improve their energy consumption by improving their implementation. The *Interpreted Approach* contains a real-time interpretation step. This software layer increases the amount of CPU operations with respect to the native approach, thus requiring more energy, but research can contribute to lower this amount of CPU operations.

We consider the *Cross-Compiled Approach* the most promising one, so we suggest to use it for implementation of new frameworks. However, it is crucial that the development of this kind of framework focus its attention on producing application code that is efficient and with performances comparable to native code. In particular, developers should pay special attention to the optimization of events management. Otherwise, a "bad" implementation can waste the advantage of this class of frameworks, as in the case of the C++ implementation of MoSync. Instead, the Javascript implementation of MoSync shows that this goal is achievable.

7. Conclusions

Market fragmentation due to the variety of existing mobile devices and to the availability of different operating systems have posed new important issues in mobile applications development, since mobile applications should reach the highest number of possible users.

To overtake this problem, cross-platform frameworks, that let develop one single application that is further deployed to each target mobile OS, are currently under development and analysis.

The main contribution of this paper is the analysis of one of the most important aspect when dealing with mobile devices: energy consumption. In particular, we considered all the possible sensors of smartphone devices that can be used to acquire information about the environment or the context of the user, and we analyzed how retrieving data from these sensors (and showing or not the retrieved value on the User Interface) influences energy consumption depending on the used framework. We defined test cases in order to assure the same test conditions for each framework, i.e., screen luminosity, application background, text size and colors, etc.

Our results showed how the adoption of cross-platform frameworks as development tools always implies an increase in energy consumption, even if the final application is a real native application, i.e., using the *Cross-Compiled Approach*. Moreover, we compared together frameworks with the aim of finding a final rank to the current best approach to follow. Actually, the *Cross-Compiled Approach*, i.e., MoSync in our test, in particular using the Javascript implementation, if available, is the one that lowers the increase of the amount of energy required to acquire data from sensors and show them, i.e., to update the User Interface.

Unfortunately, the same approach using another programming language, i.e., MoSync using C++, has the worst performance when retrieving data from sensors (except for the compass, for which the performance are comparable to the *Hybrid* approach).

Experiments also showed that is not possible to have a final and absolute rank for all the considered framework, since the set of supported features and sensors is not the same, and some frameworks are better optimized for a platform instead of the other one. Clearly, this is against the idea of *cross-platform development*. Therefore, for those applications that consider energy consumption as a big issue, at the time of writing, the native solution is better, or the framework to use should be carefully considered. Again, to state that the implementation of an efficient cross-platform framework is not possible is incorrect, but the developers of these frameworks should also consider the energy consumption issue, since it is very important and its current state-of-art is only at the beginning. In this paper we gave some indications to improve the current frameworks in term of power consumption.

Another important aspect which was highlighted by the experiments is that the update of the User Interface represents the most expensive task, and the main cause of the increase in energy consumption. Moreover, the updating frequency of data retrieved by sensors and their visualization heavily affects energy consumption. Therefore, the developer has to take into consideration very carefully these two aspects when dealing with mobile applications and, in particular, with data from sensors.

Finally, if the cross-platform framework is necessary, currently MoSync using Javascript is the one that assures the consumption more similar (but still higher) to the native solution.

References

[1] G.D. Abowd, A. Dey, P. Brown, N. Davies, M. Smith, P. Steggles, Towards a better understanding of context and context-awareness, in: H.-W. Gellersen (Ed.), Handheld and Ubiquitous Computing, in: Lecture Notes in Computer Science, vol. 1707, Springer, Berlin, Heidelberg, 1999, pp. 304–307.

[2] L. Bao, S. Intille, Activity recognition from user-annotated acceleration data, in: A. Ferscha, F. Mattern (Eds.), Pervasive Computing, in: Lecture Notes in Computer Science, vol. 3001, Springer, Berlin, Heidelberg, 2004, pp. 1–17.

[3] M. Ciman, M. Donini, O. Gaggi, F. Aiolli, Stairstep recognition and counting in a serious game for increasing users' physical activity, Pers. Ubiquitous Comput. (2016) http://dx.doi.org/10.1007/s00779-016-0968-y.

[4] W.S.J. Yong-Gyu Lee, G. Yoon, Smartphone-based mobile health monitoring, Telemed. e-Health 18 (2012) 585–590. http://dx.doi.org/10.1089/tmj.2011.0245.
[5] IDC Corporate USA, Smartphone OS Market Share, 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.
[6] A. Saldarriaga, J. Perez, J. Restrepo, J. Bustamante, A mobile application for ambulatory electrocardiographic monitoring in clinical and domestic environments, in: Health Care Exchanges, PAHCE, 2013 Pan American, 2013.
[7] J. Oresko, Z. Jin, J. Cheng, S. Huang, Y. Sun, H. Duschl, A. Cheng, A wearable smartphone-based platform for real-time cardiovascular disease detection via electrocardiogram processing, IEEE Trans. Inform. Tech. Biomed. 14 (3) (2010) 734–740.
[8] M. Ciman, K. Wac, Individuals' stress assessment using human–smartphone interaction analysis, IEEE Trans. Affective Comput. PP (99) (2016) 1–15. http://dx.doi.org/10.1109/TAFFC.2016.2592504.
[9] M. Ciman, O. Gaggi, N. Gonzo, Cross-platform mobile development: a study on apps with animations, in: Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014, 2014, pp. 757–759.
[10] H. Heitkotter, S. Hanschke, T.A. Majchrzak, Evaluating cross-platform development approaches for mobile applications, in: Web Information Systems and Technologies, Vol. 140, 2013, pp. 120–138.
[11] J. Flinn, M. Satyanarayanan, Powerscope: a tool for profiling the energy usage of mobile applications, in: Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications, WMCSA '99, 1999.
[12] S. Xanthopoulos, S. Xinogalos, A comparative analysis of cross-platform development approaches for mobile applications, in: Proceedings of the 6th Balkan Conference in Informatics, BCI '13, 2013, pp. 213–220.
[13] I. Dalmasso, S. Datta, C. Bonnet, N. Nikaein, Survey, comparison and evaluation of cross platform mobile application development tools, in: Proceedings of the 9th International Wireless Communications and Mobile Computing Conference, IWCMC, 2013, pp. 323–328.
[14] M. Palmieri, I. Singh, A. Cicchetti, Comparison of cross-platform mobile development tools, in: Proceedings of the 16th International Conference on Intelligence in Next Generation Networks, ICIN, 2012, pp. 179–186.
[15] R. Raj, S. Tolety, A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach, in: Annual IEEE India Conference, INDICON '12, 2012, pp. 625–629.
[16] W.S. El-Kassas, B.A. Abdullah, A.H. Yousef, A.M. Wahba, Taxonomy of cross-platform mobile applications development approaches, Ain Shams Eng. J. (2015) http://dx.doi.org/10.1016/j.asej.2015.08.004.
[17] L. Bloom, R. Eardley, E. Geelhoed, M. Manahan, P. Ranganathan, Investigating the relationship between battery life and user acceptance of dynamic, energy-aware interfaces on handhelds, in: Proceedings of the International Conference on Human Computer Interaction with Mobile Devices & Services, 2004, pp. 13–24.
[18] A. Rahmati, A. Qian, L. Zhong, Understanding human–battery interaction on mobile phones, in: Proceedings of the 9th International Conference on Human Computer Interaction with Mobile Devices and Services, MobileHCI '07, 2007, pp. 265–272.
[19] A. Merlo, M. Migliardi, L. Caviglione, A survey on energy-aware security mechanisms, Pervasive Mob. Comput. 24 (C) (2015) 77–90. http://dx.doi.org/10.1016/j.pmcj.2015.05.005.
[20] H. Qian, D. Andresen, Extending mobile device's battery life by offloading computation to cloud, in: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft '15, 2015, pp. 150–151.
[21] M. Weiser, The computer for the 21st century, SIGMOBILE Mob. Comput. Commun. Rev. 3 (3) (1999) 3–11.
[22] C. Wilke, S. Richly, S. Gtz, C. Piechnick, U. Amann, Energy consumption and efficiency in mobile applications: A user feedback study, in: Green Computing and Communications, GreenCom, 2013 IEEE and Internet of Things,iThings/CPSCom, IEEE International Conference on and IEEE Cyber, Physical and Social Computing, 2013, pp. 134–141. http://dx.doi.org/10.1109/GreenCom-iThings-CPSCom.2013.45.
[23] M. Ciman, O. Gaggi, Evaluating impact of cross-platform frameworks in energy consumption of mobile applications, in: Proceedings of the 10th International Conference on Web Information Systems and Technologies, WEBIST 2014, 2014, pp. 423–431.
[24] A. Charland, B. Leroux, Mobile application development: Web vs. native, Commun. ACM 54 (5) (2011) 49–53.
[25] R. Mittal, A. Kansal, R. Chandra, Empowering developers to estimate app energy consumption, in: Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, Mobicom '12, 2012, pp. 317–328.
[26] A. Carroll, G. Heiser, An analysis of power consumption in a smartphone, in: USENIX Annual Technical Conference, 2010, pp. 271–285.
[27] G. Perrucci, F. Fitzek, J. Widmer, Survey on energy consumption entities on the smartphone platform, in: Vehicular Technology Conference (VTC Spring), vol. 2011, IEEE, 73rd, 2011, pp. 1–6.
[28] S. Hao, D. Li, W.G.J. Halfond, R. Govindan, Estimating mobile application energy consumption using program analysis, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 92–101. URL http://dl.acm.org/citation.cfm?id=2486788.2486801.
[29] M. Curti, A. Merlo, M. Migliardi, S. Schiappacasse, Towards energy-aware intrusion detection systems on mobile devices, in: High Performance Computing and Simulation, HPCS, 2013 International Conference on, 2013, pp. 289–296. http://dx.doi.org/10.1109/HPCSim.2013.6641428.
[30] A. Merlo, M. Migliardi, P. Fontanelli, On energy-based profiling of malware in android, in: High Performance Computing Simulation, HPCS, 2014 International Conference on, 2014, pp. 535–542. http://dx.doi.org/10.1109/HPCSim.2014.6903732.
[31] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, J.P. Singh, Who killed my battery?: Analyzing mobile browser energy consumption, in: Proceedings of the 21st International Conference on World Wide Web, WWW '12, 2012, pp. 41–50.
[32] J. Berrocal, J. Garcia-Alonso, C. Vicente-Chicote, J. Hernndez, T. Mikkonen, C. Canal, J.M. Murillo, Analysing the resource consumption patterns of mobile applications in early development phases, Pervasive Mob. Comput. (2016). http://dx.doi.org/10.1016/j.pmcj.2016.06.011.
[33] R. Friedman, A. Kogan, Y. Krivolapov, On power and throughput tradeoffs of wifi and bluetooth in smartphones, IEEE Trans. Mob. Comput. 12 (7) (2013) 1363–1376.
[34] A. Trice, PhoneGap advice on dealing with Apple application rejection, October 2012. http://www.adobe.com/devnet/phonegap/articles/apple-application-rejections-and-phonegap-advice.html.
[35] A. Merlo, M. Migliardi, P. Fontanelli, Measuring and estimating power consumption in android to support energy-based intrusion detection, J. Comput. Secur. 23 (5) (2015) 611–637.
[36] W. Jung, C. Kang, C. Yoon, D. Kim, H. Cha, Devscope: A nonintrusive and online power analysis tool for smartphone hardware components, in: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '12, ACM, New York, NY, USA, 2012, pp. 353–362. http://dx.doi.org/10.1145/2380445.2380502, URL http://doi.acm.org/10.1145/2380445.2380502.
[37] L. Zhang, B. Tiwana, R.P. Dick, Z. Qian, Z.M. Mao, Z. Wang, L. Yang, Accurate online power estimation and automatic battery behavior based power model generation for smartphones, in: Hardware/Software Codesign and System Synthesis, CODES+ISSS, 2010 IEEE/ACM/IFIP International Conference on, 2010, pp. 105–114.
[38] X. Chen, Y. Chen, Z. Ma, F.C.A. Fernandes, How is energy consumed in smartphone display applications? in: Proceedings of the 14th Workshop on Mobile Computing Systems and Applications, HotMobile '13, 2013.