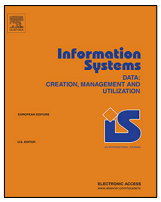
Information Systems 108 (2022) 102017

Contents lists available at ScienceDirect

Information Systems

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



A comparative study on the energy consumption of Progressive Web Apps

Stefan Huber [a,*], Lukas Demetz [a], Michael Felderer [b,c]

[a] University of Applied Sciences FH Kufstein Tirol, Andres Hofer-Straße 7, Kufstein, 6330, Austria
[b] University of Innsbruck, Technikerstraße 21a, Innsbruck, 6020, Austria
[c] Blekinge Institute of Technology, Valhallavägen 1, Karlskrona, 37179, Sweden

ARTICLE INFO

Article history:
Received 30 September 2021
Received in revised form 4 February 2022
Accepted 5 March 2022
Available online 8 March 2022
Recommended by Yannis Manolopoulos

Keywords:
Mobile software development
Energy-efficiency
Progressive Web Apps
Mobile cross-platform development

ABSTRACT

Progressive Web Apps (PWAs) are a promising approach for developing mobile apps, especially when developing apps for multiple mobile systems. As mobile devices are limited with respect to battery capacity, developers should keep the energy footprint of a mobile app as low as possible. The goal of this study is to analyze the difference in energy consumption of PWAs compared to other mobile development approaches. As mobile apps are primarily interactive in nature, we focus on UI rendering and interaction scenarios. For this, we implemented five versions of the same app with different development approaches and examined their energy footprint on two Android devices with four execution scenarios. Additionally, we extended our research by analyzing multiple real-world mobile apps to include a more practical perspective. Regarding execution environments, we also compared the energy consumption of PWAs executed in different web-browsers. The results based on sample and real-world apps show that the used development approach influences the energy footprint of a mobile app. Native development shows the lowest energy consumption. PWAs, albeit having a higher energy consumption than native apps, are a viable alternative to other mobile cross-platform development (MCPD) approaches. The experiments could not assert an inherent technological disadvantage of PWAs in contrast to other MCPD approaches when considering UI energy consumption. Moreover, the web-browser engine used to execute the PWA has a significant influence on the energy footprint of the app.

1. Introduction

Over the years, the processing power and overall hardware capabilities of smartphones have strongly improved. The lack of advancements in the accompanying battery technologies is still a limiting factor of mobile devices [1]. Users are, however, highly sensitive towards the energy-efficiency of their smartphones. For instance, users adapt their app usage patterns in relation to the battery level [2]. Also, the analysis of comments from app markets [3] indicate that the energy-efficiency of mobile apps is of significant importance to users. Building energy-efficient apps is challenging for mobile app developers. Practitioners lack intuition regarding the energy-efficiency of the source code they produce [4]. Also, the lack of tools [5] to analyze and diagnose energy issues is a hindrance to develop energy-efficient mobile apps.

Progressive web apps (PWAs) are a rather novel approach to develop mobile apps. PWAs are standard websites with additional functionality, such as offline availability and the possibility to install the PWA on the user's device. As such, PWAs are an example of mobile cross-platform development (MCPD) approaches. The idea of such development approaches is to deploy mobile apps to multiple mobile platforms from a single code base. They allow writing code once and run it anywhere [6]. App market analytics[1] indicate that these MCPD approaches are highly popular among developers, albeit they have not been researched intensively [7]. Besides MCPD approaches, mobile apps can also be developed natively, in which dedicated mobile apps have to be implemented for each mobile platform. Developing mobile apps for multiple platforms is a prominent challenge for developers [8] as dedicated code bases need to be maintained for each supported mobile platform.

Existing research showed that, depending on the focus of investigation, MCPD approaches use less energy than native development approaches in certain scenarios. For instance, in an intensive processing scenario [9] or when executing algorithm benchmarks [10], Apache Cordova uses substantially less energy

* Corresponding author.
E-mail address: mail@stefanhuber.at (S. Huber).

[1] https://appfigures.com/top-sdks/development/apps

https://doi.org/10.1016/j.is.2022.102017
0306-4379/

than Android Native. As PWAs are websites with extended functionality, they seem to be more lightweight than fully-fledged mobile apps and thus could exhibit a lower energy footprint. For instance, the Twitter PWA uses only 3% of the device storage in comparison to the Twitter Android Native app.[2] Therefore, as the main goal of this study, a comparison of the energy consumption of PWAs and other mobile development approaches with a dedicated focus on UI rendering and interactions, emerged. Based on this goal, two guiding research questions were derived:

(RQ1) How does the energy consumption of PWAs compare to other mobile development approaches, when executing typical UI interactions?
(RQ2) How does the used web-browser engine influence the energy consumption of PWAs?

To investigate the stated research questions, an experimental study was conducted. Within the study, the energy consumption (dependent variable) of five sample apps and 15 real-world apps, all created with different development approaches (independent variable), was measured. Additionally, for the PWAs, up to four different browser engines were used to examine their influence on the energy usage. The focus was set on UI interactions as mobile apps are primarily used interactively [11] and the aspect of UI interactions was identified as a gap in the research literature. For answering the research questions, the gathered observations were thoroughly analyzed using statistical tests. As an overall guideline for planning and executing the experimentation process, the handbook of Wohlin et al. [12] was followed.

This present paper is an extended version of previously published research [13]. Our initial experiments were solely based on sample apps developed by ourselves. To corroborate the results of these sample apps, real-world apps were selected as additional experimental subjects. The inclusion of real-world apps should shed more light on the question, if the energy consumption of a mobile development approach is inherent to the technology or if there is potential for app developers to influence the energy consumption, for instance, by applying energy-aware coding practices. For additional insights on the web-browser engines, two additional experimental subjects were assembled by wrapping raw web-browser engines inside native app wrappers. We not only extended our experimental setting, but also provide a more thorough background and related work section. As such, we provide a larger set of results, which we discuss in a broader background.

The main contributions of this study are: (1) an experimental comparison of the UI energy-efficiency of PWAs related to other mobile development approaches based on sample and real-world apps; (2) an experimental comparison of the energy-efficiency of up to four web-browser engines executing PWAs; (3) a discussion of the results from the perspective of mobile developers; (4) a comprehensive replication package[3] containing all research artifacts and statistical analysis steps.

The remainder of this paper is organized as follows. Section 2 lays out the background of this study, in which possible ways of developing a mobile app are described. Section 3 gives a broad overview of the related work in the field. Section 4 gives a detailed outline of the overall experimental setup to conduct the empirical study including our hypotheses and the following statistical analysis. Section 5 presents the results of this study. A thorough discussion of the results is followed in Section 6. Section 8 concludes this paper and provides possible directions of future research.

[2] https://developers.google.com/web/showcase/2017/twitter
[3] https://github.com/stefanhuber/pwa-energy-comparison

2. Mobile cross-platform development

Building mobile apps for multiple platforms is a challenging endeavor [8]. The default, native, approach to mobile development is to use the integrated development environment (IDE) and software development kit (SDK) [14] of the respective platform vendor. Today, two mobile platforms, Google Android and Apple iOS are dominating the mobile platform market. When following the path of native development, developers have to build and maintain unique code bases for each mobile platform on which the app should run. In addition, developers need to have deep knowledge in each of the platforms' subtleties.

Another prominent approach for mobile app development follows the concept of mobile cross-platform development (MCPD). The idea behind this is to build an app once and run it everywhere [6]. The creativity and expertise of developers lead to the emergence of a plethora of different MCPD approaches. Although, all these approaches share the same goal, they differ with respect to their internal functioning [15]. Commonly, MCPD approaches are classified into the 4 distinct categories [16]: hybrid, interpreted, cross-compiled and web.

Hybrid approaches use standard web-technologies to develop the user interface and behavior of the app. The web app is embedded in a native app wrapper [17], which provides unified access to features of the underlying mobile platform (e.g., access to the camera). This approach is termed hybrid, as is combines native and web development. Apache Cordova and Capacitor are examples of this approach.

Interpreted approaches also employ web-technologies, such as JavaScript and CSS. The user interface, however, is not based on HTML and is not rendered inside a web-browser. Based on an integrated interpreter, native user interface components are generated at runtime of the app [15]. React Native is a popular example following this category.

Cross-compiled approaches use a common programming language to develop a mobile app. The source code is then compiled into native code that can be executed on a mobile platform. According to [18], Flutter and Xamarin are popular examples of this approach, although, there is some disagreement in the literature (e.g., El-Kassas et al. [6] classify Xamarin as an interpreted approach).

Web approaches are based on standard web-technologies and are executed inside the default web-browser of a mobile platform. PWAs fall into this category [19].

Additionally, the literature distinguishes the category of model-driven mobile development approaches [18]. Although, model-driven approaches have shown interesting results in research [20], these approaches have not yet diffused into practice as app market analytics indicate.

PWAs are a rather new MCPD approach, first introduced in 2015 [21]. PWAs can be considered a conceptually new approach built on top of existing standard web-technologies, such as HTML, CSS and JavaScript. In addition to a mobile website, PWAs offer a set of extended functionalities enabling a native user experience for web-apps [22]. Offline availability, an installation procedure, push notifications and background sync distinguish a PWA from a regular mobile website. As PWAs are websites with enhanced functionalities, they can also be executed on a desktop computer, independent of the computer's operating system.

The extended features of PWAs are mainly based on the web app manifest [23] and a service worker [24] implementation. With the web app manifest, developers provide additional metadata (e.g., icons, title, colors, etc.), which is used by the operating system to display the PWA properly when installed. The service worker is a JavaScript program that is loaded and installed in the background, when a PWA is accessed for the first time. A

service worker intercepts network requests generated by the PWA and, by that, can implement a caching strategy to provide offline availability. In addition, an installed service worker can also handle push notifications.

In general, developers have similar options when developing PWAs and regular mobile websites. A common option is to use web-frameworks like Angular or React to build a PWA. Using frameworks, means to be dependable on standards prescribed by the framework. Alternatively, developers can use frameworkless approaches, such as, StencilJS or Svelte. These approaches compile PWAs in standard formats (i.e., web-components) reducing the load put on the devices executing the PWA.

Similar to regular mobile websites, PWAs are accessed via a URL. However, app markets start to include PWAs in their listings (e.g., Google Play Store [25] and Microsoft Store [26]). The installation procedure for a PWA, listed inside an app market, is then the same as for regular mobile apps.

Despite its advantages, PWAs still offer limited support for important functionalities, such as, access to the file system, access to user data (e.g., calendar entries) and other platform dependent APIs [19]. Others name a possible higher energy consumption as one of PWAs' drawbacks [27].

3. Related work

There is a large body of research in the field of energy optimization for mobile apps. Several techniques have been shown by researchers to be effective in optimizing the energy-efficiency of mobile apps.

As a developer, the first step to consider is the source code level. In literature, the concept of energy hotspots and energy bugs [28] have been introduced. These concepts represent different kinds of energy inefficiencies inside mobile apps, which substantially reduce battery life of a smartphone. Researchers have developed several approaches [28,29] to successfully detect such inefficiencies. Accompanying the detection of energy bugs, approaches to automatically repair [30] these kinds of bugs have been introduced.

Furthermore, the impact of refactoring common code smells on the energy consumption of mobile apps has been analyzed. Palomba et al. [31] have shown in an extensive study comprising 60 Android apps that four specific code smells have considerable impact on battery dissipation, and that refactoring has reduced the amount of energy used in all observed cases. Additionally, the application of performance-based patterns [32] have been demonstrated to reduce energy consumption in Android mobile apps. Cruz and Abreu [33] have extracted and aggregated 22 recurring design patterns from open source mobile apps, which have shown to improve the energy-efficiency of mobile apps in real-world settings.

Also, the energy-efficiency of several programming tasks has been investigated. The selection of data structures inside a mobile app should be guided by empirical results, as selecting inappropriate data structures can increase the energy use up to 300% [34]. Furthermore, Linares-Vásquez et al. [35] demonstrated energy-greedy API-usage patterns. Most notably, developers should be aware of the high impact of GUI, image manipulation and database related APIs, which were found to appear in 60% of all the analyzed cases. The selection of a MCPD development approach could introduce an energy overhead. In a comparison by Corbalan et al. [9], three execution scenarios were examined with different MCPD approaches. Results show that the energy consumption varies between scenario and MCPD approach, however, native development does not have the lowest energy footprint in all investigations. Also, Oliveira et al. [10] demonstrated that implementing parts of an Android native app with JavaScript or C++ instead of Java could also save energy.

Networking-related aspects tend to have a considerable impact on the energy consumption of mobile apps. From a user's perspective, choosing Wi-Fi over cellular could be considered as an energy saving practice, as cellular networking interfaces consume more energy due to energy overheads [36]. From a developer's point of view, techniques such as caching [37], aggregating [38] or timing [39] of network traffic have been demonstrated to be effective in reducing the consumed energy. Moreover, the selection of the networking protocol [40,41] should be considered as an energy optimization factor. It must be noted when considering PWAs, networking aspects such as service workers or caching have been shown to have no significant impact on the energy consumption [42,43].

In addition to optimizing general networking tasks, the offloading of complex tasks to a cloud environment could be beneficial for reducing the energy footprint. Cloud offloading is considered an optimization problem to minimize computation energy against the network transmission energy usage of a smartphone [44]. However, before code offloading and remote execution can be used, a partitioning approach is required. The MAUI architecture introduced by Cuervo et al. [45], as an example, enables a fine-grained partition at the method level, and a more coarse-grained partition at the process level, and has been shown to be an effective approach for code offloading to reduce energy consumption.

Notable improvements considering the battery drain can be gained by adapting the user interface. The colors used for rendering the user interface on OLED screens [46] can be considered a primary factor for optimization. Techniques such as transforming color values [47,48] of user interfaces, while preserving the attractiveness, have been proposed as effective. Additionally, dimming parts of the screen, which are not relevant for users, has also a considerable impact. Chen et al. [49] proposed FingerShadow, an approach to dim the screen around the area of the fingers of a user. ShiftMask, another approach proposed by Lin et al. [50], could save battery by dimming parts of the screen based on visual acuity. Besides color adaptations and dimming, the design of the user interface can be optimized from a usability perspective to save energy. Vallerio et al. [11] have proposed and evaluated techniques such as hot keys, user input caches or content placements to increase user efficiency and reduce energy usage.

Smartphones are equipped with a rich set of sensors. These sensors can be used by mobile apps to enrich the user experience with position, motion or other environmental data. The ineffective use of these sensors can cause increased battery drainage [51]. Especially for location-based services, researchers have demonstrated various approaches to partially substitute the energy-greedy GPS sensor with other sensors or sensor combinations [52,53], without sacrificing location accuracy. A study by Ciman and Gaggi [54] focuses on the energy consumption of various sensors within apps developed with MCPD approaches and native apps for iOS and Android. The study shows that apps developed using MCPD approaches always exhibit a higher energy consumption than native apps.

Prior work has created a large amount of optimization techniques for reducing the energy consumption of mobile apps. The related work presented in this section shows a strong focus on the Android Native development approach. The studies concerning the energy consumption of MCPD approaches [9,10,42,43,54] have emphasized on aspects such as networking, sensors and different processing scenarios or benchmarks. Energy-related aspects of UI rendering and interaction scenarios of MCPD approaches are yet a research gap in the literature.

4. Research method

To address the research questions RQ1 and RQ2, a two-step approach was followed. In the first step, sample apps (Section 4.1) were used as experimental subjects and in a subsequent second step, real-world apps (Section 4.2) were used to corroborate the gathered results. A set of repeatable UI interaction scenarios (Section 4.3) was defined, which were used to interact with the mobile apps (Section 4.6). Throughout the automatic executions of these interaction scenarios, the energy consumption of the mobile apps was measured (Section 4.4) on two mobile devices (Section 4.5). As a final step, a statistical analysis was used to answer the research questions (Section 4.7 and Section 4.8).

4.1. Sample apps

Based on the idea of a contact management app, five different versions of a sample app were implemented as experimental subjects of the study. Each app is implemented with a different development approach. To compare the apps regarding their energy consumption, each app provides exactly the same functionalities, screens and UI elements. The look and feel of all apps is based on the Google material design guidelines[4] to increase the comparability of the implementations.

Popular and widely-used mobile development approaches were selected to implement the sample apps. As development approaches Android Native and the four MCPD approaches PWA, Capacitor, Flutter and React Native were used. All the development approaches are used by and supported by large developer communities, visible on websites such as GitHub and Stack Overflow.

Furthermore, the PWA was executed in different web-browser runtime environments. The default web-browser for PWAs on Android is Google's Chrome web-browser. The Firefox web-browser was selected as an alternative web-browser to install and execute PWAs on Android.

It has to be noted that the installation of a PWA with Chrome leads to the creation of a WebAPK [55]. This procedure creates a self-contained app on the Android device. In contrast, for Firefox, the PWA is stored as a web-browser bookmark with an icon on the home screen. To investigate whether a fully-fledged web-browser comes with an overhead in energy consumption, two different render engines were wrapped inside Android Native apps. Thus, a similar state, which was established with Chrome's WebAPK, was assembled for Firefox. The wrapper apps were configured to load the PWA sample app implementation. One app uses the Gecko WebView,[5] which is based on the Gecko render engine underlying the Firefox web-browser. The other app uses the default Android System WebView[6] based on Chrome.

4.1.1. PWA & capacitor

Both the PWA and the Capacitor app are based on the same web app implementation. The web app was built with the Ionic web framework and the StencilJS web-component compiler. The Ionic framework provides reusable UI components, whose look and feel resembles the look and feel of native Android or iOS components. The framework comprises all the required functionality without the need for additional libraries.

Each screen of the app was realized using a `Stencil` component, which was compiled to a web-component. Navigation between screens was realized using `ion-router`. The drawer menu was implemented using the `ion-menu` component. The list view on the start screen was implemented using the `ion-virtual-scroll` component, which is a virtual scrollable list implementation for the web, emulating a scroll behavior inside web-browsers. The form screen was created using `ion-input` components.

In addition, for the PWA, a service worker was implemented, and a web manifest was provided to enable offline usage of the PWA and to make it installable on the smartphone. The service worker delivers all network requests from cache. As a result, no network requests were executed by the PWA, allowing full offline usage.

Capacitor is a hybrid mobile development approach and thus is a wrapper around a web app inside an Android native app. The transformation of the web app into a hybrid Capacitor app did not require any additional changes.

4.1.2. Flutter

Flutter offers a rich set of stateful and stateless widgets to build UIs, which mimic the look and feel of Android or iOS native UIs. The navigation drawer was realized with the `Drawer` widget. For navigating between screens, the `Navigator` service was used. The scrollable list was implemented with a `ListView` widget. For the form elements, `TextField` widgets were used.

4.1.3. React native

React Native provides only a few core UI components. To build the envisioned contact management app, several external libraries were required. The navigation drawer was realized with the `react-navigation-drawer` library. For navigating between different screens of an app, React Native does not provide a default approach. Therefore, we used the external libraries `react-navigation` and `react-navigation-stack`. React Native provides the `FlatList` component, which we used to display the sample entries as a scrollable list. This UI component is an abstraction over the native implementations of virtual scrollable lists, such as the `RecyclerView` on Android. The form elements were built with `InputField` components. Respective styling was applied to get the Material look and feel.

4.1.4. Native Android

The native Android app was implemented based on the Android SDK without additional libraries. The two available screens were set up as an `Activity`. Navigation between screens was implemented by using `Intents`. The start screen with the scrollable list of contact entries was implemented using a `RecyclerView`, which is the default Android component for virtual scrollable lists. For binding the `RecyclerView` with a data source, an `Adaptor` was implemented, which provides sample entries. Android provides a `NavigationView`, which can be put inside a `DrawerLayout` to realize the drawer menu. The form screen was built using basic form widgets provided by the SDK.

4.2. Real-world apps

To corroborate the results gathered with the sample apps, three real-world apps for each development approach were collected as experimental subjects. The interaction scenario i2 (see Section 4.3) was used for interacting with the apps. Therefore, as a primary selection criterion, the real-world app must comprise a list-oriented screen to execute the scrolling gestures. Additionally, it was defined that each list entry should be composed of textual and image data. In Fig. 1 an overview of five exemplary app screens is provided. All 15 selected apps had a screen with a similar appeal.

Besides selecting apps with a respective list screen, the development approach had to be verified. In a first step, a candidate set

[4] https://material.io/design

[5] https://wiki.mozilla.org/Mobile/GeckoView

[6] https://developer.android.com/guide/webapps/webview

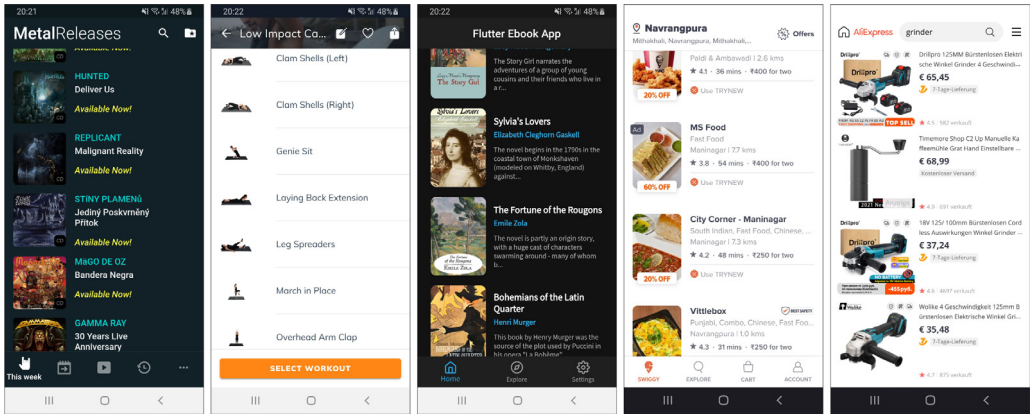


Fig. 1. Screenshots of 5 exemplary real-world app screens, from left to right: MetalReleases, Sworkit, Flutter Ebook App, Swiggy, AliExpress.

Table 1
Real-world app sources.

Development approach	source
Android Native	https://f-droid.org/en/packages
Capacitor	https://ionic.io/resources/case-studies
Flutter	https://github.com/Solido/awesome-flutter
PWA	https://github.com/hemanth/awesome-pwa
React Native	https://github.com/ReactNativeNews/React-Native-Apps

Table 2
Real world mobile apps.

App name	Development approach
NewPipe	Android Native
F-Droid	Android Native
Tiny Weather Forecast	Android Native
Ulangi	React Native
Bus Timetable	React Native
YumMeals	React Native
Flutter Ebook App	Flutter
Metal Releases	Flutter
No Man's Sky	Flutter
AliExpress	PWA
Versus	PWA
Swiggy	PWA
Sworkit	Capacitor
JustWatch	Capacitor
Untappd	Apache Cordova

of potential apps was created. For Android Native potential apps were selected from the F-Droid catalog of free and open source apps. The potential apps for MCPD approaches were selected using awesome lists from GitHub.[7] Awesome lists are community curated lists of things relevant to a certain topic. Especially for programming languages or development frameworks, the community creates lists of showcase applications to demonstrate the advantages of certain technologies. Only for Capacitor we could not find an appropriate awesome list with showcase apps, therefore, we consulted the website of the company behind the framework and used the showcase apps from there. The list of the app sources is shown in Table 1.

For the verification of the development approach of an app, a twofold approach was taken. Preferably, the source code is open source and can be examined. For React Native, Android Native and Flutter, adequate open source apps could be found. For Capacitor, no adequate open source apps could be identified, therefore, potential candidate apps were decompiled and inspected whether they resemble a typical Capacitor app. To determine whether an app was built with Capacitor, we used an approach similar to Ali and Mesbah [56] (i.e., searching for certain classes which are part of the development framework). This approach led to the identification of two adequate Capacitor apps. In addition to the Capacitor apps, one Apache Cordova app was included. Apache Cordova is basically built on the same technical approach [57], by wrapping a web-app inside a native app. Thus, we considered it as appropriate to include this app. For clarity, the Capacitor and Apache Cordova apps are referred to as hybrid apps in the following. PWAs did not require a verification step. Table 2 shows the selected real-world mobile apps.

4.3. Interaction scenarios

Four different interaction scenarios (i1-i4), which can be executed automatically and repeatably on the test devices, were set up. For setting up the scenarios the guidelines provided by Cruz and Abreu [58] were carefully followed. Therefore the Android UI Automator testing framework[8] was selected for codifying the interactions. In addition the interactions are based on pixel coordinates and not on energy-greedy UI element lookups within the UI hierarchy.

The interaction scenarios codify typical user interactions as executable scripts. The script can be compiled and deployed to any Android device. Moreover, when connected to an Android device via the Android Debugging Bridge (adb), the controlling computer can start the interaction scenarios via command line. Related research in the field of energy profiling is based on a similar setup, e.g., Cruz and Abreu [32] identified the effect of performance optimizations on the energy footprint of mobile apps and Linares-Vásquez et al. [35] used typical usage scenarios of mobile apps to detect energy-greedy API patterns.

In the following, the four interaction scenarios are described with reference to the sample app:

- **i1** Open the navigation drawer menu (change between first and second screen in Fig. 2) with a left-to-right swipe gesture, and after one second close it with a tap outside the drawer menu. The interaction is repeated five times.
- **i2** Scroll down the list entries (first screen in Fig. 2) with five consecutive bottom-to-top swipe gestures.
- **i3** Tap on the top-right "Add Entry" menu icon (first screen in Fig. 2) to navigate to the entry form screen (third screen

7 https://github.com/topics/awesome

8 https://developer.android.com/training/testing/ui-automator

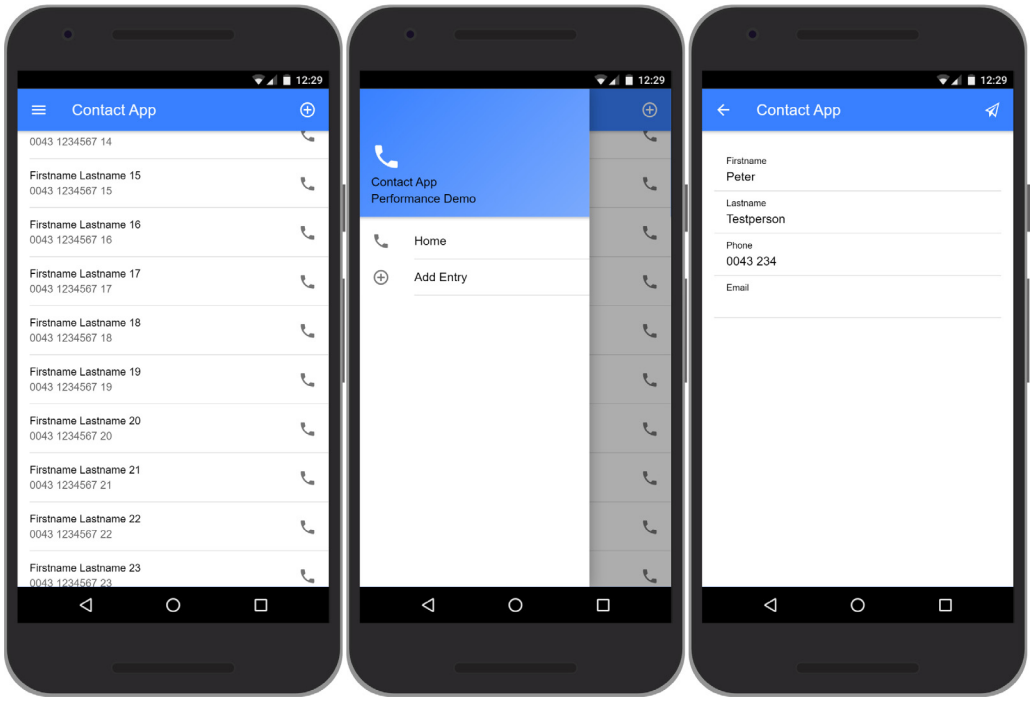


Fig. 2. Overview of the different screens of the sample contact app [13].

in Fig. 2) and after one second tap on the top-left back button to navigate back to the list screen. The interaction is repeated five times.

i4 Enter form data for the input fields "Firstname", "Lastname" and "Phone" (third screen in Fig. 2).

Within the respective implementations of the sample apps, all four interaction scenarios can be executed, as the respective user interface corresponds with the user interactions from the scenarios. For controlling the real-world apps, only the scenario i2 could be used.

4.4. Energy measurement

For measuring the energy consumption of the sample apps used in the experiments, the `batterystats` service was used. The service is available on every Android device and ensures, thus, the reproducibility of the experiments.

For each Android device, a power profile specifies power states of all hardware components of the device. The `batterystats` service is using power models [59] to estimate the energy consumption by combining the utilization of a hardware component with the corresponding power profile. This allows to estimate the power consumption of single hardware components, such as, display and CPU.

The service estimates the energy consumption in milli ampere hours (mAh). To be comparable to related research, we converted the measured values to Joule. For the conversion also the voltage values were used from the battery statistics and the following formula was used:

$$Energy\ [\mathrm{J}] = \frac{Charge\ [\mathrm{mAh}]}{10^6} * Voltage\ [\mathrm{V}] * 3600$$

The battery statistics are estimated for the device as a whole and additionally for each active app running on the device. In this study, the battery statistics at the app-level were used.

4.5. Test devices

For being representative of modern smartphones, only devices which at least support Android 10 (released in March 2019) were selected. Additionally, the device's hardware specification was used as a proxy. Thus, a lower-end — Samsung Galaxy S9 — and a higher-end — Samsung Galaxy S21 — device were used. Table 3 details the specifications of the selected devices.

4.6. Experimental procedure

Before the experiments could be executed, the devices were set up accordingly. As such, we performed the following steps:

1. The mobile devices were connected to the controlling computer via USB and all installations (i.e., sample apps, real-world apps, interaction scenarios) and accompanying configurations were performed.
2. The mobile devices were put into airplane mode and all active apps were closed. Additionally, services which were not required for the experiments, such as GPS or Bluetooth, were deactivated. Wi-Fi was enabled as it is required for controlling the execution of the experiments.
3. Before starting the experiment, it was ensured that the battery is fully loaded and the brightness is dimmed to approximately 50%. A higher brightness level would lead to a faster discharging of the battery. To allow for a longer run of the test procedure without reloading and still allow the operator to monitor the experiment, a lower brightness level was set. As a software-based metering approach is used, the measured energy consumption of the apps under scrutiny were unaffected by the brightness level.
4. Finally, the device was connected via Wi-Fi with the controlling computer and the USB cable was disconnected. From this point on, the device was discharging the battery.

Python scripts were used on the controlling computer to operate the apps on the connected mobile device and to start and

Table 3
Mobile device specifications.

Class	Device	CPU	RAM	Android	Battery capacity
Lower-end	Samsung Galaxy S9	Snapdragon 845 4 × 2.7 GHz/4 × 1.8 GHz	4 GB	10	3.000 mAh
Higher-end	Samsung Galaxy S21	Snapdragon 888 1 × 2.9 GHz/3 × 2.80 GHz/4 × 2.2 GHz	8 GB	11	4000 mAh

stop the interaction scenarios. For gathering the energy consumption data during the execution of the interaction scenarios, the `batterystats` (see Section 4.4) service was used. Before starting an interaction scenario, the `batterystats` service was reset and after the run the resulting measurements were collected.

For controlling the experiments with the sample app implementations, the Python script started an app and executed the four interaction scenarios (i1-i4). Afterwards, the app was stopped, and the next app was started. For the real-world app experiments, the Python script only executed the i2 interaction scenario per app. The correctness of the experiment execution was continuously monitored by an operator.

To answer RQ1 and RQ2, the test procedures were executed on each device with the sample app implementations and the real-world apps. The procedure yielded 1.920 samples from the sample app implementations and additionally 1.080 samples regarding the real-world app executions. It is noteworthy that the sample PWA was executed in four different web-browser engines (Chrome, Firefox, two WebView wrappers), yielding 240 per engine. Additionally, the real-world PWAs were executed in Chrome and Firefox, yielding 180 samples per web-browser.

4.7. Hypotheses formulation

Based on the two research questions, statistical hypotheses were derived. The focus of RQ1 is to test whether there is a difference in energy consumption of PWAs compared to other mobile development approaches. Thus, a two-tailed null hypothesis ($H1_0$) and corresponding alternative hypothesis ($H1_a$) was formulated:

$$H1_0 : \mu_{PWA\ (chrome)} = \mu_{MDA}$$

$$H1_a : \mu_{PWA\ (chrome)} \neq \mu_{MDA}$$

where $\mu_{PWA\ (chrome)}$ represents the mean energy consumption of the PWA executed in the default Chrome web-browser. μ_{MDA} represents the mean energy consumption of the other mobile development approaches. For simplicity, μ_{MDA} is considered as a variable for each of the four other considered approaches (i.e., Android native, Flutter, Capacitor and React Native). The same hypothesis is used for testing the sample apps and also the real-world apps.

$H1$ was tested with the sample apps for each of the four UI interactions with each of the four mobile development approaches against the PWA counterpart (executed in Chrome). $H1$ was also tested with the real-world apps vs the PWA (Chrome) with respect to the i2 interaction scenario. When considering the two test devices, overall 42 hypothesis tests were executed with respect to $H1$.

RQ2 focuses on differences in energy consumption of the web-browser engines executing the PWA. Thus, the following two-tailed null hypothesis ($H2_0$) and corresponding alternative hypothesis ($H2_a$) were formulated:

$$H2_0 : \mu_{PWA\ (chrome)} = \mu_{PWA\ (alternative\ web-browser)}$$

$$H2_a : \mu_{PWA\ (chrome)} \neq \mu_{PWA\ (alternative\ web-browser)}$$

where $\mu_{PWA\ (chrome)}$ represents the mean energy consumption of the PWA execution in the default Chrome web-browser. $\mu_{PWA\ (alternative\ web-browser)}$ represents the mean energy consumption in the alternative web-browser engines, that is, Firefox, Gecko WebView and Android default WebView.

$H2$ tested the PWA sample app executed in Chrome against the PWA in Firefox and the two embedded web-browser engines (Chrome WebView and Gecko WebView). For the sample app, all four interaction scenarios were included. Additionally, $H2$ also tested the i2 interaction scenario executed with the real-world PWAs in Chrome and Firefox. When considering the two devices and overall of 24 hypothesis tests were executed with respect to $H2$.

Table 4
Calculated α levels for hypothesis tests.

	Sample app implementations	Real-world apps
$H1$	0.0125	0.0125
$H2$	0.0167	0.05

4.8. Data analysis

The test executions on the two devices produced 3.000 samples for the energy consumption. Each sample consists of the energy consumed by one app executing one interaction scenario on one device in Joule.

To select a corresponding parametric or non-parametric statistical test, a test for normality of the sample distributes was made by applying the Shapiro–Wilk test [60]. According to the test with an alpha level below 0.05 ($\alpha = 0.05$) 56% of the samples of the sample app implementations and 86% of the samples of the real-world app were not normally distributed. Therefore, the non-parametric Mann–Whitney U test [61] was selected as an appropriate hypothesis test.

The samples for the PWA executed in Chrome were used in multiple statistical tests, which leads to an increased chance of incorrectly rejecting a null hypothesis. By applying the Bonferroni correction [62], a stricter significance level was defined to counteract this problem. The calculated α values for the hypothesis tests regarding $H1$ and $H2$ with sample app implementations or real-world apps is shown in Table 4.

To go beyond a sole p-value interpretation, the final step in the analysis chain was the calculation of effect sizes for the corresponding hypothesis tests. Cliff's Delta [63] effect size was used in this study as it is a non-parametric effect size and was selected based on the results of the tests for normality of the samples. The resulting effect size lies in the closed interval $[-1, 1]$. A value near 0 marks a high overlap between the values of the sample distributions of the two observed groups, which means a low difference between the groups. Values near 1 or -1 mark the absence of an overlap of the sample distributions. The maximum delta of 1 or -1 occurs if there is no overlap of the distributions. For this study, a positive value for the effect size means that the mobile development approach has a higher energy consumption than the corresponding PWA executed in Chrome. A negative value for the effect size means the opposite.

All statistical analysis steps executed with regard to the real-world apps are based on an aggregation of the individual app

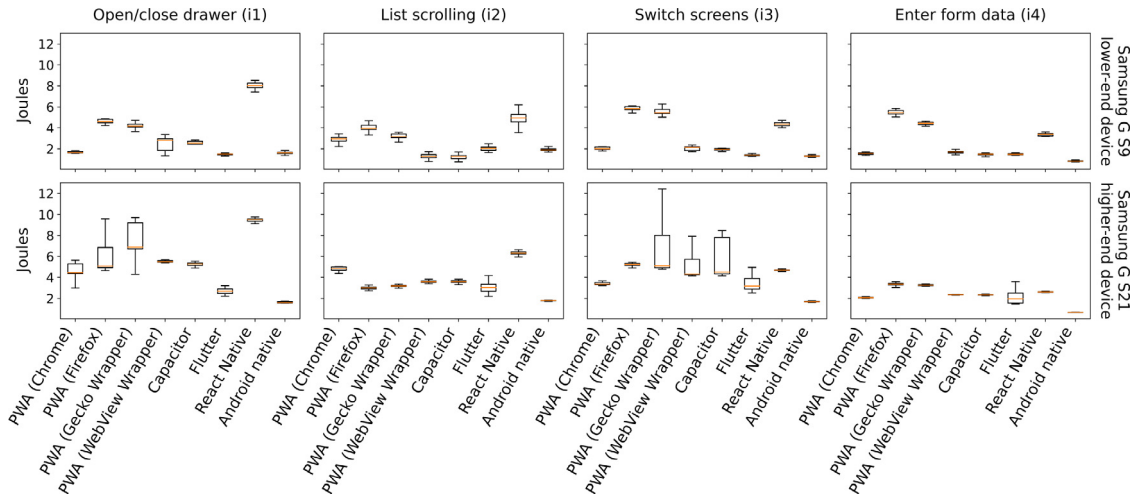


Fig. 3. Energy consumption (Joules) results of different UI interaction scenarios.

measurements. The individual app measurements were aggregated by the underlying development approach (e.g., the app measurements of NewPipe, F-Droid and Tiny Weather Forecast were aggregated as Android Native).

5. Results

This section reports the results obtained from the experiment execution. The results considering RQ1 are separated by reporting on the sample apps (Section 5.1) and the real-world apps (Section 5.2). Similarly, the results regarding RQ2 are reported in Sections 5.3 and 5.4.

5.1. RQ1: Sample apps

In Fig. 3 an overview of the results concerning the sample app implementations is provided. The mobile devices, which executed the experiments, are shown row-wise and the interaction scenarios (i1-i4) are shown column-wise. For each chart in the grid, the *x*-axis represents the mobile development approaches and the *y*-axis represents the energy used in Joule.

For a comprehensive overview, Fig. 4 shows a heatmap of the calculated effect sizes regarding the sample app experiments. Each of the two maps represents one of the mobile devices used in the experiment. The interaction scenarios executed within the PWA (Chrome) are shown row-wise. The corresponding development approaches are displayed column-wise and the associated effect size is listed at the intersection. The cells are color coded and mark the direction of the difference. A red color value marks a higher energy consumption and a blue color value marks a lower energy consumption of the PWA executed in Chrome regarding the corresponding development approach. Moreover, the brightness of the color values reflects the magnitude of the difference.

Considering RQ1, the results regarding the sample app experiments show a substantial difference between the mobile development approaches. Noticeable is the difference between React Native and the PWA executed in Chrome. For all comparisons, React Native has a significantly higher energy consumption, that is, all $H1_0$ hypotheses in the context of the PWA (Chrome) and React Native were rejected. This is also reflected in the calculated effect sizes shown in the sixth columns of the heatmaps in Fig. 4.

In contrast to React Native, for Android Native the opposite can be observed. In all but one cases, Android Native exhibits a significantly lower energy consumption than the PWA executed in Chrome. Interaction scenario i1 on the Samsung S9 has a lower energy consumption by interpreting the descriptive results, but has no significance. The stark difference between Android Native and the PWA can also be examined in the last columns of the heatmaps in Fig. 4.

The Flutter sample app implementation tends to have a lower energy consumption compared to the PWA (Chrome) counterpart. Five of the eight statistical tests show a significantly lower energy consumption. Also, the other three comparisons show a lower energy consumption, but with no significance. The calculated effect sizes are shown in the fifth column of the heatmaps in Fig. 4.

Finally, the Capacitor development approach shows rather varying results with regard to the sample app experiments. The i1 interaction scenario exhibits a significantly higher energy consumption for Capacitor on both tested devices. In contrast, interaction scenario i2 a significantly lower energy consumption in regard to the PWA. However, for i3 and i4 Capacitor has lower energy consumption on one device (Samsung S9) and higher energy consumption on the other device (Samsung S21). The variability of the results is also visible in the fourth column of the heatmaps in Fig. 4.

5.2. RQ1: Real-world apps

To extend and also corroborate the results from the sample app implementations, real-world apps were used as experimental subjects. In Fig. 5 a descriptive overview of the i2 interactions scenario applied to three real-world apps per development approach is shown. In the style of Fig. 3, the results regarding the two mobile devices are shown row-wise and the development approaches are listed column-wise. An exception is the first column, which gives an aggregated view on the results. Results for individual apps are aggregated into one box plot by the identified development approach. For each chart in the grid, the *x*-axis lists the development approach or app used and the *y*-axis shows the energy measured in Joule.

In addition, Fig. 6 shows a heatmap of the calculated effect sizes regarding the real-world app experiments. The heatmap is based on the aggregation of the data by the identified development approach of the real-world apps. The rows of the heatmap show the interaction scenario i2 of the PWA (Chrome) and the used mobile device. The columns show the corresponding development approaches with the calculated effect size values at the intersecting cells. The colors encode direction and magnitude of the values, analogous to the style of Fig. 4.

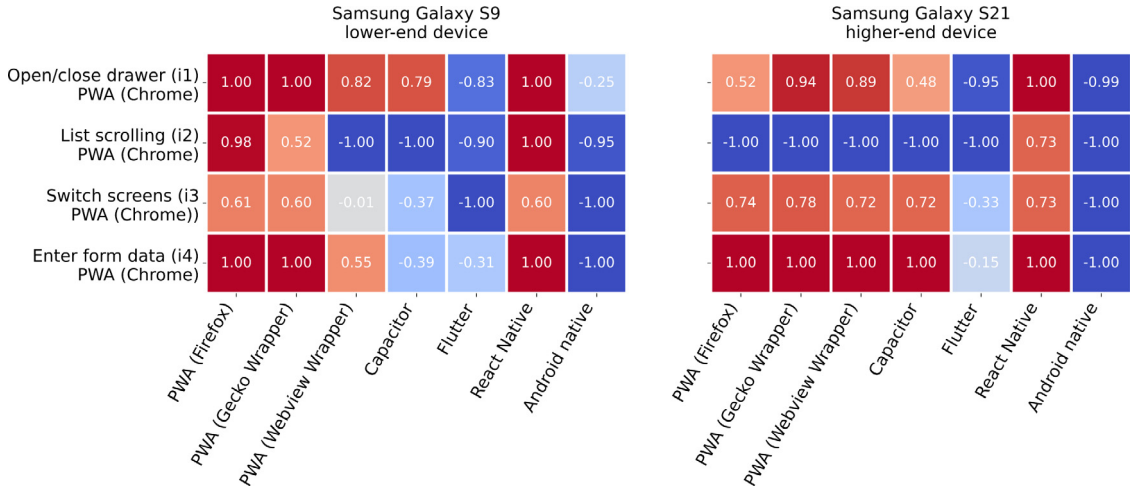


Fig. 4. Heatmap of the calculated Cliff's delta effect sizes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

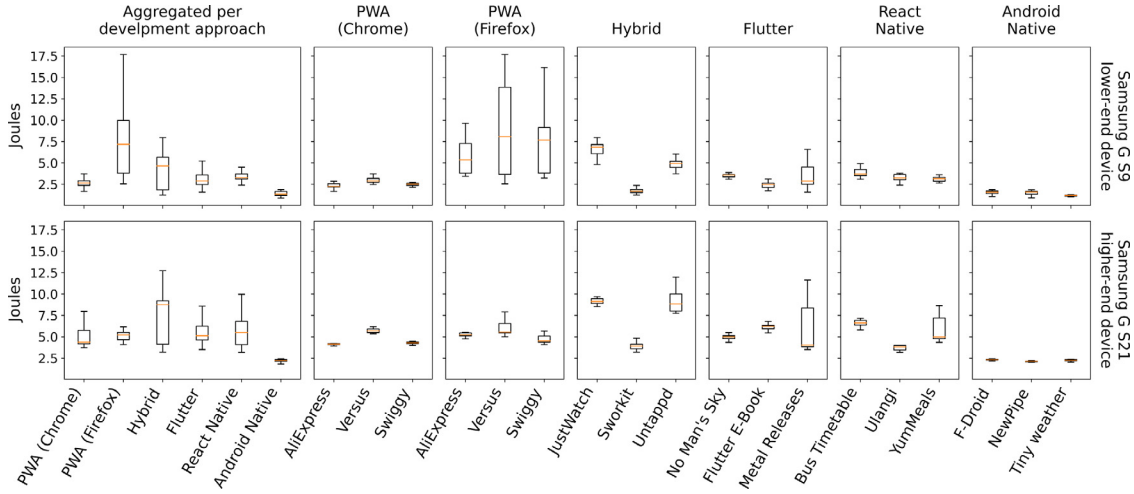


Fig. 5. Energy consumption (Joules) of real-world app results regarding the i2 interaction scenario.

Regarding RQ1, the results from the angle of the real-world app experiments cannot confirm all findings from the experiments with the sample apps. A clear confirmation of the sample app results can be asserted for the difference between the PWAs executed in Chrome and the Android Native apps. On both devices, Android Native has a significantly lower energy consumption than the PWAs.

The results from the sample apps between the PWAs (Chrome) and the React Native apps only hold for the lower-end device. For the higher-end device, no significant differences in energy consumption can be asserted. Also, the magnitude of the difference between React Native apps and the PWAs (Chrome) is lower compared to the sample apps, as shown in the fourth column of the heatmap in Fig. 6.

For Flutter, the results are not in line with the results from the sample apps. In comparison with the real-world apps, Flutter exhibits a higher energy consumption than the PWAs executed in Chrome, although with no significance. The effect sizes are shown in the third column of the heatmap in Fig. 6.

The results concerning the hybrid apps are also not in line with the sample app results. The PWAs (Chrome) consume significantly less energy than the hybrid apps on both devices. The difference is only of a lower magnitude, shown in the second column of the heatmap in Fig. 6.

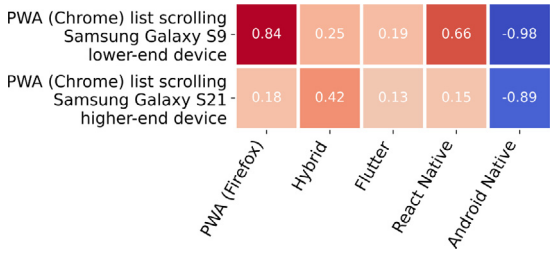


Fig. 6. Heatmap of the calculated Cliff's delta effect sizes regarding real-world apps.

5.3. RQ2: Sample apps

Considering the results for the sample app experiments with respect to RQ2, the PWA executed in Chrome has a significantly lower energy footprint compared to the alternative web-browser engines for most of the cases. However, a notable exception is interaction scenario i2. The PWA executed in Chrome has a significantly higher energy consumption than the alternative web-browser engines for this scenario on the higher-end device and for the Android default WebView on the lower-end device.

5.4. RQ2: Real-world apps

The results for the real-world app experiments show how the PWAs executed in Chrome compare to the PWAs executed in Firefox. The interaction scenario i2 was the only scenario used with the real-world apps. The results indicate that the PWA executed in Chrome has a lower energy footprint. Statistical significance could only be asserted on the lower-end device.

6. Discussion

This section starts with a discussion of the results (Section 6.1), follows with a discussion of implications (Section 6.2) and ends by providing an outlook and possible directions for future research (Section 6.3).

6.1. Interpretation of results

Contrary to the authors' assumption that PWAs are more lightweight than fully-fledged mobile apps and thus consume less energy, the Android Native development approach had a lower energy consumption than the PWAs. As this holds for sample apps as well as real-world apps, an inherent advantage of Android Native development could be attested with regard to UI energy-efficiency, compared to PWAs and to other examined MCPD approaches.

A noticeable result of this study is the lower energy footprint of PWAs executed in Chrome with regard to the React Native development approach. All experiments have shown a lower energy consumption of the PWAs than the React Native apps (only one result shows no significance). From a technological perspective, React Native is creating native UI components, similarly to Android Native. Android Native, however, has shown to have a lower energy footprint than the PWAs. It could be concluded that the integrated interpreter, which creates native UI components from JavaScript execution at runtime, is responsible for this overhead. A potential approach to optimize the energy efficiency of React Native could be to pre-render user interfaces at built time of an app. To the best of our knowledge, such an approach is not yet available for React Native. Although, for the underlying JavaScript framework React, server-side rendering approaches, such as Next.js, are available and widely-used.

The results showed a noticeable outlier with regard to the sample app PWA executed in Chrome on the higher-end device. By closely monitoring the scrolling behavior during the experiment execution, a faster scroll speed for Chrome on the higher-end device, in contrast to the other web-browser engines, was noticeable. An examination of the underlying source code of the virtual scrolling component for the sample app, showed that the scroll speed is, amongst others, determined by the screen height, but not by any web-browser specific adaptations. It must also be noted that this faster scroll speed could not be observed in the default system WebView Wrapper, which is based on Chromium and also not on the lower-end device. This indicates, that web-browser engines have minor subtleties in their implementations, which are hard to reproduce and tackle by UI component developers. In the specific case of our experiment, the faster scrolling produced a higher workload for the web-browser engine and thus a higher energy footprint.

Concerning Flutter and the hybrid approach, the results of the sample apps showed a significantly lower energy consumption for the i2 interaction scenario in comparison to the PWA executed in Chrome. The real-world apps in contrast showed opposing results. From a sole interpretation of the visual appeal of the selected real-world apps, no striking differences could be identified by us, which could explain a higher energy consumption for Flutter and the hybrid approach. From a developer perspective, subtle optimization techniques, such as caching or pre-loading, could be applied and thus could result in lower energy footprints for PWAs or web applications. These results allow for the conclusion that Flutter and the hybrid approach have no inherent technological advantage when considering the energy-efficiency of user interfaces.

Considering RQ2, the results indicate that the default web-browser for PWAs on Android (Chrome) has the lowest energy consumption compared to alternative web-browser engines. As explained above, only for the sample app on the higher-end device, the list scrolling interaction scenario i2 showed a significantly lower energy consumption for the alternative web-browser engines compared to Chrome. Also, for the real-world apps on the higher-end device, a slightly faster scroll speed was observable in Chrome than in Firefox. A sole descriptive or visual interpretation of the data would allow for the interpretation, that the web-browsers are almost on a par with each other when considering the energy-efficiency. The faster scrolling in Chrome has not produced an energy overhead compared to Firefox, which is another indication that Chrome is using less energy than Firefox.

Furthermore, the use of the Gecko web-browser engine in a standalone setting without the Firefox context did not show any noticeable improvements with respect to the energy consumption. In addition, the execution of the PWA in the WebView wrapper caused a significantly higher energy consumption in most cases. The list scrolling interaction scenario i2, however, led to a substantially higher energy consumption on both devices. At least on the higher-end device, the faster scrolling is a possible explanation, which is, however, not noticeable on the lower-end device. As a further insight, the results showed that the WebAPK created by Chrome is not just a default system WebView wrapped inside an Android Native app. A further investigation on how WebAPKs could be used by alternative web-browser vendors to elicit opportunities for energy savings is necessary.

6.2. Implications

In general, MCPD approaches provide the advantages of lower development cost and lower maintenance effort, as apps are deployable to multiple platforms. Developers face a difficult decision when choosing among the plethora of available MCPD approaches. Rieger and Majchrzak [64] introduced a comprehensive framework based on weighted criteria to support the decision-making process. Although, the framework does not contain explicit criteria for energy-efficiency, we argue for the importance of energy-related aspects in deciding for a development approach.

In related work, considering the energy consumption of MCPD approaches, Oliveira et al. [10] have shown that in 75% of all examined benchmarks, JavaScript (executed inside a hybrid app) was more energy-efficient than Java inside an Android Native app. Also, in Corbalan et al. [9] regarding an intensive processing scenario, the hybrid app used substantially less energy compared to an Android Native app. The results of this study focused on the energy-efficiency of UI related aspects of apps and showed an inherent advantage of Android Native for UI-intensive apps. With respect to the related work, PWAs could still be considered an alternative to native development when focusing on energy saving, if complex processing tasks are a requirement for the app. The use of WebAssembly could even increase the energy-efficiency of complex processing tasks for PWAs.

In comparison to other MCPD approaches, we consider PWAs a viable alternative with respect to the UI energy-efficiency. Based on the results, PWAs are preferable over React Native. The results of Flutter and the hybrid development approach did not show an

inherent technological advantage in comparison to PWAs. Therefore, we conclude that energy-aware coding practices within MCPD approaches, which also have been researched for Android Native development, are an interesting area for future research.

Other main advantages of PWAs are (1) that they are available for desktop scenarios and (2) that they can easily be transformed into a hybrid app. The transformation into a hybrid app would extend the available app features (e.g., access to a user's calendar entries), which are not permitted to a PWA. However, as especially the results from the real-world apps suggest, this transformation could go along with a decrease in energy-efficiency and must be carefully evaluated.

The experiments also showed that minor web-browser subtleties could lead to different behavior of UI components and therefore differences in energy-efficiency among devices. Thus, mobile developers, who want to optimize the energy-efficiency of user interfaces, should be aware of device dependent factors and also test optimizations on different devices to be sure that optimizations can be generalized.

As an overall finding with respect to RQ2 and also as a suggestion for users, the results of this study indicate that using the default Chrome web-browser with PWAs is a sensible choice when considering energy-efficiency.

6.3. Outlook

The results of the study identified several factors, which could influence the energy-efficiency of UI components of PWAs. One aspect is the web-browser engine and also minor subtleties of the web-browser implementation, which could cause different behaviors among devices with the same web-browser. Another important aspect is the implementation of UI components and the potential applicability of energy-aware coding practices.

The design space for tuning UI components with regard to energy-efficiency is extensive, for example, appearance related aspects, such as colors, styling, shadow or transparency effects; interactivity related aspects, such as animation speed or animation effects; implementation related aspects, such as applying WebAssembly or multi-threading. Increasing energy-efficiency by preserving visual attractiveness of UI components appears as a complex optimization problem. Experimental studies in this field would help mobile developers to tune certain UI elements for energy-efficiency, or even allow framework developers to elicit optimization opportunities.

To extend the usefulness of such studies also iOS should be integrated as a mobile platform. In addition, other app-enabled and battery-powered device classes should be considered. Devices such as smartwatches, smart glasses and portable VR/AR headsets would benefit from insights in the field of energy-efficient user interfaces, especially in the context of cross-platform development.

7. Threats to validity

This section reports on the potential threats to validity of the study. Threats to external validity (Section 7.1), internal validity (Section 7.2), construct validity (Section 7.3) and finally conclusion validity (Section 7.4) are discussed in the following.

7.1. External validity

The external validity is concerned with the ability and limitations to generalize the results of the conducted study. To gather our first evidence, sample apps for each development approach were created. Throughout the implementation phase of these apps, the documentation and best practices for the respective development approaches were closely followed. In addition, the material design guidelines introduced by Google were consulted to produce a typical and representative sample implementation. Also, the interaction scenarios were created by including common interactions found in mobile apps. Swiping gestures, form filling and screen transitions can be found in every typical mobile app. Furthermore, for each development approach, three real-world apps were selected by defined criteria. All selected apps have a certain popularity and also have real users, thus the threat towards the bias of lab-only apps was mitigated. To have representative devices, two devices were carefully selected. It was important to select modern smartphones, which represent current technologies, thus only device which support Android 10 and above were used. Also, by using a lower-end and a higher-end device class, we made sure to have representatives of the population of modern smartphones. A limitation of the device selection is that only devices of a single manufacturer were selected. However, Samsung has the largest market share of all Android device manufacturers and we assured that our replication package is usable with Android devices from other manufacturers.

7.2. Internal validity

This threat is related to designing and executing the experiment in order to properly elicit the effect of the treatment (mobile development approaches) on the outcome (energy consumption).

There are several confounding factors which could influence the reliability of the conducted energy measurements in our experiments. We identified factors which could influence the energy measurement results, such as the distance to the Wi-Fi router, the brightness level of the screen, active sensors, cellular network connections and also the effect of other apps and services running in the background of the device. To mitigate those potential threats to validity, we aimed to create a minimal, easy-to-use and easy-to-replicate experimental setting. All sensors and services of the device, which were not in use, were deactivated. In addition, we selected a software-based approach for metering energy consumption, which is even available on every Android device per default. After a thorough investigation of the inner workings of the `batterystats` service and the related energy models [59], we concluded that an isolated energy measurement of only the app under study can be conducted.

Moreover, there are threats which can influence the accuracy of the energy measurements conducted in our experiments. As a software-based approach was used, based on models to estimate the energy consumption, inaccuracies have to be accepted when estimating the absolute energy consumption. We consider the approach adequate for comparing the relative difference between different development approaches, as the measurements are calculated by taking the utilization of the device's hardware components into account. Also, the approach we followed in our measurement approach goes in line with Di Nucci et al. [65], who based their measurement approach on the `batterystats` service as well. Di Nucci et al. [65] evaluated their measurements with a hardware-based ground truth, and the results showed that the relative mean error was less than 5%.

7.3. Construct validity

Threats to the construct validity are concerned with the design of the experiment, and its ability to reflect the construct appropriately. For mitigating the threat of an inadequate explication of the constructs, all artifacts regarding the design of the experiment (i.e., research questions, treatments, variables and hypotheses) were defined before executing any experiments. To mitigate the

threat of a mono-operation bias (i.e., under-representing the construct by using a single independent variable), different user interaction scenarios as well as three real world apps per development approach were used to corroborate the results. Additionally, 30 runs for each experiment were executed to capture different states in which a smartphone might reside.

7.4. Conclusion validity

This kind of threat is concerned with drawing the correct statistical conclusions about the relationship between the treatments (different mobile development approaches) and the outcomes of the executed experiments. The statistical tests were carefully selected and are based on a preliminary analysis of the normality of the samples. Thus, the possibility of misleading results was minimized by selecting non-parametric tests and non-parametric effect size measures. As the samples for PWA energy consumption were used in several hypothesis tests, a conservative Bonferroni correction was applied, which resulted in different alpha levels as shown in Table 4. To mitigate potential threats to conclusion validity, the replication package contains the measured raw data of the experiments and the statistical analysis as a Jupyter notebook. Also, all p-values of the hypothesis tests are listed in the notebook for a more detailed investigation.

8. Conclusion

Developing the same app for multiple platforms is challenging for practitioners [8]. In this study, we compared the energy-efficiency of PWAs with other mobile development approaches, based on interaction scenarios with sample apps and real-world apps. In addition, the energy footprint of different web-browser engines was examined while executing PWAs. In summary, PWAs can be considered as a viable alternative in terms of energy-efficiency to other MCPD approaches. However, the Android Native development approach exhibited the lowest energy consumption in all the conducted experiments. That renders the native development approach still the most energy-efficient.

The study focused on a comparative analysis of the energy footprint of different mobile development approaches. The aim was to provide developers a guideline when facing the difficult decision of choosing a mobile development approach. The results indicate that PWAs are preferable over React Native. The results regarding the MCPD approach Flutter and the hybrid development approach show no inherent technological disadvantage for PWAs. Therefore, we conclude that the application of energy-aware coding practices is important, when developing PWAs.

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.

References

[1] P.K.D. Pramanik, N. Sinhababu, B. Mukherjee, S. Padmanaban, A. Maity, B.K. Upadhyaya, J.B. Holm-Nielsen, P. Choudhury, Power consumption analysis, measurement, management, and issues: A state-of-the-art review of smartphone battery and energy usage, IEEE Access (2019).

[2] S. Hosio, D. Ferreira, J. Goncalves, N. van Berkel, C. Luo, M. Ahmed, H. Flores, V. Kostakos, Monetary assessment of battery life on smartphones, in: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, 2016, pp. 1869–1880.

[3] C. Wilke, S. Richly, S. Götz, C. Piechnick, U. Aßmann, Energy consumption and efficiency in mobile applications: A user feedback study, in: 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, IEEE, 2013.

[4] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, J. Clause, An empirical study of practitioners' perspectives on green software engineering, in: 2016 IEEE/ACM 38th International Conference on Software Engineering, ICSE, IEEE, 2016, pp. 237–248.

[5] G. Pinto, F. Castor, Y.D. Liu, Mining questions about software energy consumption, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 22–31.

[6] 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. (2017).

[7] L. Baresi, W. Griswold, G. Lewis, M. Autili, I. Malavolta, C. Julien, Trends and challenges for software engineering in the mobile domain, IEEE Softw. 38 (1) (2021) 88–96.

[8] M.E. Joorabchi, A. Mesbah, P. Kruchten, Real challenges in mobile app development, in: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, IEEE, Baltimore, MD, USA, 2013, pp. 15–24.

[9] L. Corbalan, J. Fernandez, A. Cuitiño, L. Delia, G. Cáseres, P. Thomas, P. Pesado, Development frameworks for mobile devices: a comparative study about energy consumption, in: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft, ACM/IEEE, Gothenburg, Sweden, 2018.

[10] W. Oliveira, R. Oliveira, F. Castor, A study on the energy consumption of android app development approaches, in: Proceedings of the 14th International Conference on Mining Software Repositories, 2017, pp. 42–52.

[11] K.S. Vallerio, L. Zhong, N.K. Jha, Energy-efficient graphical user interface design, IEEE Trans. Mob. Comput. 5 (7) (2006) 846–859.

[12] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.

[13] S. Huber, L. Demetz, M. Felderer, PWA Vs the others: A comparative study on the UI energy-efficiency of progressive web apps, in: M. Brambilla, R. Chbeir, F. Frasincar, I. Manolescu (Eds.), Web Engineering. ICWE 2021, in: Lecture Notes in Computer Science book series (LNCS, vol. 12706), Springer International Publishing, ISBN: 978-3-030-74296-6, 2021, pp. 464–479, http://dx.doi.org/10.1007/978-3-030-74296-6_35.

[14] S. Xanthopoulos, S. Xinogalos, A comparative analysis of cross-platform development approaches for mobile applications, in: Proceedings of the 6th Balkan Conference in Informatics on - BCI'13, ACM Press, 2013.

[15] T.A. Majchrzak, A.B. rn Hansen, T.-M. Grønli, Comprehensive analysis of innovative cross-platform app development frameworks, in: Proceedings of the 50th Hawaii International Conference on System Sciences, Hawaii International Conference on System Sciences, Hawaii, USA, 2017, pp. 6162–6171.

[16] C.R. Raj, S.B. Tolety, A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach, in: 2012 Annual IEEE India Conference, INDICON, IEEE, 2012, pp. 625–629.

[17] A. Ribeiro, A.R. da Silva, Survey on cross-platforms and languages for mobile apps, in: 2012 Eighth International Conference on the Quality of Information and Communications Technology, Ieee, 2012, pp. 255–260.

[18] A. Biørn-Hansen, T.-M. Grønli, G. Ghinea, A survey and taxonomy of core concepts and research challenges in cross-platform mobile development, ACM Comput. Surv. (2018).

[19] A. Biø rn Hansen, T.A. Majchrzak, T.-M. Grønli, Progressive web apps: The possible web-native unifier for mobile development, in: Proceedings of the 13th International Conference on Web Information Systems and Technologies, WEBIST 2017, SciTePress, Porto, Portugal, 2017.

[20] H. Heitkötter, T.A. Majchrzak, Cross-platform development of business apps with MD2, in: J. vom Brocke, R. Hekkala, S. Ram, M. Rossi (Eds.), Design Science at the Intersection of Physical and Virtual Design, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-642-38827-9, 2013, pp. 405–411.

[21] A. Russell, Progressive web apps: Escaping tabs without losing our soul, 2015, https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/.

[22] T.A. Majchrzak, A. Biørn-Hansen, T.-M. Grønli, Progressive web apps: the definite approach to cross-platform development? in: Proceedings of the 51st Hawaii International Conference on System Sciences, HICSS 2018, IEEE, Hawaii, USA, 2018.

[23] M. Cáceres, K. Christiansen, A. Kostiainen, M. Giuca, A. Gustafson, Web application manifest, 2021, URL: https://www.w3.org/TR/appmanifest.

[24] A. Russell, J. Song, J. Archibald, M. Kruisselbrink, Service workers 1, 2019, URL: https://www.w3.org/TR/service-workers.

[25] M. Firtman, Google play store now open for progressive web apps, 2019, URL: http://bit.ly/3dKYSOp.

[26] Microsoft, Progressive web apps in the microsoft store, 2020, URL: http://bit.ly/3qXRAum.

[27] I. Malavolta, Beyond native apps: Web technologies to the rescue! (keynote), in: Proceedings of the 1st International Workshop on Mobile Development, in: Mobile! 2016, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450346436, 2016.
[28] A. Banerjee, L.K. Chong, S. Chattopadhyay, A. Roychoudhury, Detecting energy bugs and hotspots in mobile apps, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 588–598.
[29] Y. Liu, C. Xu, S.-C. Cheung, J. Lü, Greendroid: Automated diagnosis of energy inefficiency for smartphone applications, IEEE Trans. Softw. Eng. 40 (9) (2014) 911–940.
[30] A. Banerjee, L.K. Chong, C. Ballabriga, A. Roychoudhury, Energypatch: Repairing resource leaks to improve energy-efficiency of android apps, IEEE Trans. Softw. Eng. 44 (5) (2017) 470–490.
[31] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, On the impact of code smells on the energy consumption of mobile applications, Inf. Softw. Technol. 105 (2019) 43–55.
[32] L. Cruz, R. Abreu, Performance-based guidelines for energy efficient mobile applications, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft, IEEE, 2017, pp. 46–57.
[33] L. Cruz, R. Abreu, Catalog of energy patterns for mobile applications, Empir. Softw. Eng. 24 (4) (2019) 2209–2235.
[34] 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, 2016, pp. 225–236.
[35] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, D. Poshyvanyk, Mining energy-greedy api usage patterns in android apps: an empirical study, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 2–11.
[36] J. Huang, F. Qian, A. Gerber, Z.M. Mao, S. Sen, O. Spatscheck, A close examination of performance and power characteristics of 4G LTE networks, in: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, 2012, pp. 225–238.
[37] K. Dutta, D. Vandermeer, Caching to reduce mobile app energy consumption, ACM Trans. Web (TWEB) 12 (1) (2017) 1–30.
[38] D. Li, Y. Lyu, J. Gui, W.G. Halfond, Automated energy optimization of http requests for mobile applications, in: 2016 IEEE/ACM 38th International Conference on Software Engineering, ICSE, IEEE, 2016, pp. 249–260.
[39] L. Chen, L. Wang, D. Zhang, S. Li, G. Pan, Enup: Energy-efficient data uploading for mobile crowd sensing applications, in: 2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress, UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld, IEEE, 2016, pp. 1074–1078.
[40] C.L. Chamas, D. Cordeiro, M.M. Eler, Comparing REST, SOAP, socket and gRPC in computation offloading of mobile applications: An energy cost analysis, in: 2017 IEEE 9th Latin-American Conference on Communications, LATINCOM, IEEE, 2017, pp. 1–6.
[41] 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, Vol. 1, SANER, IEEE, 2016, pp. 529–540.
[42] I. Malavolta, K. Chinnappan, L. Jasmontas, S. Gupta, K.A.K. Soltany, Evaluating the impact of caching on the energy consumption and performance of progressive web apps, in: 7th IEEE/ACM International Conference on Mobile Software Engineering and Systems 2020, 2020.
[43] I. Malavolta, G. Procaccianti, P. Noorland, P. Vukmirović, Assessing the impact of service workers on the energy efficiency of progressive web apps, in: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, IEEE Press, Buenos Aires, Argentina, 2017.
[44] Y. Wen, W. Zhang, H. Luo, Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones, in: 2012 Proceedings Ieee Infocom, IEEE, 2012, pp. 2716–2720.
[45] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, Maui: making smartphones last longer with code offload, in: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, 2010, pp. 49–62.
[46] M. Wan, Y. Jin, D. Li, W.G. Halfond, Detecting display energy hotspots in Android apps, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST, IEEE, 2015, pp. 1–10.
[47] T. Agolli, L. Pollock, J. Clause, Investigating decreasing energy usage in mobile apps via indistinguishable color changes, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft, IEEE, 2017, pp. 30–34.
[48] 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: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C, IEEE, 2017, pp. 11–14.
[49] X. Chen, K.W. Nixon, H. Zhou, Y. Liu, Y. Chen, Fingershadow: An OLED power optimization based on smartphone touch interactions, in: 6th Workshop on Power-Aware Computing and Systems, HotPower 14, 2014.
[50] H.-Y. Lin, P.-C. Hsiu, T.-W. Kuo, ShiftMask: Dynamic OLED power shifting based on visual acuity for interactive mobile applications, in: 2017 IEEE/ACM International Symposium on Low Power Electronics and Design, ISLPED, IEEE, 2017, pp. 1–6.
[51] Y. Liu, C. Xu, S.-C. Cheung, Where has my battery gone? Finding sensor related energy black holes in smartphone applications, in: 2013 IEEE International Conference on Pervasive Computing and Communications, PerCom, IEEE, 2013, pp. 2–10.
[52] T.O. Oshin, S. Poslad, A. Ma, Improving the energy-efficiency of GPS based location sensing smartphone applications, in: 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, 2012, pp. 1698–1705.
[53] S. Taleb, N. Abbas, H. Hajj, Z. Dawy, On sensor selection in mobile devices based on energy, application accuracy, and context metrics, in: 2013 Third International Conference on Communications and Information Technology, ICCIT, IEEE, 2013, pp. 12–16.
[54] M. Ciman, O. Gaggi, An empirical analysis of energy consumption of cross-platform frameworks for mobile development, Pervasive Mob. Comput. (2017).
[55] P. LePage, Webapks on android, 2021, URL: https://developers.google.com/web/fundamentals/integration/webapks.
[56] M. Ali, A. Mesbah, Mining and characterizing hybrid apps, in: Proceedings of the International Workshop on App Market Analytics, 2016, pp. 50–56.
[57] CapacitorJS, Cordova and PhoneGap, 2020, https://capacitorjs.com/docs/cordova.
[58] L. Cruz, R. Abreu, On the energy footprint of mobile testing frameworks, IEEE Trans. Softw. Eng. (2019).
[59] 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. (2015).
[60] S.S. Shapiro, M.B. Wilk, An analysis of variance test for normality (complete samples), Biometrika 52 (3/4) (1965) 591–611.
[61] P.E. McKnight, J. Najab, Mann-whitney U test, in: The Corsini Encyclopedia of Psychology, Wiley Online Library, 2010, p. 1.
[62] C. Bonferroni, Teoria Statistica Delle Classi E Calcolo Delle Probabilita, Vol. 8, Pubblicazioni Del R Istituto Superiore Di Scienze Economiche E Commericiali Di Firenze, 1936, pp. 3–62.
[63] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions, Psychol. Bull. 114 (3) (1993) 494.
[64] C. Rieger, T.A. Majchrzak, Towards the definitive evaluation framework for cross-platform app development approaches, J. Syst. Softw. 153 (2019) 175–199.
[65] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, A. De Lucia, Software-based energy profiling of android apps: Simple, efficient and reliable? in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2017, pp. 103–114.