Analyzing the Resource Usage Overhead of Mobile App Development Frameworks

Wellington Oliveira
woliveira@fc.ul.pt
University of Lisbon
Lisbon, Portugal

Bernardo Moraes
bernardodemsj@gmail.com
Federal University of Pernambuco
Recife, Brazil

Fernando Castor
f.j.castordelimafilho@uu.nl
Utrecht University
Utrecht, The Netherlands
Federal University of Pernambuco
Recife, Brazil

João Paulo Fernandes
jpaulo@fe.up.pt
LIACC & DEI-FEUP
University of Porto
Porto, Portugal

ABSTRACT

Mobile app development frameworks lower the effort to write and deploy apps across different execution platforms, e.g., mobile, web, and stand-alone PCs. At the same time, their use may limit native optimizations and impose overhead, increasing resource usage. In mobile devices, higher resource usage results in faster battery depletion, a significant disadvantage. In this paper, we analyze the resource usage of Android benchmarks and apps based on three mobile app development frameworks, Flutter, React Native, and Ionic, comparing them to functionally equivalent, native variants written in Java. These frameworks, besides being in widespread use, represent three different approaches for developing multiplatform apps: Flutter supports deployment of apps that are compiled and run fully natively, React Native runs interpreted JavaScript code combined with native views for different platforms, and Ionic is based on web apps, which means that it does not depend on platform-specific details. We measure the energy consumption, execution time, and memory usage of ten optimized, CPU-intensive benchmarks, to gauge overhead in a controlled manner, and two applications, to measure their impact when running commonly mobile app functionalities. Our results show that cross-platform and hybrid frameworks can be competitive in CPU-intensive applications. In five of the ten benchmarks, at least one framework-based version exhibits lower energy consumption and execution time than its native counterpart, up to a reduction of 81% in energy and 83% in execution time. Furthermore, in three other benchmarks, framework-based and native versions achieved similar results. Overall, Flutter, usually imposes the least overhead in execution time and energy, while React Native imposes the highest in all the benchmarks. However, in an app that continuously animates multiple images on the screen, without interaction, the React Native version uses the least CPU and energy, up to a reduction of 96% in energy compared to the second-best framework-based version. These findings highlight the importance of analyzing expected application behavior before committing to a specific framework.

1 INTRODUCTION

While the smartphone market started out with multiple platforms competing for space [1], it has become a *de facto* duopoly [2] in which two operating systems have 99.35% of the global market share: Android, with 73.32% and iOS, with 27.03% [3]. To maximize the potential market of the apps they build, companies have to support applications that run on both platforms. It is also necessary for companies to support the web, in case the user does not have access to the application or prefers to use it through a browser.

Mobile applications are usually built using native code for a particular platform. However, the development process for mobile ecosystems varies drastically, with elements such as programming languages, productivity tools, libraries, and frameworks being completely different. For example, Android apps are typically written in Java and Kotlin [4] and iOS developers use Swift or Objective-C [5]. As a consequence of this diversity, developer expertise in writing Android apps does not translate to iOS and vice-versa. If a company decides to use native code, it will then have to maintain at least two versions, one for each platform, seeking to cover most of its potential customers. Furthermore, if the company is interested in providing support to browser-based interaction, it will have another version with the application developed in JavaScript running on top of a web application framework (e.g., Angular, React, Vue).

To reduce development effort, a number of mobile app development frameworks models were created [6], such as *hybrid* and *cross-platform app development frameworks*. Hybrid app development frameworks such as Ionic, PhoneGap and Monaca, render web components into the UI. Web technologies, albeit familiar for many developers, can suffer from lack of responsiveness. To address this issue, in recent years cross-platform app development frameworks[1] have grown in popularity. One of the philosophies behind cross-platform frameworks is that the code generated by the development platform must be native to each of the different operating systems. These frameworks have been receiving great acceptance from the market and developers [7]. React Native, Flutter, Xamarin, and NativeScript are examples of widely adopted cross-platform app development frameworks.

Another popular scenario can be observed in the case of *low-code* platforms, where the developer makes intensive use of abstract visual models to specify their applications. In order to generate code for different operating systems, these platforms can, as part of their

[1] Encompassing both *interpreted* and *cross compiled* definitions [6].

infrastructure, make use of hybrid or cross-platform development tools. Such is the case of OutSystems[2], which uses Cordova to access native APIs in mobile devices [8].

In this work, we analyze mobile app development frameworks under the perspective of the potential resource overhead introduced in the apps generated by them, aiming to assist potential developers in choosing the tool that best suits their needs. One of these resources, energy, is considered essential by both users and developers [9]. Previous work [10] has shown that building mobile apps that are energy-efficient is difficult and different smartphone models, usage profiles, and implementation choices can have significant impact on their energy consumption. Mobile app development frameworks add an extra layer of complexity, since they include sophisticated infrastructures and use diverse programming languages: i) JavaScript for React Native, NativeScript and Ionic, ii) Dart for Flutter, and iii) C# for Xamarin. As reported in previous works, programming languages can have a major impact on energy consumption in mobile devices [11–13].

We analyze the resource impact of two of the most popular cross-platform app development frameworks, React Native and Flutter, and one hybrid app development framework, Ionic [14]. These frameworks, besides being in widespread use, represent three different approaches for developing multiplatform apps: Flutter supports deployment of apps that are compiled and run fully natively, React Native runs interpreted JavaScript code combined with native views for different platforms, and Ionic is based on web views, which means that it does not depend on platform-specific details.

Considering the growing popularity of cross-platform frameworks and the perspective of new applications being made using one of the selected frameworks, knowledge on how they handle resource management can be decisive [15]. This paper tries to shed some light on the following research question:

RQ: What is the energy, CPU, and memory usage overhead imposed by mobile app development frameworks?

To answer this question, we conducted a series of experiments to analyze the difference in resource usage imposed by the frameworks. At first we executed benchmarks to exercise particular aspects of these frameworks. These benchmarks were collected from the *Computer Language Benchmark Game* (CLBG) [16], a website which focuses on comparing the performance of programming languages that has been used in multiple studies [10, 17, 18]. Because benchmarks may not be representative of a mobile application [19], in particular with regards frequently used app functionalities, two simple apps were used to exercise these aspects: ContactApp [20], a contact manager app; and RotatingApp, an app that we developed and that shows and animates a number of images on the screen.

Our results show that cross-platform and hybrid frameworks can be competitive in CPU-intensive applications. In 60% of the scenarios we analyzed targeting benchmarks, framework-based versions consumed less energy than native versions implemented in Java. Considering execution time, half the scenarios had framework-based versions as winners. In the best bases, we observed reductions of 81% in energy and 83% in execution time. This has practical impact for developers. CPU-intensive apps can be made multiplatform without paying a high performance cost.

Among the multiplatform frameworks, Flutter, which compiles all the application code to native, performed the best on benchmarks (both CPU and IO intensive), which could translate to it working well with applications that are compute-intensive. It also performed well in experiments with an interactive app. React Native performed poorly on almost every benchmark scenario but showed good performance in an app that rotates a large number of images simultaneously. This positive result stems from the use of preexisting native library that reduces the need for JavaScript code to interact with the native UI elements of an application. Considering the importance of applications that manipulate videos and images, e.g., Instagram and Tiktok, a solution that combines interpreted JavaScript code with native libraries for tasks such as animation can be a sound trade-off between ease of development and performance in this scenario. At the same time, in this scenario, a purely web-based solution such as Ionic exhibits poor performance, since it cannot depend on native elements.

2 METHODOLOGY

Our analysis was conducted on some of the most popular platforms used by developers [7], namely: React Native, Flutter, and Ionic. The used versions of these frameworks were: React Native v0.68.2, Flutter v3.0.3, and Ionic v6.2.7 (running on top of Capacitor v4.2.0). As a comparison baseline, we also executed the experiments on the Android SDK. All applications were created using the default options on each framework and native.

Being multi-platform development frameworks, React Native and Flutter can be used to compile applications for Web, Linux, Windows, or MacOS [21–23]. No resource usage analysis was made for these platforms. Xamarin was not included in this work because it was deprecated in 2020 in favor of .NET Multi-platform App UI (.NET MAUI) [24]. Because this framework is still in the first major release (released in May 2022) and could significantly change in future versions, we did not include it in this work.

Across the different scenarios, the benchmarks and applications we use must be implemented in at least three programming languages: Java or Kotlin for Android SDK, Dart for Flutter, and JavaScript for React Native and Ionic. We opted to use Java as the programming language for the native solutions because it is still the most widely used language for developing Android apps [25]. The three frameworks we consider are representatives of different technological approaches to build multi-platform apps. Flutter compiles directly to native code, whereas Ionic runs on top of a web view. React Native uses native UI elements and a JavaScript thread to run code. The two interact by means of React Native bridging.

We perform a set of mixed tests with benchmarks, exercising specific parts of the system (e.g., memory consumption, CPU usage and disk access), simulating the different scenarios of a potential application by a user. We used all ten benchmarks that are available on CLBG: binary-trees, fannkuch-redux, fasta, k-nucleotide, n-body, mandelbrot, pidigits, regex-redux, reverse-complement, and spectral-norm. The inputs for these benchmarks were selected individually, taking into consideration the hardware limitations.

[2]Outsystems has been repeatedly named by Gartner® as a Leader in the Magic Quadrant™ for Enterprise Low-Code Application Platforms.

Table 1: Benchmarks, apps, and input sizes we considered.

BENCHMARKS

benchmark	input[↓]	input[↑]
binarytree	21	22
fannkuch	11	12
fasta	1.75E+07	5.25E+07
knucleotide	2.50E+06	5.00E+06
nbody	2.25E+07	4.05E+08
mandelbrot	7000	32000
pidigits	13000	19500
regex-dna	2.50E+06	1.00E+07
revcomp	5.00E+06	4.00E+07
spectral	13000	19500

ROTATING APP

#images / #columns	28 / 4 252 / 12	112 / 8 448 / 16

CONTACT APP

functionalities	Switch Screen Use Drawer Scroll Up	Insert DB Scroll Down

That means that a benchmark b on a device d will run the same input i for each app development framework. Table 1 summarizes our experimental space.

Table 1 summarizes the benchmarks, apps, and inputs provided to them. The sizes of the inputs were selected in pairs for each framework. Input values were selected to keep the execution time for each benchmark at a minimum of 500 milliseconds, to reduce the measurement fluctuation [12]. At the same time, this is low enough to allow us to measure thousands of executions. Furthermore, we employed two additional criteria to select input sizes: (a) the largest value that every framework could run (b) the largest value for which at least two frameworks could finish the execution. When referring to each input, we use the symbols [↓] for the former and [↑] for the latter. For example, for fannkuch, inputs 11 and 12 are identified as fannkuch[↓] and fannkuch[↑], respectively. Both the benchmarks and apps were installed using the release mode. No optimizations or modifications on the source code were made in any cases to reduce the application size.

Each benchmark was executed 45 times; the first 15 were discarded as warm-up to reduce bias [26]. The minimum execution time of 500-milliseconds was established because it was enough to execute every single native benchmark that did not result in `OutOfMemoryError` and to limit the time it would take to collect the data. In some cases, a benchmark took several minutes to finish on React Native while taking a couple seconds in the other implementations. Because of the sheer number of benchmark executions in our study (2835 valid benchmark executions), establishing a minimum was necessary to reduce the overall measurement time. We employed a maximum threshold of 120s for single benchmark execution. Most successful benchmark runs require only a few seconds. The 120s threshold is sufficient for every benchmark on native implementation to end its execution, when an `OutOfMemoryError` does not happen first. For the benchmarks pidigits, revcomp, and spectral, input sizes that executed within the 120s threshold on React Native execute too fast to have a reliable measurement in the other implementations. The React Native version was two orders of magnitude slower than the other versions. For this reason, the React Native versions of these benchmarks were not analyzed.

In CLBG, every benchmark has multiple versions. For example, there are five versions of nbody written in Java. Some JavaScript benchmark implementations from CLBG use explicit threading through the library worker_threads [27]. However, this feature is not supported by React Native and Ionic. Thus, for fairness, we opted to use the fastest single-threaded version for each benchmark. For the benchmarks that only had parallel implementations, Java regex and Dart knucleotide, that version was used.

The RotatingApp, created for this work, has two functions: i) load a number of images on the screen. ii) rotate the images 9 times clockwise for 18s. The images were collected randomly[3]; to avoid the overhead of downloading them while running experiments, all images were deployed with the application. All images have the same size (100x100 pixels) and use the same format (JPG). This experiment was further discriminated, with different numbers of images that the application should show on the screen. To achieve that, we used an aspect ratio of 4:7 and four columns sizes: 4, 8, 12 and 16, resulting, respectively, in 28, 112, 252, and 448 images on screen. Figure 1 shows the application running with 28 images.

The ContactApp was developed by Huber et al. [28] as an application that manages a list of contacts. It has five functions: add a new contact; scroll over the contact list (up and down); open the menu drawer; and change to a different screen. Android UI Automator was used for executing the app. A single execution consisted of five steps: (i) opening and closing the drawer menu five times; (ii) and (iii) scroll down (and then up) the list of contacts using five swipes; (iv) switch to the AddContact screen; (v) type the contact information and add it. The data was collected after each step. This methodology has been previously validated [28].

Although the apps presented in this work are simple, they cover some of the most commonly uses of an Android app, such as: swapping between different screens; smooth scrolling through a list; adding and consulting elements to a local database; and image processing and animation.

Analyzing the overhead of different frameworks imposes a significant challenge. If one uses real-world apps, it is nigh impossible to directly compare them since they have different combinations of functionalities and implementation approaches. To have a fairer comparison, we use simpler apps that include functionality that is available in real apps but enable us to perform comparisons in a more direct manner, controlling for external factors (such as additional functionality running on parallel that is not involved in the comparison). Previous work has either used real-world apps and performed comparisons disregarding common functionalities [20] or employed simpler apps that sacrifice some external validity but make it possible to more precisely pinpoint the effect of the analyzed treatments [28–32].

During our experiments, we used a Samsung Galaxy S20FE (SM-G780G). It is composed of a Snapdragon 865, octa-core (1x2.84 GHz Cortex-A77, 3x2.42 GHz Cortex-A77, 4x1.80 GHz Cortex-A55), 6GB RAM, and a 4500mAH battery running on Android 12.

To collect the energy consumption, execution time, CPU time, we use batterystats and to collect the memory data, we use meminfo. Both are available through the Android Debug Bridge (adb). The usual way to execute the adb commands is connecting the

[3]We have used images taken from Lorem Picsum, https://picsum.photos/

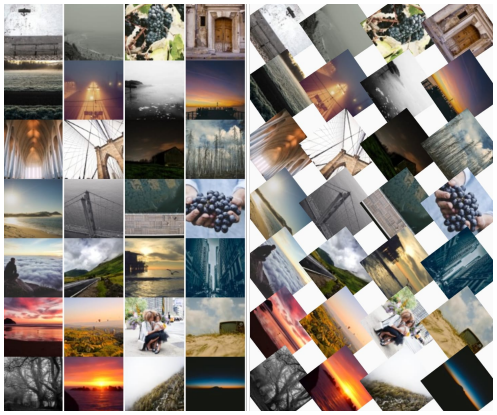

Figure 1: Two images of the RotatingApp. The image on the left-hand side shows the starting position. The one on the right-hand side shows the rotated images.

device to a computer via an USB cable. Because in our case it was necessary to measure the energy consumed during the execution, we connected it via Wi-fi, using the ADB commands TCPIP and CONNECT. All executions and data collecting were done automatically via script. No test suite was used. All input data was sent directly to the applications. To ensure data from a single execution, the app process was killed after each execution via ADB and the command BATTERYSTATS --RESET was executed to clean the power consumption data. Furthermore, the BATTERYSTATS power consumption is collected using power profiles and models [33], therefore there is no risk of data misalignment [34].

To reduce the risk of any bias in the measurement process, we never let the battery charge go below 50%, executed only the application that was being analyzed, kept the brightness at 25% and activated airplane mode (with the Wi-fi turned on). The methodology used to measure and collect the energy data is in line with previous works on the area of mobile energy consumption [10, 20].

3 STUDY RESULTS

This section is divided in three parts: Section 3.1 reports the benchmark results; Section 3.2, results from RotatingApp; and Section 3.3, results from ContactApp. This section focuses on presenting the obtained results of the study. Section 4 presents potential explanations for some of the more interesting results.

As the standard deviation was very small in most cases (less than 5% of relative standard deviation), all graphs will be presented using mean values, which will be shown using bar charts. We also ran the Wilcoxon-Mann-Whitney test [35] comparing the results between the benchmark versions for each framework and the native versions. Differences were statistically significant ($\rho < 0.01$), with only a few exceptions, on: Ionic knucleotide[↓] and Flutter binarytree[↑] on memory usage, Ionic mandelbrot[↓] on execution time, and Flutter revcomp[↑] on energy consumption. All data presented here can also be found on the companion site [36].

3.1 Benchmarks

We compared the benchmarks on four different aspects: energy consumption, execution time, memory usage and application size. Figure 2 presents a summary of the benchmark results.

Previous experiments suggest that memory usage usually does not have a significant impact on energy consumption [37, 38] but it can be a limiting aspect when running an app. We noticed that there exists a significant difference regarding memory usage across the frameworks. As an example, when running knucleotide, Ionic used 107MB of memory while React Native used more than 2GB, which may be prohibitive for lower-end smartphones. We have also looked into the application size for the same reasons.

For some [↑] benchmark inputs, executions did not finish because they caused an error during execution (usually out of memory error) or it exceeded our timeout limit. This happened two times for Ionic (pidigits[↑] and revcomp[↑]) and Java (fannkuch[↑] and regex[↑]). Moreover, when running React Native, most benchmarks did not finish, either because they took too long or because they failed. Across a total of twenty benchmark-input pairs, only seven correctly executed the four versions, all running the [↓] input: binarytree[↓], fannkuch[↓], fasta[↓], knucleotide[↓], mandelbrot[↓], nbody[↓], and regex[↓]. Flutter did not fail to execute any of the selected inputs.

Execution time. Across the twenty benchmark-input pairs, the Java implementation was faster in ten, Flutter in eight and Ionic in two. Among these three implementations, Java was the slowest in three cases, Flutter in one and Ionic in twelve. React Native was the slowest in every single benchmark it managed to finish executing.

In some cases, using a framework imposed a low overhead in execution time. That was the case for three benchmarks running on Flutter: nbody[↓], (Java: 3.28s, Flutter: 3.72s, 13.4% more), revcomp[↓] (Java: 0.75s, Flutter: 0.85s, 13.3% more), and regex[↓] (Java: 8.38s, Flutter: 8.54s, 2% more). For five other benchmarks (binarytree, fannkuch, mandelbrot, pidigits, and regex), the frameworks were even faster than the Java implementation. On pidigits, Flutter was 5.91x([↓]) and 4.64x([↑]) faster than Java. On mandelbrot, Flutter was 38%([↓]) and 39%([↑]) faster than Java. On regex, Ionic was 2.38x([↓]) faster than Java and finished the execution on regex[↑] while Java could not. On binarytree, Flutter was 56%([↓]) and 2.76x([↑]) faster than Java and Ionic was 31% faster than Java on binarytree[↑]. On fannkuch[↓], Flutter and Ionic were 4.33x and 2.96x faster, respectively. Both frameworks also managed to finish fannkuch[↑] while Java could not. The input selection can impact the performance of a framework. While Ionic was faster than Java running binarytree[↑], Java was 77% faster while running binarytree[↓].

Some benchmarks were considerably slower on a specific framework. Running on Ionic, revcomp[↓] took, on average, 64.99s to finish. The Flutter and Java versions took, on average, 0.85s and 0.75s, respectively. This means that if a developer chooses to use Ionic on this problem, it will take, on average, 76x more time for the same workload when compared to a more efficient framework.

Energy consumption. Java consumed less energy in eight of the twenty benchmark-input pairs, Flutter in ten and Ionic in two. Among these implementations, Java consumed the most in four

Figure 2: Execution time, energy and memory data running the CLBG benchmarks. Each chart has its own individual scale.

cases, Flutter in one and Ionic in eleven. In some cases, the difference was large, e.g., the Java version of pigidits[↑] consumed 3.17x more energy than Flutter's. In others, the results were close, e.g., running nbody[↓] on Flutter imposed a low energy consumption overhead, going from an average 5.27J in Java to 5.61J with Flutter. As with execution time, React Native was always the least energy-efficient.

For the same five benchmarks (binarytree, fannkuch, mandelbrot, pidigits, and regex) where the frameworks were faster, they also consumed less energy. On pidigits, Java consumed 4.17x([↓]) and 5.30x([↑]) more energy than Flutter. On regex[↓], Java consumed 84% and 54% more energy than Ionic and Flutter, respectively. On binarytree, Java consumed 86%([↓]) and 2.00x([↑]) more energy than Flutter. On fannkuch[↓], Java consumed 3.03x and 46% more energy than to Flutter and Ionic, respectively. On mandelbrot, Java consumed 14%([↓]) and 53%([↑]) more energy than Flutter.

The increase in input changed which framework performed the best: Flutter was the most energy efficient framework running knucleotide[↓] and Ionic running knucleotide[↑].

Although spectral executed faster in Java than in any framework, Flutter and Ionic consumed less energy than Java for both inputs. For spectral[↓] the consumption of Flutter, Ionic and Java were 17.05J, 18.74J, and 21.87J, respectively. Meanwhile, for spectral[↑], the consumption was 33.00J, 41.24J, and 48.21J. Being a benchmark that highly benefits from parallelism, we also executed the parallel version of spectral on Java and Flutter, analyzing the execution time and energy consumption while changing the number of threads. More details about this analysis can be found on Section 4.

Memory Usage. Looking at memory usage, Java consumed the least in fifteen cases, Flutter in one, and Ionic in four. Java consumed

the most memory in three cases, Flutter in six, and Ionic in five. React Native had the highest energy usage in all but two cases, fannkuch[↓] and mandelbrot[↓]. In these cases, Ionic consumed more memory than the others.

Among the frameworks, Flutter had a non negligible memory footprint, which is heavily influenced by benchmark and input. The Flutter version of revcomp[↓]consumed 294MB while revcomp[↑] consumed 1.8GB. Nevertheless, Flutter was able to finish the execution with all inputs while Java's memory usage on some benchmarks incurs in problems, such as out of memory errors for regex[↑] or in explicit invocations of the garbage collector in binarytree[↑]. The memory usage of React Native was high across all benchmarks.

Application size. The average sizes of the APKs generated by the frameworks and native solutions were small: Ionic had 5.2MB; Flutter 6.9MB; Java 9.3MB; and React Native 28.0MB.

> The results of the benchmarks show that multi-platform solutions can beat native ones in execution time and energy consumption. Depending on application requirements, this suggests that it is possible to have both: multi-platform apps and high performance on Android. In general, **Flutter** was the most energy efficient and the fastest but it imposes a non-negligible memory overhead. **Ionic** had a single benchmark where it was the best. **React Native** performed poorly on almost every scenario.

3.2 RotatingApp

The RotatingApp loads a number of images on the screen and rotates them 9 times, for a total of 18s. Two delays of 2s were inserted before the images started to rotate and after they finished for a total of 22s of execution. Since this app runs for a fixed amount of time, there is no point in presenting the execution time. Instead, CPU time will be presented and reported. Figure 3 presents the data summary of the executions.

The results point to a significant overhead when using multiplatform frameworks to load and rotate the images, with a noticeable increase in energy consumption, CPU and memory usage over the native solution. React Native consumed the least energy across all frameworks. However, it also used more CPU than Flutter when executing on 252 and 448 images and more than Ionic on 448 images. It also had the largest memory footprint of all frameworks. On the most demanding case (while running 448 images), it used more than twice Flutter's memory.

The results suggest that Flutter does not scale up well with an increasing number of images. While running with 28 images, Flutter consumed only 30% more energy than Java, consuming less energy than the other frameworks (compared to Java, React Native consumed 66% more and Ionic 247% more). However, the energy consumption increase was higher than observed on the remaining frameworks. While running 448 images, Flutter consumed almost the same energy as Ionic, which was 5x more energy than Java and 2.17x more energy than React Native.

Ionic consumed more energy in every scenario, used more CPU in almost all cases with the exception of 448 images, and was only behind React Native in memory usage. This is a coherent result. A framework based on web views such as Ionic is a poor fit for apps that require frequent or intensive screen updates. A framework that compiles to native (Flutter) or that includes native UI components (React Native) can better leverage the underlying hardware.

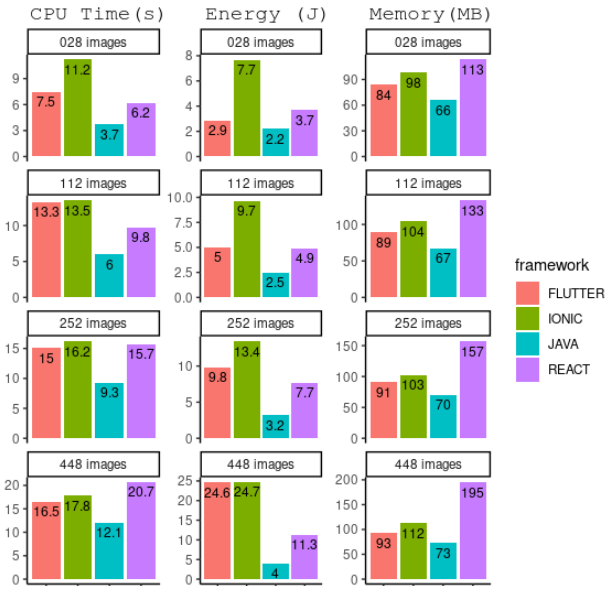

Figure 3: CPU time, energy and memory usage for RotatingApp. Each chart has its own individual scale.

The RotatingApp application size of all frameworks, from smallest to largest: Java 4.6MB; Flutter 7.1MB; Ionic 7.3MB; and React Native 32.3MB.

> Solutions that leverage native elements exhibited consistently lower energy consumption for an app that requires a combination of image processing and intensive screen updating. **React Native** was generally the most energy efficient while using a noteworthy amount of memory. **Flutter** uses less CPU and memory, consuming the least energy when animating a small amount of images. **Ionic** performed poorly in all cases.

3.3 ContactApp

The ContactApp is an application developed to simulate a contact manager app. It has five different functionalities: switch to contact creation screen (SwitchScreens); fill form data to add a new contact in a local database (EnterFormData); open or close the side menu (OpenCloseDrawer); and scroll down or up the list of contacts (ScrollDownList and ScrollUpList, respectively). As with the case of RotatingApp, the operations used during the execution of ContactApp are timed, leading each operation to have the same execution time. Figure 4 presents the data summary.

Java used less CPU and memory and consumed less energy in all but one case: SwitchScreens, when Flutter consumed 27% less energy than Java. React Native consumed more energy, more CPU and more memory than the other solutions. For example, for functionality EnterFormData, it consumed 350% more energy

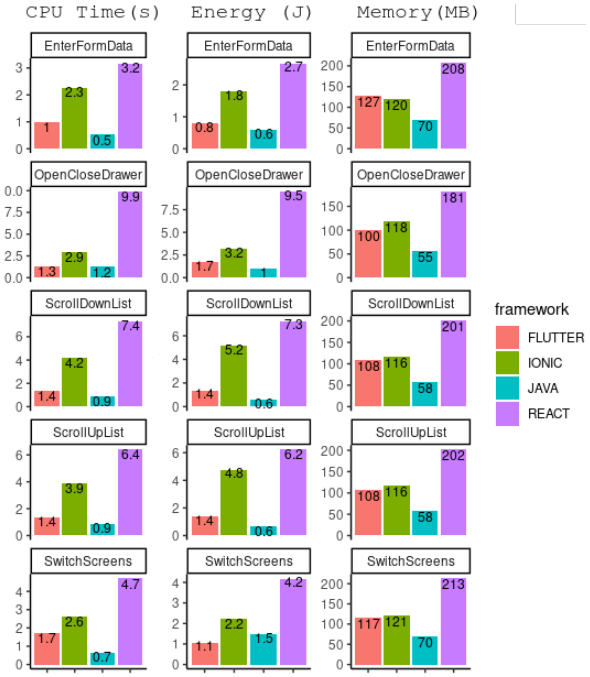

Figure 4: CPU time, energy and memory data for the ContactApp. Each chart has its own individual scale. Each row corresponds to a different functionality of the app.

than Java, 50% more than Ionic and 238% more than Flutter. Between the two remaining options, Flutter and Ionic, the former used the resources more efficiently. It consumed less energy and used less CPU in every scenario. The only point in which Ionic has an edge over Flutter is when executing EnterFormData, where Flutter consumes more memory than Ionic.

Some functions appear to have greater impact on the energy consumption of specific frameworks. The scroll functions appears to be the most energy consuming. While executing ScrollDownList and ScrollUpList, Ionic consumed 8.32x and 7,46x more energy than Java while React Native consumed 11.76x and 9.74x.

React Native running OpenCloseDrawer also consumed more energy than the average: 9.71x more energy than Java. This was not the case for Ionic, in which this functionality consumed just 3.34x more energy. The remaining functionalities did not consume as much energy. EnterFormData, and SwitchScreens consumed 3x and 1.53x more energy on Ionic and 2.86x and 4.43x more energy on React Native. Flutter did not appear to have any outlier consuming function.

The ContactApp application size of all frameworks, from smallest to largest are the following: Java 1.2MB; Ionic 3.1MB; Flutter 18.4MB; and React Native 30.0MB.

> **Flutter** imposed the least overhead over the native solution and was able to save some energy for one functionality. **Ionic** and **React Native** imposed a significant overhead with the latter having the worst results for every scenario.

Table 2: Expected overhead when using mobile app development frameworks. Each cell present the *mean*. Animation also present [*min*, *max*].

Function	FW	CPU (%)	Energy (%)	Memory (%)
Animation	Flutter	**68** [36, 101]	**254** [30, 515]	**29** [26, 31]
	Ionic	**88** [47, 201]	**364** [248, 517]	**51** [46, 54]
	React	**68** [61, 71]	**130** [68, 182]	**116** [70, 166]
Scroll Up/Down	Flutter	**118**	**52**	**85**
	Ionic	**691**	**348**	**100**
	React	**975**	**661**	**246**
Drawer Use	Flutter	**74**	**4**	**80**
	Ionic	**233**	**138**	**113**
	React	**871**	**696**	**228**
Switch Screen	Flutter	**-27**	**156**	**68**
	Ionic	**53**	**296**	**73**
	React	**186**	**612**	**205**
Insert Database	Flutter	**36**	**85**	**81**
	Ionic	**201**	**323**	**72**
	React	**343**	**488**	**197**

4 DISCUSSION

This section analyzes the results presented in the previous section and discusses their implications.

There is no free lunch. Our results show that using an alternative solution to Java is not free. Although each mobile development framework imposes a different overhead over the native solution, there is a clear overhead to use them. Among the different scenarios presented in this paper there was always a difference in execution time, energy consumption, CPU usage and memory allocated.

We explored two profiles of apps with UI elements: apps that make heavy usage of images and animations, with intense screen updating; and, apps that do not execute heavy computation, are less demanding in terms of screen updates, but have a lot of user interaction. As a guide for app developers, we present the expected impact of each framework on each resource, taking into account our results, in Table 2. It's worth noting that these are approximations and some frameworks behaved better on an adequate scenario (e.g., Flutter handles a small number of images well but it becomes increasingly more costly when handling more images).

Parallelism can improve performance, and maybe energy. Parallelism can greatly impact the performance of applications that make intensive use of CPU. Each of the analyzed frameworks has a specific course of action when dealing with parallelism. As an example, React Native is sequential in nature. While the rendering is executed in multiple threads, the JS thread execution is sequential [39]. Moreover, without the usage of 3rd party libraries, there is no easy-to-use approach for parallelism in React Native.

The parallel version of spectral makes intense use of parallelism; it instantiates as many threads as the max number of available cores. For our experimental device (SM-G780G), that means eight threads to run the code. Previous work has shown that, in scenarios involving parallelism, the relationship between performance and energy becomes less clear [17, 40].

To investigate this matter, we conducted an additional experiment. We executed the benchmark spectral[↑], using the same rules

presented in Section 2 but changing the number of threads, going from one to eight in the two approaches that natively supported parallelism: Java and Flutter. Figure 5 summarizes our results and presents the energy savings and the speedup. Energy saving and speedup were calculated using the following formulae, respectively: (i) $1 - (e_n/e_1)$ and (ii) t_1/t_n, where e_n is the energy consumed and t_n the execution time by n number of threads.

The maximum speedup occurred when running with four threads for Java and eight threads for Flutter. However, Flutter version using four threads had a similar gain in time execution and better energy efficiency. The maximum energy efficiency for Java was achieved using the eight available threads but two threads in Flutter. For Flutter, the maximum energy-delay-product [41] is achieved running it with four threads. Further increasing the number of threads lead to a degradation of energy efficiency. The native results are aligned with previous results analyzing the energy efficiency of modern multiprocessors [42].

The results suggest that a larger number of threads mostly likely reduces execution time but may not reduce energy consumption.

Native libraries make up for React Native's overhead, for animations. Overall, React Native had the worst results among the frameworks. For almost all benchmarks and every functionality on ContactApp, it consumed more energy, more memory, more time, and/or used more CPU. However, for RotatingApp, it consumed less energy in all but one case, and used less CPU in half the scenarios. This seemingly conflicting result led us to investigate deeper how React Native executed RotatingApp.

The standard way to use animations on React Native is to use the Animated library [43]. Using this library it is then possible to delegate the animation functionality to the native UI, which is done through the useNativeDriver setting. The documentation explains that using it, the developer may give all the control over the animation to Java: *"(...) everything about the animation (is sent) to native before starting the animation, allowing native code to perform the animation on the UI thread without having to go through the bridge on every frame."* [44]. This optimization avoids overusing the JavaScript bridge and increases performance. Trying to follow the most idiomatic implementation, our React Native implementation of RotatingApp made use of useNativeDriver.

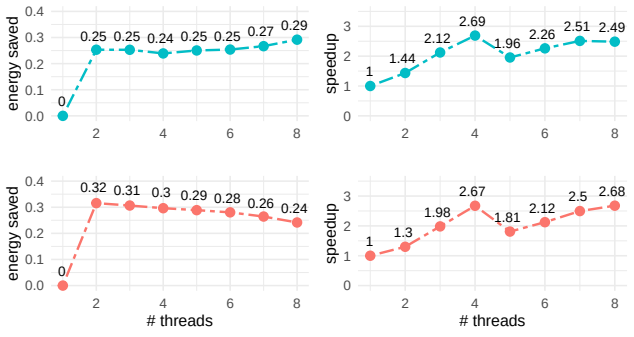

Figure 5: Energy saved and speedup of parallel spectral. Upper graphs from Java and lower graphs from Flutter.

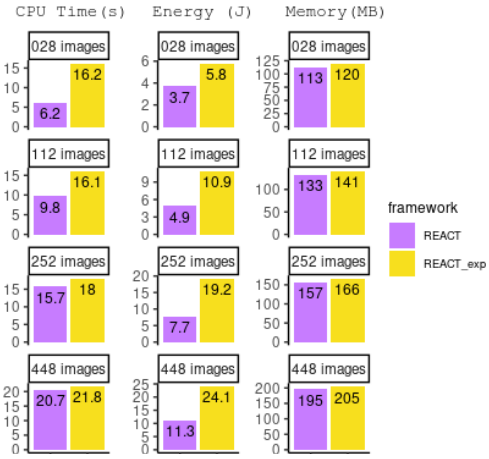

Figure 6: React Native RotatingApp with and without useNativeDriver (REACT and REACT_exp, respectively)

To analyze the impact of using the native driver on RotatingApp, we executed it one more time with the native driver disabled. Figure 6 compares the results of this execution with the previous ones. Although only a small increase in memory is observable, CPU usage and, specially, energy had increased significantly. In the worst case scenario, not using the native driver increases the energy consumption by 2.49 times (i.e., running 252 images). Compared to the other frameworks (Figure 3), React Native would have consumed more energy than Flutter and Ionic in all cases but running 448 images.

Among the analyzed frameworks, this type of optimization only makes sense for React Native. Because it makes use of native UI elements, it can optimize the whole animation, using the native driver to run it. None of the other frameworks can utilize this kind of resource. One can see that in the case of ContactApp, in which there are almost no animations and the entire app is executed on React Native, with intense use of the bridge between JavaScript and the native UI components. As a result, the performance is worse than the other frameworks. Nevertheless, this illustrates a case in which using React Native has an advantage over the other frameworks. In instances in which it is possible to leverage native code through libraries, React Native energy and CPU overhead may be smaller than the alternative frameworks, even ones such as Flutter (with Dart), which produce compiled native code.

Implications. Using a cross-platform or hybrid framework will impose an overhead on resource usage. Using the native language is, in most cases, the best option when trying to optimize resource usage. However, the advantages of developing apps using a cross-platform or hybrid framework (e.g., portability, single repository, programmer expertise, among others) may present a positive trade-off. In that case, each framework has scenarios in which they excel.

Ionic being a hybrid framework and running on top of a WebView makes it easier for web-developers to start their applications but imposes a significant overhead on resource usage, in particular when running animations and scrolling. To minimize the impact in this setting, developers should reduce the number of animations.

React Native is an interpreted framework that makes use of a bridge to communicate with the native environment. In this kind

of framework, the bridge usage may impose a high overhead, since cross-language invocations tend to be expensive in Android [12]. In our experiments, it performed poorly on almost all cases, presenting the biggest memory and application size footprint. The exception was the case where it could leverage the native animations using useNativeDriver. In this case, the usage of bridging was reduced and the animation itself could be performed entirely natively. This, imposed the least amount of overhead across all frameworks.

In case the application is CPU intensive, a cross compiled framework such as Flutter seems to be the best option. It's performance and energy consumption is on par with the native solution. However, it may impose a non-negligible overhead on memory usage. Nevertheless, it seems to be the best alternative for optimizing resources on a mobile device, while reaping the benefits of multiplatform applications.

5 RELATED WORK

Green computing has received significant attention in the last years which resulted in different aspects of energy consumption being analyzed. These studies cover a plethora of different elements, from energy smells [45–47], user interface elements [48], data structures [49], byte-code energy optimization [50] and specific guidelines [51]. Some works have compared the energy consumption of different programming languages [18, 52], but there's still a gap in the knowledge about the resources used by applications created using app development frameworks.

Execution time have been analyzed through different angles over the years. When comparing different programming languages, previous works have looked into how much time they take to execute in sequential [53] and parallel [54] environments. To reduce bias, they have also used code made by expert developers available on sites such as Rosetta Code [18, 55] and the CLBG [18].

Corbalan et at. [29] analyzed the differences in energy consumption of some alternative ways to create Android applications. Three simple apps were implemented to analyze different aspects: a CPU intensive app, a video playback, and an audio playback. They implemented Android apps using Cordova, Titanium, Android NDK, NativeScript, Xamarin, Android SDK and Corona.

Ciman et al. [56] has analyzed the resource management of web apps, hybrid apps (using PhoneGap), interpreted apps (using Titanium) and cross-compiled apps (using MoSync). Their experiments are divided by usage of hardware components. That is, how much energy an application, developed using one of the frameworks, will consume while using a specific sensor.

Hansen et al. [30] have analyzed the CPU and memory overhead of different alternatives to native development: Ionic, Flutter, NativeScript, React Native and MAML / MD2. They implemented benchmarks that exercises some important aspects of a mobile phone: the accelerometer; creating a contact in the smartphone contact list; reading stored files; and geolocation usage.

Dorfer et al. [32] implemented a restaurant finder app and analyzed the CPU, memory and battery overhead of React Native. They focused on the map component, location tracking, and networking.

The previously mentioned works are similar in context to our own, but we have adopted different methodology, measurement tools, frameworks, benchmarks, and resources measured. Even with these differences, the results in the present paper are in line with previous knowledge about mobile app development frameworks.

It is worth noting that recently, cross-platform frameworks have matured, and in the process, gained support of developers, being among the most loved technologies to work with [7]. However, there is still a gap in knowledge on how the apps created by these frameworks manage their resources [15].

More recently, Huber et al. [20] compared the energy consumed by progressive web apps (PWA) in different browsers while running five versions of self-made applications. These same apps were replicated on app development frameworks (Capacitor, React Native, and Flutter). They found that PWA had increased the energy consumption over the native implementation. However, PWA energy overhead was smaller than other solutions, such as React Native.

The present work distinguishes itself from [20] in that: (i) optimized benchmarks were used to analyze the resource management metrics (ii) we analyze different resources not only energy; (iii) we focus the analyzes on the framework overhead over native. (iv) we further investigate some of the causes of perceived overhead. These differences are fundamental when trying to guide the developer choice, as they may impact the long term viability of an application.

6 THREATS TO VALIDITY

During our experiments with images, we used a single fixed size for all images of 100x100 pixels. Our early experiments used different sizes, increasing by powers of two, starting with 2^0 (250x250), up to 2^4 (4000x4000). The last two cases, 2^3 and 2^4, caused out of memory errors. The results with the remaining options did not have a significant impact in resource usage. Nevertheless, using different image quality and formats could affect frameworks resource management.

The results presented in this paper are from the execution on an Android device SM-G780G. Previous works analyzing the energy consumption of mobile application across different multiplatform app development [20, 28, 56] or programming language [12, 25] using multiple devices have shown that even if the absolute values of the experiments change across devices, the relation between the analyzed frameworks or languages stay mostly the same. Nevertheless, we executed our experiments in a second device (SM-G781B) and found similar results to the ones presented throughout this paper. The results from this second device can be found on the companion site [36]. Notwithstanding, our results may change if the experiments were to be executed in different devices.

This study uses three applications types to run the experiments. Among these, ContactApp was implemented by Huber et al [20] and the CLBG benchmarks were implemented by experts and are publicly available [16]. The RotatingApp were implemented by the authors. To reduce the impact of developer expertise bias, we followed the developed patterns of each framework as indicated on their website. To further validate the apps, Google Development Experts [57] (Android and Flutter) and senior developers (Ionic and React Native) were consulted.

The JavaScript Interface (JSI) API was not used for React Native. JSI eliminates the need of the bridge to make the communication between the JavaScript code and the native code, increasing the performance of the app. The results presented in this work reflect the present state of React Native and the apps developed using it.

In this paper we used Android's BATTERYSTATS to collect the energy consumption of the devices. This tool has been used in other studies analyzing energy consumption on mobile devices [10, 20, 58, 59]. DiNucci et al. [60] has also compared these tools to physical equipment to measure energy consumption and has not found statistical difference at 95% confidence.

7 CONCLUSION

Nowadays, there are basically two operating systems when developing for mobile: Android and iOS. Cross-platform and hybrid frameworks have as their main objective reduce the work duplication of creating two applications that have exactly the same function, one for each mobile OS. However, using one of these frameworks could have an impact on the application resource usage and developers have no way to guess what these impacts could be. In this paper, we try to shed some light on the energy, CPU, execution time, and memory overhead imposed by three of the most popular mobile app development frameworks, namely Flutter, React Native and Ionic.

Our results show there is an overhead associated with using each one of these frameworks. However, the size of this overhead depends on the type of the application and on the expected workload. Cross compiled frameworks, such as Flutter, seems to be the best bet in most cases, being on par with native solutions for CPU intensive tasks. However it may use a significant amount of memory; interpreted frameworks, such as React Native, may leverage native animations efficiently but, overall, presented the worst resource management; and hybrid frameworks, such as Ionic, had the worst running animations and presented a mediocre performance on most scenarios. As *future work*, we plan to increase the application diversity and include Microsoft MAUI as another framework option and further investigate the impact of parallelism aspects on mobile app development frameworks.

REFERENCES

[1] TechCrunch+. Everything you need to know about the fragmented mobile developer ecosystem. Techcrunch+. https://techcrunch.com/2010/07/05/mobile-developer-economics-2010/, Accessed 2022-10-18, July 2010.

[2] Natasha Lomas. Uk's antitrust watchdog finally eyes action on apple, google mobile duopoly. Techcrunch+. https://techcrunch.com/2022/06/10/apple-google-mobile-duopoly-cma-market-study, Accessed: 2022-10-18, June 2022.

[3] Mobile operating system market share. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: 2023-01-08.

[4] Native android development languages. https://developer.android.com/guide/components/fundamentals. Accessed: 2023-01-08.

[5] ios development languages. https://www.swift.org/blog/swift-linux-port/. Accessed: 2023-01-08.

[6] C.P R. Raj and S. B. Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *INDICON*, 2012.

[7] Stack overflow developer survey. https://survey.stackoverflow.co/2022/ Accessed: 2023-01-15, 2022.

[8] Outsystems. https://www.outsystems.com/evaluation-guide/what-native-capabilities-does-outsystems-support/. Accessed: 2022-10-18.

[9] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause. An empirical study of practitioners' perspectives on green software engineering. In *ICSE*, 2016.

[10] W. Oliveira, R. Oliveira, F. Castor, G. Pinto, and J. P. Fernandes. Improving energy-efficiency by recommending java collections. *EMSE*, 26(3):55, Apr 2021.

[11] X. Chen and Z. Zong. Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation. In *2016 BDCloud-SocialCom-SustainCom*, pages 485–492, 2016.

[12] W. Oliveira, R. Oliveira, and F. Castor. A Study on the Energy Consumption of Android App Development Approaches. In *MSR*, 2017.

[13] Z. Kholmatova. Impact of programming languages on energy consumption for mobile devices. In *ESEC/FSE*, page 1693–1695. ACM, 2020.

[14] Kotlin. The six best cross-platform app development frameworks. https://kotlinlang.org/docs/cross-platform-frameworks.html, Accessed 2022-10-18, September 2022.

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

[16] Computer language benchmark game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Accessed: 2022-10-18.

[17] L. G. Lima, F. S., P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. On haskell and energy efficiency. *Journal of Systems and Software*, 149:554–580, 2019.

[18] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[19] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In *WebApps*, 2010.

[20] Stefan Huber, Lukas Demetz, and Michael Felderer. A comparative study on the energy consumption of progressive web apps. *ISJ*, 108:102017, 2022.

[21] Flutter multiplatform. https://flutter.dev/multi-platform. Accessed: 2023-01-08.

[22] React native windows and macos. https://microsoft.github.io/react-native-windows/. Accessed: 2023-01-08.

[23] React native web. https://github.com/necolas/react-native-web/. Accessed: 2023-01-08.

[24] David Ortinau. The new .net multi-platform app ui. Microsoft. https://devblogs.microsoft.com/xamarin/the-new-net-multi-platform-app-ui-maui/, Accessed 2022-10-18, February 2021.

[25] M. Peters, G.n . Scoccia, and I. Malavolta. How does migrating to kotlin impact the run-time efficiency of android apps? In *SCAM*, pages 36–46, 2021.

[26] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017.

[27] Node.js worker threads. https://nodejs.org/api/worker_threads.html. Accessed: 2022-12-30.

[28] S. Huber, L. Demetz, and M. Felderer. Pwa vs the others: A comparative study on the ui energy-efficiency of progressive web apps. In *Web Engineering*, 2021.

[29] L. Corbalan, J. Fernandez, A. Cuitiño, L. Delia, G. Cáseres, P. Thomas, and P. Pesado. Development frameworks for mobile devices: A comparative study about energy consumption. In *MOBILESoft*, page 191–201, 2018.

[30] A. Biørn-Hansen, C. Rieger, T. Grønli, T. Majchrzak, and G. Ghinea. An empirical investigation of performance overhead in cross-platform mobile development frameworks. *Empirical Software Engineering*, 07 2020.

[31] S. Huber. and L. Demetz. Performance analysis of mobile cross-platform development approaches based on typical ui interactions. In *ICSOFT*. INSTICC, 2019.

[32] T. Dorfer, L. Demetz, and S. Huber. Impact of mobile cross-platform development on cpu, memory and battery of mobile devices when using common mobile app features. *Procedia Computer Science*, 175:189–196, 2020.

[33] M. A. Hoque, M. Siekkinen, Y. Khan, K. N.and Xiao, and S. Tarkoma. Modeling, profiling, and debugging the energy consumption of mobile devices. *ACM Comput. Surv.*, 48(3), 2015.

[34] R. Saborido, V. Arnaoudova, G. Beltrame, F. Khomh, and G. Antoniol. On the impact of sampling frequency on software energy measurements. Technical report, PeerJ PrePrints, 2015.

[35] Daniel S. Wilks. *Statistical methods in the atmospheric sciences*. Elsevier Academic Press, Amsterdam; Boston, 2011.

[36] Anonymous. Companion website. https://github.com/frameworkresource/frameworkresource,Accessed 2022-10-18.

[37] G. Pinto, K. Liu, F. Castor, and Y. D. Liu. A comprehensive study on the energy efficiency of java thread-safe collections. In *ICSME*, 2016.

[38] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. Spelling out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software*, 161:110463, 2020.

[39] React Native. Thread model. https://reactnative.dev/architecture/threading-model, Accessed 2022-10-18, October 2022.

[40] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA*, 2014.

[41] J. H. Laros III, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Vandyke, and C. Vaughan. Energy delay product. In *Energy-Efficient High Performance Computing*. Springer, 2013.

[42] D. Loghin and Y. Teo. The energy efficiency of modern multicore systems. In *ICPP*, 2018.

[43] React Native. Animations. https://reactnative.dev/docs/animations, Accessed 2022-10-18, October 2022.

[44] J. Duplessis. Using native driver for animated. https://reactnative.dev/blog/2017/02/14/using-native-driver-for-animated, Accessed 2022-10-18, 2017.

[45] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. Anti-patterns and the energy efficiency of Android applications. *CoRR*, abs/1610.0, 2016.

[46] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105:43–55, 2019.
[47] Luis Cruz and Rui Abreu. Catalog of energy patterns for mobile applications. *EMSE*, 24(4):2209–2235, Aug 2019.
[48] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Penta, R. Oliveto, and D. Poshyvanyk. Multi-objective optimization of energy consumption of guis in android apps. *TOSEM*, 27(3), 2018.
[49] R. Saborido, R. Morales, F. Khomh, Y. Guéhéneuc, and G. Antoniol. Getting the most from map data structures in Android. *Empirical Software Engineering*, 2018.
[50] Abdul Ali Bangash, Karim Ali, and Abram Hindle. Black box technique to reduce energy consumption of android apps. In *ICSE-NIER*, pages 1–5, 2022.
[51] A. Bangash, D. Tiganov, K. Ali, and A. Hindle. Energy efficient guidelines for ios core location framework. In *ICSME*. IEEE Computer Society, 2021.
[52] S. Georgiou and D. Spinellis. Energy-delay investigation of remote inter-process communication technologies. *Journal of Systems and Software*, 162:110506, 2020.
[53] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.
[54] H. Loidl, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. Michaelson, R. Peña, S. Priebe, Á.J. Rebón, and P. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16:203–251, 09 2003.
[55] S. Georgiou, M. Kechagia, and D. Spinellis. Analyzing programming languages' energy consumption: An empirical study. In *PCI*, 2017.
[56] Matteo Ciman and Ombretta Gaggi. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *PMC*, 39:214–230, 2017.
[57] Google developer experts. https://developers.google.com/community/experts. Accessed: 2023-01-08.
[58] I. Malavolta, E. . Grua, C. Lam, R. de Vries, F. Tan, E. Zielinski, M. Peters, and L. Kaandorp. A framework for the automatic execution of measurement-based experiments on android devices. In *ASE Workshops*, page 61–66, 2020.
[59] Fares Bouaffar, Olivier Le Goaer, and Adel Noureddine. Powdroid: Energy profiling of android applications. In *ASEW Workshops*, pages 251–254, 2021.
[60] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-based energy profiling of Android apps: Simple, efficient and reliable? In *SANER*, pages 103–114, 2017.