Native or Web? A Preliminary Study on the Energy Consumption of Android Development Models

Wellington Oliveira, Wesley Torres, Fernando Castor, Bianca H. Ximenes
Federal University of Pernambuco
Recife, PE, Brazil
{woj, wst, castor, bhxmm}@cin.ufpe.br

Abstract—Energy consumption has become an increasingly important topic in software development, especially due to the ubiquity of mobile devices, and the choice of programming language can directly impact battery life. This paper presents a study aiming to shed some light on the issue of energy efficiency on the Android platform, comparing the performance and energy consumption of 33 different benchmarks in the two main programming languages employed in Android development: Java and JavaScript. Preliminary results show that Java applications may consume up to 56x more energy, with a median of 2.28x, than their JavaScript counterparts, in benchmarks that are mostly CPU-intensive. In some scenarios, though, the Java benchmarks exhibited better energy efficiency, with JavaScript consuming up to 2.27x more energy. Based on these results, two Java applications were re-engineered, and through the insertion of JavaScript functions, hybrid applications were produced. In both cases, improvements in energy efficiency were obtained. Considering that Android apps written in Java are the norm, the results from this study indicate that using a combination of JavaScript and Java may lead to a non-negligible improvement in energy efficiency.

I. INTRODUCTION

As smartphones become popular and diverse platforms emerge (Android, iOS, Windows Phone), developers must choose whether to create applications using the native language of a mobile OS or the Web toolkit (HTML, CSS and JavaScript), subsequently porting the app using a specific framework (Cordova\(^1\), Ionic\(^2\)). In addition to having to decide which model to employ, developers have little to no information available on the difference between the performance and battery consumption for these approaches, making it more difficult to determine which one is more adequate.

Currently, the most popular OS in smartphones and tablets is Android\(^3\). Previous work has analyzed the energy consumption of Android from a number of different perspectives \(^4\), \(^5\), \(^6\). However, one perspective that has not yet been analyzed is the impact of different approaches for Android development on energy consumption. Android apps can be written entirely in Java (native apps), in JavaScript-related technologies (Web apps), or in a combination of both (hybrid apps) Most of the Android apps, however, are written entirely in Java. In a sample of 108 projects we examined from F-Droid\(^7\) only 4% use Javascript. It is not yet clear whether this approach leads to energy-efficient applications.

This paper sheds some light on the issue of the energy efficiency of Android app development approaches. We compare the energy consumption and performance of 33 benchmarks developed by several authors from Rosetta Code\(^8\) and The Computer Language Benchmark Game\(^9\). To measure energy consumption, we employed Android’s Project Volta \(^1\). Our goal with this study is to provide a preliminary answer to the following research question:

- **RQ1. Is there a more energy-efficient approach among the two most common Android development models?**

The JavaScript versions of 26 out of the 33 analyzed benchmarks exhibited lower energy consumption. The Java versions of six of these benchmarks outperformed their JavaScript counterparts, even though they consumed more energy. This result indicates that, at least for CPU-intensive apps, Java may not be the most energy-efficient solution. Most of their execution time is spent waiting for user input or using sensors. This led us to question whether one could save energy by using a hybrid approach and, e.g., adopting JavaScript in the more CPU-intensive parts of applications. Thus, we also provide an initial answer for the following additional research question:

- **RQ2. Is it possible to reduce the energy consumption of an app built using a single approach by making it hybrid?**

We reengineered two existing apps from F-Droid written in Java and made parts of them run in JavaScript. We analyzed different models for the Java part of the apps to invoke the JavaScript part and measured the energy consumption in all the cases. Our results indicate that it is possible to save energy using this hybrid approach. In one of the apps, TriRose, the hybrid version saved up to 30% energy by grouping invocations to the JavaScript part. Performing the same grouping of operations using only Java yielded only marginal gains in terms of energy consumption.

Knowing whether small modifications in the code promote a non-negligible reduction in energy consumption empowers developers, as they can then make informed decisions on which language they should use, or even opt for a hybrid approach. Moreover, tool builders can introduce cross-language

---

\(^1\)https://cordova.apache.org/
\(^2\)http://ionicframework.com/
\(^3\)http://f-droid.org
\(^4\)http://benchmarksgame.alioth.debian.org
\(^5\)http://rosettacode.org/wiki/Rosetta_Code
\(^6\)http://benchmarksgame.alioth.debian.org
\(^7\)http://www.idc.com/prodserv/smartphone-os-market-share.jsp
refactorings that support developers in reengineering existing applications when a hybrid approach may be beneficial. Also, program analyses in these tools can help developers in identifying these cases.

II. METHODOLOGY

The aim of this study is to analyze the two most popular programming approaches for Android app development and to establish whether they differ in terms of energy efficiency and performance. In this Section, we describe how we selected the analyzed benchmarks (Section II-A) and explain our experimental procedure (Section II-B).

A. Benchmarks

In this section, we present the benchmarks we studied to address RQ1 (Section II-A). The two apps we have reengineered to employ a hybrid approach are discussed in Section III-B.

Benchmarks were extracted from Rosetta Code and The Computer Language Benchmark Game (TCLBG). Rosetta Code is a programming chrestomathy site. It includes a large number of programming tasks and solutions to these tasks in different programming languages. TCLBG is a website whose main purpose is to compare the performance of several programming languages. Both have been employed in prior work for comparing different programming languages [9] and to analyze energy efficiency [7].

The benchmark set encompasses 23 benchmarks from Rosetta Code and 10 from TCLBG. All the benchmarks originally had versions written in both Java and JavaScript. Table I lists all the benchmarks employed in this study. Since the benchmarks from Rosetta Code were not built with performance as a design goal, their performances vary widely. For example, for the nQueens benchmark, the solutions available at Rosetta Code took 20s to finish in Java and 69s to finish in JavaScript. By converting the JavaScript version to use the same algorithm as the Java version, it took 12s to finish. Aiming to compare the languages and not the implementation of an algorithm, we have analyzed all the Rosetta Code benchmarks and, where necessary, performed similar modifications.

In TCLBG, the approach the developers employ to algorithm optimization is different. They are trying to achieve the best possible performance for the main objective of this platform is to compare which language is the fastest to perform a given benchmark. The only modifications applied to the TCLBG benchmarks were the ones needed to execute them in the Android environment.

All benchmarks were executed using a preset workload, individual to each benchmark. The size of each workload was determined in a way that the benchmark was executed for at least 20s. Considering the two versions of each of the 33 benchmarks, for only 5 (out of the 66) we had a standard deviation greater than 18% of the mean value for the energy measurements. This indicates that, for the experimental configuration we employed, the results are stable.

B. Running the experiments

For the tests, the device used was a Nexus 5 (2013), running Android 5.1, with 16gb of flash memory, 2gb of RAM memory, chipset Qualcomm MSM8974 Snapdragon 800, CPU Quad-core 2.3 GHz Krait 400 and Li-Po 2300 mAh battery. Each benchmark and app was executed 8 times in each programming language. All data about energy consumption was collected using Project Volta. Execution time measurements were obtained using custom-built scripts.

All the experiments were executed observing the following step-by-step procedure:

1) Verify whether the device battery is at least 80% charged, as a means to prevent it from entering battery-save mode and keep the voltage in at least 4V.
2) Close all running applications not involved in the tests, activating airplane mode, and immediately rebooting the device. This step aims to isolate the application behavior, preventing other apps from remaining running or sensors such as Wi-Fi or GPS from interfering with the results.
3) Connect the device to the computer and reset all data of battery consumption, disconnecting it afterwards.
4) Execute the application or benchmark.
5) Prevent the app from running in the background at all times, not locking the screen, not allowing the screen to shut down or changing to another app. By locking the screen, the app starts to run in the background, not using the CPU as it normally would.
6) Plug the device to the computer to verify battery consumption data and execution time.

III. STUDY RESULTS

This section presents the results for the two research questions. Section III-A presents the results for RQ1 whereas Section III-B examines RQ2.

A. Is there a more energy-efficient app development approach?

In our experiments, the JavaScript versions of most benchmarks consumed less and energy and were faster. Moreover, this result was not universal and in some of the benchmarks the Java version was faster or consumed less energy.

Figure 1 shows the execution time (lines) and energy consumption (bars) of the Rosetta Code benchmarks. Overall, the JavaScript benchmarks exhibit lower energy consumption and execution time. The Java versions of these benchmarks...
consume a median 2.09x more energy than their JavaScript counterparts. Furthermore, the Java versions spend a median 1.52x more time to finish the execution. The figure shows that in 18 out of 22 of the benchmarks from Rosetta, the JavaScript versions consume less energy and in 16 they exhibited lower execution time. The results for the benchmark “sequence of non-squares” are not in Figure 1. It was omitted for the sake of graphic readability. The average time and consumption in Java were 670s and 570J and in JavaScript they were 16s and 10J, respectively. Although the result of this benchmark is not present in the graph, it was included in the calculations for consolidated results alongside all other benchmarks. Finally, in 3 of the 7 benchmarks where Java was faster, it also consumed more energy than JavaScript, which suggests a non-linear relation between energy and performance.

Figure 2 presents the results for the benchmarks from TCLBG. The Java versions of these benchmarks consume a median 1.82x more energy than their JavaScript counterparts. The figure shows that 7 out of the 10 benchmarks consume less energy in JavaScript. However, differently from the benchmarks from Rosetta Code, the median execution time of the Java versions of the benchmarks from TCLBG is 0.67x that of the JavaScript versions. Overall, 6 benchmarks are faster in Java than in JavaScript. The main reason for this behavior is the use of parallelism. Whereas all the JavaScript versions run sequentially, the Java versions of 8 benchmarks are capable of leveraging multicore processors to improve performance. For 5 of these the Java version outperforms the corresponding JavaScript version. However, only two of them also exhibit lower energy consumption. This is consistent with previous work [12] that found that, for programs capable of benefiting from multicore processors, performance is often not a proxy for energy consumption.

Analyzing all benchmarks from Rosetta and TCLGB, in the cases where JavaScript had the best results, Java applications consumed a median 2.28x more energy and their executions where a median 1.59x longer than their JavaScript counterparts. When Java performed better, JavaScript consumed a median 1.40x more energy and a median 1.40x more time.

In a preliminary study such as this, it is difficult to define general heuristics to determine which approach is better in a given situation. However, we observed that benchmarks that relied heavily on the CPU and performed many simple mathematical operations were the ones with the biggest differences in energy consumption, favoring JavaScript. Moreover, it is easier to leverage parallelism in Java, which contributes positively to the performance of the Java versions of the benchmarks. Nonetheless, that extra performance does not seem to amount to reduced energy consumption in general. Finally, benchmarks that relied heavily on memory or files did not favor any language.

B. Can a hybrid approach to app development save energy?

The results discussed in Section III-A suggest that the two approaches for app development in Android have different trade-offs in terms of energy consumption. However, Android applications are predominantly written in Java, independently of the impact of this approach on energy consumption. For example, in a random sample of 108 apps among the 1,600 in F-Droid, only 5 employed JavaScript in any way. However, in Android, it is possible for Java code to invoke JavaScript and vice-versa. Thus, since Java is the predominant approach to write Android apps, it may be possible to save energy by retrofitting existing apps to perform part of their work in JavaScript. The major obstacle to this approach is that there is the overhead of cross-language invocations [4]. In this section we examine whether it is possible to overcome this overhead so as to make existing apps more energy-efficient.

Benchmarks are not useful to provide an answer for RQ2, since they work differently from apps [13]. Therefore, we have used two real-world, open-source apps for this part of the study. These apps appear in the last row of Table 1. TriRose is an app that mathematically generates unique and intricate rose graphs (rhodonea curves) and anDOF is an app to calculate depth of field for photography. These apps were chosen because, even though they spend much of their time on input and output operations, they perform a non-trivial amount of computation. TriRose comprises 1kLoC and anDOF 1.7kLoC.

When reengineering parts of a Java app to use JavaScript, it is important to determine the frequency with which the Java part will invoke the JavaScript part. If the former invokes the latter too frequently, much of the execution time and possibly energy consumption will be dominated by the overhead of cross-language invocations. If these invocations are too infrequent, application functionality may be compromised. In this
work, we used three different approaches to manage cross-
language invocations. In the Stepwise approach, one method in Java is mapped directly to a function in JavaScript and each time the method is supposed to be called, the function is used instead. In the Batch approach, one method in Java is mapped to a function in JavaScript, bundling several calls of the method. This function returns an aggregated result that the Java part processes to update the screen. This approach reduces the communication overhead by dividing processing duties between JavaScript and Java. Finally, in the Export approach, all the work to be performed in a sequence of method invocations in the original version is mapped to a single JavaScript function.

Tri Rose and anDOF have different behaviors. For the former, the delay of waiting for JavaScript to perform the entire computation (the Export approach) is not acceptable whereas it is for anDOF. Thus, for anDOF we employed the Stepwise and Export approaches and for Tri Rose Stepwise and Batch. The specific workload for each app is determined in way to try to make each execution run in approximately 30s, keeping a low relative standard deviation. The workload for the Stepwise approach for anDOF is 3,000 changes in a scroll that controls the DOF. For each change, a method is called to recalculate the depth of field. The workload for Export is 4,000,000 changes since, for this benchmark, it runs three orders of magnitude faster than the Stepwise approach. In Tri Rose, the workload for both the Stepwise and Batch approaches consisted of drawing 1.5k lines on the screen, since the execution times were more similar.

Figure 2 presents the results. In both apps, using Stepwise degraded performance and boosted energy consumption. This indicates that the amount of overhead generated by the thousands of requests made in JavaScript increases the energy consumption to a point where it is not possible to reverse the situation with a possibly faster execution in JavaScript. This result suggests that unless it is possible to group parts of the work so as to minimize this overhead, building a hybrid app will not save energy. In every Stepwise test, the original Java version had a better performance and lower energy consumption.

Using Batch, the execution was changed to keep the results of the calculations of the points that were used to draw the curves in a buffer of 110k positions. We apply this modification to both the Java version and the hybrid version, thus producing two Java versions of this app, besides the hybrid version. The Java version using this Batch approach consumed 30% more energy than the hybrid version. One can note that despite the great number of data computed in JavaScript, this application spent most of the time drawing on the screen, using Java code. JavaScript had a lower energy consumption but Java had a better performance.

We only apply the Export to anDOF. It would not make sense in the context of Tri Rose because the latter needs to continually update the screen. Updating the screen from JavaScript code in a Java app is non-trivial because Java and JavaScript employ different paradigms for user interface. The results using the Export approach represent a (potentially unrealistic) best case scenario, since the cross-language invocation overhead is almost entirely diluted. Nevertheless, it indicates that if an application needs to perform a substantial number of calculations, using JavaScript could lead to a significant improvement in energy efficiency and performance. The Java version consumed 35.69x more energy and took 32.77x longer.

Even though these modifications promoted considerable improvements in energy efficiency, they did not require large-scale modifications. JavaScript files for each app had 160 (anDOF) and 100 (Tri Rose) LoC. Changes were relatively simple and represent less than 10% of the code of each app.

### IV. Threats to Validity

To minimize the risk that one defective device would undermine the research, all benchmarks from TCLBG were executed in a second device with similar specs, but from
another manufacturer and Android version. Raw values of time and energy were similar and the relation between which of the languages performed better for each benchmark remained the same. The second device used was a Moto G3 (2015) using Android 5.1.1, with 8gb of HD, 1gb of RAM memory, chipset Qualcomm MSM8916 Snapdragon 410, CPU Quad-core 1.4 GHz Cortex-A53 and Li-Ion 2470 mAh battery.

Project Volta is a new tool for measuring energy and its accuracy has not yet been assessed. However, even if the measurements in Volta are not precise, the results of this research are still relevant because the most important data is the comparison and relation between the performance and energy consumption for each language, and not their values.

Another approach to measuring energy consumption is the comparison and relation between the performance and energy consumption for each language, and not their values. Benchmarks do not represent the behavior of an application using JavaScript as main programming language [13], and for that reason it is not possible to extrapolate the results for all applications, since applications are usually much more IO-intensive. Although this is true, benchmarks provide insight on scenarios where the gain in performance is measured, by isolating usage pattern behavior. In our case, the focus was on apps that make intensive use of the CPU.

It is possible to write parts of the app in C/C++ using the Native Development Kit (NDK), which aims to improve performance. Nevertheless, it is more common in Android development to code in Java, JavaScript, or to combine both than otherwise, due to the innate complexities of the NDK, as it is pointed out in Google’s own website[7]. It is not possible to create a app entirely with NDK. Therefore, our choice is based on industry practices.

V. RELATED WORK

Energy consumption is a hot issue in software development as a whole, not only in mobile development. One can find several papers regarding software energy consumption optimization [11], [12]. Pathak et al. [10] proposed the first fine-grained energy profiler to investigate where the energy is spent inside an app. Some hve attempted to find which methods [3], API-calls [8] and applications [14] for Android are more energy-hungry. Some even specifically aim to find which code lines are the most battery-draining [5], [6]. To the best of our knowledge, no previous work has analyzed the impact of different app development approaches on energy consumption.

The study by Charland et al. [2] is the closest work to this paper, since it compares the native and Web app development approaches. However, it focuses on user interface code, user experience, and performance for remote web apps. In particular, it does not present data on battery consumption or performance of native applications and local web applications. Nanz et al. [9] have used the Rosetta Code tasks to compare eight different programming languages in terms of the runtime performance, and memory usage, among other factors. It does not focus on Android, however, nor analyze energy consumption.

VI. CONCLUSION

This research aimed to start an investigation on whether there is a more energy-efficient approach for android app development. It sheds some light on the strengths and weaknesses of each approach, based on experiments with benchmarks and hybridized apps that use both Java and Javascript. Preliminary results suggest that, at least for CPU-intensive operations, JavaScript outperforms Java. We have also conducted a preliminary analysis on the potential benefits of using a hybrid approach for app development, considering that most apps are written in Java. We found out that, if it is possible to avoid the need to perform multiple cross-language invocations, using a hybrid approach may lead to energy savings. In the future, we plan to do a more exhaustive investigation on the reasons that why one approach outperforms or consumes less energy than the other.

REFERENCES


[12] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: comparing languages performed better for each benchmark remained the same. The second device used was a Moto G3 (2015) using Android 5.1.1, with 8gb of HD, 1gb of RAM memory, chipset Qualcomm MSM8916 Snapdragon 410, CPU Quad-core 1.4 GHz Cortex-A53 and Li-Ion 2470 mAh battery.

Project Volta is a new tool for measuring energy and its accuracy has not yet been assessed. However, even if the measurements in Volta are not precise, the results of this research are still relevant because the most important data is the comparison and relation between the performance and energy consumption for each language, and not their values.

Benchmarks do not represent the behavior of an application using JavaScript as main programming language [13], and for that reason it is not possible to extrapolate the results for all applications, since applications are usually much more IO-intensive. Although this is true, benchmarks provide insight on scenarios where the gain in performance is measured, by isolating usage pattern behavior. In our case, the focus was on apps that make intensive use of the CPU.

It is possible to write parts of the app in C/C++ using the Native Development Kit (NDK), which aims to improve performance. Nevertheless, it is more common in Android development to code in Java, JavaScript, or to combine both than otherwise, due to the innate complexities of the NDK, as it is pointed out in Google’s own website[7]. It is not possible to create a app entirely with NDK. Therefore, our choice is based on industry practices.

V. RELATED WORK

Energy consumption is a hot issue in software development as a whole, not only in mobile development. One can find several papers regarding software energy consumption optimization [11], [12]. Pathak et al. [10] proposed the first fine-grained energy profiler to investigate where the energy is spent inside an app. Some hve attempted to find which methods [3], API-calls [8] and applications [14] for Android are more energy-hungry. Some even specifically aim to find which code lines are the most battery-draining [5], [6]. To the best of our knowledge, no previous work has analyzed the impact of different app development approaches on energy consumption.

The study by Charland et al. [2] is the closest work to this paper, since it compares the native and Web app development approaches. However, it focuses on user interface code, user experience, and performance for remote web apps. In particular, it does not present data on battery consumption or performance of native applications and local web applications. Nanz et al. [9] have used the Rosetta Code tasks to compare eight different programming languages in terms of the runtime performance, and memory usage, among other factors. It does not focus on Android, however, nor analyze energy consumption.

Google about NDK: “(...) has little value for many types of Android apps. It is often not worth the additional complexity it inevitably brings to the development process.”