2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

A Study on the Energy Consumption of Android App Development Approaches

Wellington Oliveira, Renato Oliveira, Fernando Castor
Federal University of Pernambuco
Recife, PE, Brazil
{woj, ros3, castor}@cin.ufpe.br

***Abstract*—Mobile devices have become ubiquitous in the recent years, but the complaints about energy consumption are almost universal. On Android, the developer can choose among several different approaches to develop an app. In this paper, we investigate the impact of some of the most popular development approaches on the energy consumption of Android apps. Our study uses a testbed of 33 different benchmarks and 3 applications on 5 different devices to compare the energy efficiency and performance of the most commonly used approaches to develop apps on Android: Java, JavaScript, and C/C++ (through the NDK tools). In our experiments, Javascript was more energy-efficient in 75% of all benchmarks, while their Java counterparts consume up to 36.27x more energy (median of 1.97x). On the other hand, both Java and C++ outperformed JavaScript in most of the benchmarks. Based on these results, four Java applications were re-engineered to use a combination of Java and either JavaScript or C/C++ functions. For one of the apps, the hybrid solution using Java and C++ spent 10x less time and almost 100x less energy than a pure Java solution. The results were not uniform, however. For another app, when we restructured its implementation so as to minimize cross-language method invocations, the hybrid solution using Java and C++ took 8% longer to execute and consumed 11% more energy than a hybrid solution using Java and JavaScript. Since most Android apps are written solely in Java, the results of this study indicate that leveraging a combination of approaches may lead to non-negligible improvements in energy-efficiency and performance.**

I. Introduction

In the recent years, computers have gone through substantial hardware changes, comprising high-definition screens and multicore processors. Besides, new patterns of use for gadgets are emerging, as people have the need to be constantly connected and mobile. However, since these devices run on battery power, energy efficiency has grown in importance. Battery life is seen as very important to most smartphone owners [1], with as much as 92% considering battery life as a significant factor when purchasing a new smartphone, based on a survey with 400 people. According to the same survey, 63% of all smartphones owners are somewhat unsatisfied with theirs devices' battery and 66% would pay more for a device with longer battery life.

In recent years, Android has become a ubiquitous platform. More than 85% of all smartphones in the world use this operating system [2] and the Google Play store has more than 2 million apps [3] available for download. In this paper, our focus is on app developers. Currently, there are three approaches to develop apps that are directly endorsed by Google: Java, the default language for Android development; JavaScript, since, with the support of some frameworks, one can develop a full app using the web toolkit (HTML, CSS and JavaScript); and C/C++, through the Native Development Kit (NDK) [4], which makes it possible for a developer to write the majority of an application in C/C++. In the latter case, there is a trade-off between an increase in complexity and the benefit of (potentially) improved performance. In summary, Android apps can be written entirely in Java (**native** apps), in JavaScript-related technologies (**web** apps), or in a combination of more than one programming language (**hybrid** apps)[1]. Most of the Android apps, however, are written entirely in Java. In a sample of 109 projects we examined from F-Droid, only 4% use Javascript and 4% use the NDK resources to improve performance.

Although one can find scientific and anecdotal evidence about the performance of Android apps written using these different development approaches, it is hard to find data about the differences in energy consumption. It is not yet clear whether the three aforementioned approaches lead to energy-efficient applications.

This paper aims to shed more light on the issue of energy efficiency among the different Android app development approaches. We compare energy consumption and performance of 33 benchmarks developed by several authors from Rosetta Code [5] and The Computer Language Benchmark Game (TCLBG) [6]. Our study consisted of executing multiple versions of each benchmark on a number of different mobile devices, while measuring execution time and energy consumption.

To measure energy consumption, we used the resources available on Android since version 5, previously called Project Volta [7]. These resources enable us to collect energy consumption information on a per-app basis. Our goal with this study is to provide an answer to the following research question:

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

We found out that for 26 out of the 33 analyzed benchmarks, JavaScript versions exhibited lower energy consumption than their Java counterparts. The Java versions of six of these benchmarks outperformed their JavaScript counterparts, even

[1]The hybrid app demonination its also used to refer to JavaScript-powered apps. In this work, we consider a hybrid app to be any app that uses more than one development approach.

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.66

though they consumed more energy. This result indicates that, at least for CPU-intensive apps, Java may not be the most energy-efficient solution.

For the TCLBG benchmarks, we compared Java, JavaScript, and C++ versions, executing them on five devices with different characteristics, achieving similar results. We found out that, although there are some small variations in performance, the energy consumption relation between Java, JavaScript, and C++, remained similar across devices, with JavaScript having the edge over the other approaches concerning energy consumption, while exhibiting a good trade-off between energy and performance. We also noticed that using the NDK does improve performance.

Even though these are interesting results, Android apps in general behave differently from CPU-intensive benchmarks [8]. 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, adopting JavaScript and C++ in the more CPU-intensive parts of applications. Thus, we also provide an initial answer for the following research question:

- **RQ2. Is it possible to reduce the energy consumption of a native app by making it hybrid?**

We reengineered four open-source apps: three existing apps from the F-Droid repository [9] and one app developed by the National Institute of Science and Technology for Software Engineering (INES) [10]. Three of those apps are also available at the Play Store. Each app was written in Java and we made parts of them run in JavaScript and C++. Our goal was to analyze whether using these approaches alongside Java impacted performance and energy consumption. From the benchmark analyses, we knew that using JavaScript and C++ often led to an improvement in performance, energy consumption, or both. However, we had no information about whether it was possible to make improvements in performance and energy consumption by using a hybrid approach. We analyzed different models for invoking Javascript and C++ snippets using Java code and measured the energy consumption in all the cases. Our results indicate that it is possible to save energy using this hybrid approach - for one of the apps, a hybrid version using a combination of Java and C++ consumed 0.37J, under a certain workload, whereas the original version written entirely in Java consumed 32.92J.

Knowing whether small modifications in the code promote a non-negligible reduction in energy consumption empowers developers. Moreover, tool builders can introduce cross-language refactorings that support developers in reengineering existing applications when a hybrid approach may be beneficial. All the data from this study can be found at https://secaada.github.io/msr2017/

II. Related Work

For a long time, the study of the energy consumption of computing systems was targeted mainly at the hardware and operating system levels. Tiwarei et al. [11] showed, however, that software is also a fundamental part of energy consumption. Since then, several papers have proposed solutions to help software developers in measuring [12], [13], analyzing [14], [15], [16], and optimizing [15], [17] the energy consumption of the systems they build.

Pathak et al. [18] proposed the first fine-grained energy profiler to investigate where the energy is spent inside an app. This study reveled that the majority of the energy in apps is spent on I/O operations. In particular, free apps spend a considerable amount of energy due to third-party advertisement modules. Since I/O operations happen without much interaction with methods or functions, it is proposed to group the I/O operations together in a bundle, as a way to minimize the energy consumption of those operations.

Some have attempted to find which methods [19], API-calls [20], and apps [21] for Android are more energy-hungry. Li et al. [13] developed an accurate approach to find which source code lines are the most battery-draining for Android apps, using a combination of hardware, path profiling, and instrumentation to associate the energy data with the application. An evaluation tool was made to evaluate the approach and executed on five apps from the Play Store. The evaluation tool was developed to be run on apps that used the Dalvik Virtual Machine, which was discontinued after the fifth version.

More recently, Di Nucci et al. [22] developed an energy profiling tool to assist in the measurement and analysis of the energy consumption of Android applications. It works by installing an apk file containing the application to be tested on an Android device, running it with pre-defined test cases and using the Android APIs to collect information about energy. Experimental results have shown that its accuracy is comparable to that of an external hardware monitor.

A promising approach to save energy without the need for specialized knowledge is to leverage design diversity [23], more specifically, the availability of diversely designed implementations of the same abstractions. Recent work has explored the impact of different thread management constructs [24], [25], data structures [26], [27], API calls [20], and concurrency control primitives [24]. This work complements previous studies by analyzing how the availability of different development approaches can impact energy in Android apps.

The study by Charland et al. [28] 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. [29] have used the Rosetta Code tasks to compare eight different programming languages in terms of the run-time performance and memory usage, among other factors. They have not focused on Android and do not analyze energy consumption.

This paper is an extension of a previous work, published on the early research achievements (ERA) of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) [30]. We extended it in several ways: the benchmarks were executed in four more devices and in each of the new devices, the benchmarks were executed three times more; we used C++ versions of the benchmarks and apps through the NDK tools to compare performance and

energy consumption with Java and JavaScript; we performed a deeper analysis of the data, analyzing several metrics; we tested two more apps and compared hybrid implementations using JavaScript and C++ with the original implementations; we compared the performance of all benchmarks executed in the previous work with their counterparts on a traditional computer.

III. Android App Development

Traditionally, in mobile environments, native applications, i.e., developed using a native language of the operating system, have been regarded as faster, safer, and more adaptable to changes in the operating system [31]. In the iOS ecosystem, native apps are written in Objective-C and Swift whereas Android native apps are written in Java. On the other hand, web apps, i.e., applications developed using Web technologies, are seen as more portable, with lower maintenance and development costs, and are faster to get to market [31]. At the same time, it is expected that these web apps run slower than native ones. This section provides a high level overview of web development frameworks and briefly presents the NDK, a toolkit aiming to support the development of apps with strict performance requirements through the use of C and C++ code.

Web application framework. Developers who choose to develop web apps without prior experience in mobile development may find it more difficult, as there are not many tutorials or support from IDEs as in the native language. Android Studio, Google's recommended IDE for Android development, offers limited support for building web apps. In addition, most APIs are native, which means that they are directly accessible to Java code but require a plugin in the case of web apps.

Mobile development frameworks aim to ease the construction of smartphone applications based on Web technologies, such as HTML5, CSS3, and JavaScript. These frameworks are intended to allow the use of standard web development technologies for cross-OS development, i.e., the application is developed once and ported to several different mobile operating systems freeing the developer from having to deal with the native language of each operating system. These applications are then wrapped, so the operating system can identify them as traditional applications and give them access to native APIs.

This work was developed using the Apache Cordova mobile development framework. Cordova is open source and is used as basis for other popular frameworks like Phonegap, Ionic, Intel XDK, Telerik, Monaca and TACO. Among the contributors to the Apache Cordova project are IBM, Google, Microsoft, Intel, Adobe, Blackberry, and Mozilla[2]. We employed Cordova mainly because our experiments involved the execution of multiple benchmarks that were implemented in JavaScript but did not specifically target Android.

When using Cordova, the application is developed as a web page, being able to reference CSS files, JavaScript code, images, media files, or other resources needed to run it. This app will run over an encapsulated WebView within a native application. An application using Cordova and native components can communicate, i.e., JavaScript code can invoke native snippets directly. Ideally, JavaScript APIs for native code are consistent across multiple platforms and operating systems. There are also externally available plugins that allow a developer to use features not available on all platforms. In Android, web apps produced using Cordova will be compiled into an .apk file, which is the default distribution format for Android applications.

Native Development Kit. In Android apps that have strict performance requirements, it is possible to develop parts of an application using C/C++ through the Native Development Kit (NDK). This approach is often only employed in specialized scenarios, most notably games. For example, in the sample of 109 apps we examined (Section I), only 5 employed the NDK and all of them are games. Although it might seem a good idea to use the NDK as much as possible to make an application achieve maximum performance, Google does not recommend it. According to the Android website [4], there are two situations where the use of the NDK is recommended: (i) when it is necessary to *"squeeze extra performance out of a device to achieve low latency or run computationally intensive applications, such as games or physics simulations"*; and (ii) to *"reuse your own or other developers' C or C++ libraries"*.

The NDK is an optional package that can be installed using the *Android SDK Manager*. Unlike building native and web apps, using the NDK is not straightforward, as pointed out on the Android website: *"(...)The NDK may not be appropriate for most novice Android programmers who need to use only Java code and framework APIs to develop their apps."*. One example of complication that arises from the use of the NDK is the need to manually manage memory, since Android keeps two separate heaps, one for the Android Runtime and another one for the native parts of the app. Moreover, the setup of an app that uses the NDK requires additional steps such as the construction of an additional build script and the definition of at least one method that will serve as the interface between the Java and C/C++ code.

IV. Methodology

The aim of this study is to analyze the most popular development approaches for Android and to establish whether they differ in terms of energy efficiency and performance. The metric we used for performance evaluation in this paper was execution time.

In this section, we explain how we selected the analyzed benchmarks and apps (Section IV-A), how we measured energy consumption (Section IV-B), and how we executed the experiments (Section IV-C).

A. Benchmarks and apps

Benchmarks were extracted from two software repositories: Rosetta Code and The Computer Language Benchmark Game (TCLGB). Rosetta Code is a programming chrestomathy[3] site. It includes a large number of programming tasks and solutions

[2]http://wiki.apache.org/cordova/who

[3]https://en.wikipedia.org/wiki/Chrestomathy

TABLE I
THE SELECTED SET OF BENCHMARKS AND APPLICATIONS.

Source	Benchmark or App
Rosetta Code	bubblesort, combinations, count in factors, countingSort, gnomesort, happy numbers, heapSort, hofstadterq, insertsort, knapsack bounded, knapsack unbounded, matrix mult, man or boy,mergeSort, nqueens, pancakesort, perfect number, quicksort, seqnonsquares, shellsort,sieve of eratosthenes, tower of hanoi and zero-one knapsack
The Computer Language Game Benchmark	binarytrees, fannkuck, fasta, fasta parallel, knucleotide, nbody, regexdna, regexdna parallel, revcomp and spectral
F-Droid	anDOF, BioNucleus, EnigmAndroid, TriRose

to these tasks in different programming languages. TCLGB 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 [29] and to analyze energy efficiency [24].

Table I lists all the benchmarks analyzed in this study. The benchmark set encompasses 23 benchmarks from Rosetta Code and 10 from TCLBG. Most of the benchmarks from Rosetta have already been used in other studies [29]. All benchmarks from TCLGB that have implementations in all the languages were used.

Since the benchmarks from Rosetta were not built with optimal performance in mind, their performances vary widely. For example, in the nqueens benchmark, the solutions available at Rosetta took 20s to finish in Java and 69s in JavaScript. By converting the JavaScript version to use the same algorithm as the Java version, this new version took 12s to finish. As the research focus is on the development approaches and not algorithm implementations, we analyzed all of Rosetta Code's benchmarks and, whenever necessary, performed similar modifications. Those modifications make the comparisons more balanced. As both languages are syntactically similar, adaptations were straightforward. Since the Rosetta Code benchmarks do not have a strong emphasis on performance, we have only used Java and JavaScript versions of them.

In TCLBG, all implementations try to achieve the best possible performance. Therefore, the only modifications applied to those benchmarks were the ones necessary to execute them in the Android environment. We executed Java, JavaScript, and C++ versions of these benchmarks. Five implementations of benchmarks in Java and C++ used parallelism to solve the problems, thus improving performance. Single-core implementations of such benchmarks, whenever available, were also analyzed, as a way to verify whether a slower execution would lead Java and C++ to be more energy-efficient. All benchmarks in JavaScript were optimized for a single core. As the TCLBG benchmarks were more reliable, we executed them on every device available.

Benchmarks may not be useful to provide an answer for RQ2, since they work differently from apps [8]. Therefore, we have used four real-world, open-source apps for the study. These apps appear in the last row of Table I. anDOF is an app to calculate depth of field for photography, BioNucleus is an app that aims to facilitate the collection of waste cooking oil, EnigmAndroid is an app that simulates the enigma machine from the 2nd World War and TriRose is an app that mathematically generates unique and intricate rose graphs. 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. BioNucleus comprises 1kLoC, TriRose 1.1kLoC, anDOF 1.7kLoC and EnigmAndroid 4.6kLoC. anDOF, BioNucleus and TriRose can also be found at the Play Store.

The four analyzed apps were written entirely in Java. We have reengineered their most compute-intensive parts to use JavaScript and C++, creating hybrid web apps and hybrid NDK apps, respectively. To detect those parts, we used the profiler from Android Studio. It is important to determine the frequency with which Java will invoke the part written in another language. 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 models to manage cross-language invocations. In the *Stepwise* model, one method in Java is mapped directly to a function in JavaScript or C++ and each time the method is supposed to be called, the function is used instead. In the *Batch* model, one method in Java is mapped to a function in JavaScript or C++, bundling several calls of the method, returning the aggregated result to Java. This model reduces the communication overhead by dividing processing duties between Java and the other language. Finally, in the *Export* model, all the work to be performed in a sequence of method invocations in the original version is mapped to a single JavaScript or C++ function. Creating a hybrid app by modifying an existing one instead of coding a new app allowed us to verify whether it was possible to increase energy efficiency with minor modifications, meaning minimum effort for developers. Each execution of the hybrid apps had the same output as the original version.

All benchmarks and applications were deployed using Android Studio v2.2.2.

B. Measuring energy consumption

In order to measure the energy consumption of the benchmarks and apps, it is possible to use a coarse-grained or fine-grained energy measurement approach. Coarse-grained energy measurement attempts to measure the energy consumption of a mobile device as a whole, making no distinction among the various factors that can impact energy consumption. Usually, physical devices are used to make the measurements, using software only to analyze the collected data. The main advantage of coarse-grained energy measurement is that it tends to be precise, due to the use of specialized equipment. In addition, the measurement equipment is external to the device whose energy consumption is being measured and thus does not interfere with it. However, the accuracy of this approach depends on the quality of the employed equipment and on the duration of the phenomenon to be measured [32].

Fine-grained measurement approaches attempt to isolate the energy consumed by specific components of a system. They can focus on hardware components (screen, GPS, processor,

TABLE II
THE DEVICES USED IN THE EXPERIMENTS AND THEIR CHARACTERISTICS

Devices	Year	Android	HD	RAM	Chipset	CPU	Battery (mAh)
LG L90	2014	5.0.2	8GB	1GB	Snapdragon MSM8226 SD 400	Quad-core 1.2 GHz Cortex-A7	2540
Nexus 5	2013	5.1	16GB	2GB	Snapdragon MSM8974 SD 800	Quad-core 2.3 GHz Krait 400	2300
Nexus 7	2012	5.1.1	16GB	1GB	Tegra 3	Quad-core 1.2 GHz Cortex-A9	4325
Samsung J7	2015	5.1	16GB	1.5GB	Exynos 7580	Octa-core 1.5 GHz Cortex-A53	3000
Zenfone Selfie	2015	6.0.1	32GB	3GB	Snapdragon MSM8939 SD 615	2x Quad-core Cortex-A53	3000

etc.), applications, or even more specific elements, such as methods and lines of code. The most prominent advantage of fine-grained energy measurement is that it is possible to analyze energy consumption for specific hardware and software components, sometimes at a very fine level of granularity [13]. Its main disadvantage is that obtaining information at a fine level of granularity is difficult, often imposing instrumentation overhead or requiring imprecise estimates.

Since the release of the fifth version of Android (Lollipop), Google launched several tools to help developers analyze the energy consumption of their apps at a fine level of granularity. Initially called Project Volta, this set of tools automatically collects data about each application being used since the last time the device was charged. This data comprises the energy consumed by each app, the time spent on the application, the CPU time that this particular application used and, in case the application used some resource like Wi-Fi or GPS, how much energy this resource spent.

In this work, we chose to use the tools made available by Google because of some advantages over other fine-grained approaches. Firstly, there is no need to instrument the code. Thus, it is possible to measure the energy consumption of any app that is running on an Android device. Secondly, this solution is not an app, i.e., there is no risk of another app interfering on the measurements. Finally, it is available for any device running (or upgradeable to) Android 5+.

The energy and performance tools available in Android give us the information about execution time and CPU usage in minutes and seconds, but the energy information is given in milliampere-hour. One cannot use milliampere-hour to analyze energy consumption as this is a unit of electric charge. We converted it to joules by using the following equalities:

1) $1 milliampere \times hour = 3.6 coulomb$
2) $Energy = Charge \times Voltage$

The voltage of a mobile device is dynamic, so we collected voltage from two devices (Samsung J7 and Zenfone) and compared it with a linear distribution using the Kolmogorov-Smirnov (KS) test. We got a p-value of 0.847 for the Samsung J7 and 0.964 for the Zenfone, suggesting that the voltage of those devices decreases in linear progression. For the rest of the devices, we used the mean between the maximum and the minimum voltage for each device. This approximation was only possible because we assumed that the decrease in voltage was linear as indicated by the KS test.

C. Running the experiments

All benchmarks were executed using a preset workload, individual to each benchmark. The size of each workload was determined so as to guarantee it was executed for at least 20s even though sometimes that was not possible due to memory limitations. All web versions of the benchmarks were executed inside a wrapper originated from Apache Cordova.

When working with the C++ versions of the benchmarks, it was not always possible to pick the fastest version available in TCLBG. This is due to the fact that some of the low-level approaches that were used in order to achieve maximum performance were not available in NDK. For example, the fastest solution of the revcomp benchmark uses the not thread safe version of the functions *fwrite* and *fgetc*. For that reason we tried to use the fastest compiling version of the benchmarks.

Table II lists devices we employed in this study. Five devices were used to run the benchmarks, four smart phones and a tablet. We tried to achieve diversity by selecting devices with different manufacturers, chipsets, and CPUs. All devices run the Lollipop version of Android (5.x) or a more recent one. Updates to Android were only applied by means of its updating tool, to emulate a realistic usage scenario. Alternatively, we could have manually installed the same version on every device but this method would require unofficial versions. Thus, we ended up with some slight variations of Lollipop.

We executed every version (Java, JavaScript, C++) of the TCLBG benchmarks at least 10 times in each device. In most cases, however, each benchmark version was executed 30 times in each device. Only for the Nexus 5 smartphone did we execute the Java and JavaScript versions of the benchmarks 10 times. The reason for this lower number of executions is that the phone stopped working before we could perform the full 30 executions. For the remaining four devices, we performed 30 executions of each version of each TCLBG benchmark. All the C++ versions of the benchmarks were executed at least 30 times in each device. The versions of the Rosetta Code benchmarks were executed in the Nexus 5 only. The execution time and battery measurements were obtained using custom-built scripts. To collect measurement data, we have developed a dashboard that, combined with the aforementioned scripts, automatically performs clean-up on the device, loads the app to be executed, executes the app, and records the results.

We have also executed each version (native, hybrid web, and hybrid NDK) of each of the apps mentioned in Section IV-A 30 times. Each app was executed using a predefined workload aiming to simulate realistic usage scenarios. Alternatively, we could also have employed an automated testing tool, such

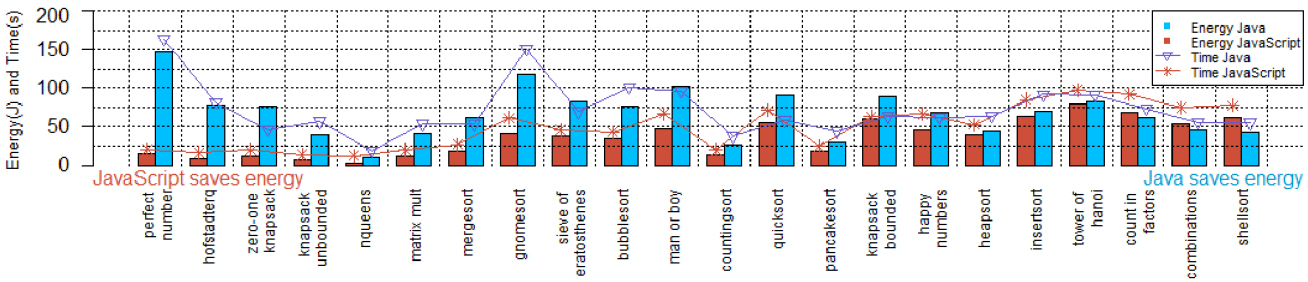

Fig. 1. Results of the benchmarks from Rosetta Code. The bars are sorted using the relative gain in energy consumption for each benchmark.

as monkeyrunner [33]. This approach would be particularly advantageous for apps with a stronger emphasis on user interaction.

All the executions of the apps were performed on the Samsung J7. Each experiment starts with the device fully charged and keeps running until either all the executions are over or the battery reaches 40% charge. Using this conservative threshold, we aim to guarantee that the device will never go into battery-saving mode.

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

1) Close all running applications not involved in the tests, activating airplane mode, and immediately rebooting the device. It is important to note that part of the automated solution for collecting the data uses the Wi-fi connection;
2) Connect the device to the computer using the Android Debug Bridge;
3) Reset all data regarding battery consumption;
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.
6) Gather the execution data from the device.

V. Study Results

This section presents the results of this study. Section V-A presents the results for RQ1 whereas Section V-B examines RQ2. All the data from the benchmarks and apps can be found at https://secaada.github.io/msr2017/

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

We separated the benchmark results in two parts. The first one analyzes the data collected from the benchmarks from Rosetta Code and the second one from TCLBG. For benchmarks with both sequential and parallel versions, we marked the parallel version with a 'p' at the end of the benchmark name, e.g., fasta and fastap.

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 of 1.52x more time to finish their execution. The figure shows that in 18 out of 22 benchmarks from Rosetta, the JavaScript versions consumed less energy and in 16 they exhibited lower execution time. 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 shows the execution time (lines) and energy consumption (bars) of the TCLBG benchmarks across all devices. The benchmark fastap consumed the lowest amount of energy in Java, with JavaScript, and C++ consuming a median of 1.72x and 3.33x more energy across all devices, respectively. The benchmark regexdnap was the most energy-efficient in JavaScript, with the other versions in Java and C++ consuming a median of 13.93x and 9.05x more energy across all devices, respectively. The benchmark revcomp was the most energy-efficient in C++, with Java and JavaScript consuming a median of 2.80x and 19.64x more energy across all devices, respectively. In spite of these differences between the development approaches, Figure 2 shows that their relationship does not differ much across devices, in terms of energy consumption. For example, for the spectral benchmark, Java consumed the most energy in all devices and JavaScript consumed the least, although the amount of energy consumed in each device was different. This can be observed for most of the benchmarks.

When comparing approach-to-approach the 50 benchmark-device pairs, contrasting JavaScript and Java, the former exhibited a lower energy consumption in 36 out of the 50 benchmark-device pairs and better performance in 19 out of the 50. When compared with C++, JavaScript exhibited lower energy consumption in 37 out of 50 and better performance in 25 out of the 50. In a direct comparison between Java and C++, the latter had a lower energy consumption in 28 out of the 50 benchmark-device pairs and a better performance in 27 out of the 50. These results suggest there is no overall winner in terms of performance, although JavaScript consumed less energy and had the worst performance on average across all devices.

To get a better understanding of which approach exhibits the best performance and energy consumption for each device, we analyzed the 50 benchmark-device pairs across all approaches. Performance-wise, Java had the best performance in 13 out of the 50 pairs, JavaScript in 18, and C++ in 19. Energy-wise, Java consumed less energy in 9 out of the 50 pairs, JavaScript in 31, and C++ in 10. Table III graphically summarizes these results. In this table, we use blue to indicate that Java won for that particular benchmark running on that particular device, red for JavaScript, and green for C++. Device names are

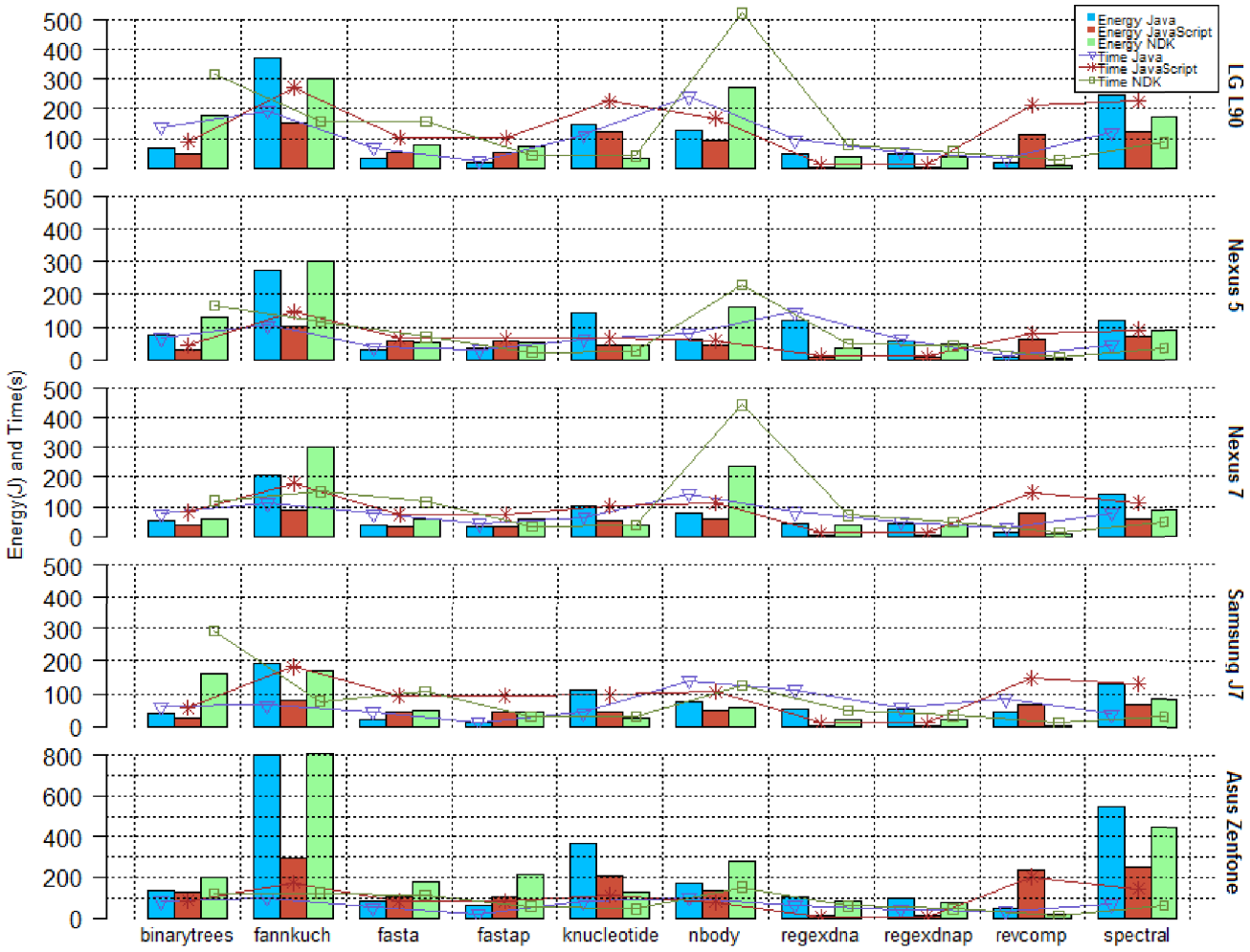

Fig. 2. Performance and energy results of the benchmarks from The Computer Language Benchmark Game (TCLBG) on all devices

abbreviated: L90 stands for LG L90, N5 for Nexus 5, N7 for Nexus 7 to N7, J7 for Samsung J7, and ZF for Zenfone. For example, JavaScript had the lowest energy consumption on the binarytrees benchmark on the LG L90 whereas C++ had the lowest energy consumption for the revcomp benchmark on the same device.

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

The results discussed in Section V-A suggest that the three approaches for app development in Android have different trade-offs in terms of energy consumption. However, Android applications are predominantly written in Java - in a random sample of 109 apps among the 1,600 apps in F-Droid, we had only 5 apps using JavaScript and 5 using NDK in any way. Nevertheless, 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 or C++. The major obstacle to this approach is that there is the overhead of cross-language invocations [34]. In this section we examine whether it is possible to compensate for this overhead so as to make existing apps more energy-efficient. It is worth noting that using two (or more) programming languages to develop an app could reduce its maintainability.

The four apps we have analyzed, anDOF, EnigmAndroid, BioNucleus, and TriRose have different behaviors. In TriRose, the delay of waiting for another development approach to perform the entire computation (the *Export* model – Section IV-A) is not acceptable, while it is for EnigmAndroid. On the other hand, the only model that makes sense for the EnigmAndroid is *Export*. For this case, the *Stepwise* model (Section IV-A) would increase the overhead with more cross-language invocations with no advantage to the user and the *Batch* model (Section IV-A) would be suboptimal, as we would have needed to pass smaller parts of the workload as arguments. For anDOF, the *Stepwise* model was the only realistic model. BioNucleus mainly focuses on IO inputs (from sensors and web data) and the information is presented to the user in real time, making Stepwise the only viable model. In summary, for anDOF and BioNucleus, we employed the *Stepwise* model, for Tri Rose, the *Stepwise* and *Batch* models, and for EnigmaAndroid, the *Export* model.

The specific workload for each app is determined in a way to

TABLE III
THE RIGHT-HAND SIDE PRESENTS THE DEVELOPMENT APPROACH WITH THE BEST RESULTS FOR ENERGY AND THE LEFT-HAND SIDE THE DEVELOPMENT APPROACH WITH THE BEST PERFORMANCE, FOR EACH DEVICE.

Energy	L90	N5	N7	J7	ZF
binarytrees	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript
fannkuch	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript
fasta	Java	Java	JavaScript	Java	Java
fastap	Java	Java	Java	Java	Java
knucleotide	C++	C++	C++	C++	C++
nbody	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript
regexdna	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript
regexdnap	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript
revcomp	C++	C++	C++	C++	C++
spectral	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript

Performance	L90	N5	N7	J7	ZF
binarytrees	JavaScript	JavaScript	Java	JavaScript	Java
fannkuch	C++	Java	Java	Java	Java
fasta	Java	Java	C++	Java	Java
fastap	Java	C++	C++	Java	Java
knucleotide	C++	C++	C++	C++	C++
nbody	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript
regexdna	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript
regexdnap	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript
revcomp	C++	C++	C++	C++	C++
spectral	C++	C++	C++	C++	C++

Legend: Java, JavaScript, C++

TABLE IV
RESULTS FOR THE MODIFIED APPS. STDEV STANDS FOR STANDARD DEVIATION.

App	Approach	Time(s)	Stdev	Energy(J)	Stdev
AnDOF Stepwise	Java	62.79	0.60	32.92	0.07
	Web	250.32	0.70	271.53	0.41
	NDK	6.26	0.17	0.37	0.03
BioNucleus Stepwise	Java	122.40	0.45	6.50	0.30
	Web	122.27	0.24	6.65	0.36
	NDK	122.30	0.18	6.55	0.44
Enigma Export	Java	89.21	2.46	38.13	0.11
	Web	27.78	0.89	12.22	0.06
	NDK	30.11	0.76	13.58	0.06
TriRose Stepwise	Java	27.84	0.03	4.03	0.02
	Web	53.85	0.05	7.78	0.01
	NDK	27.92	0.56	4.14	0.02
TriRose Batch	Java	26.74	0.02	3.95	0.02
	Web	27.49	0.03	4.21	0.02
	NDK	26.75	0.02	3.97	0.02

try to make each execution run in approximately 30s, keeping a low relative standard deviation as explained before. The workload for anDOF was 24×10^5 changes in a scroll that controls the depth of field. For each change, a method is called to recalculate it. The workload for Bionucleus consisted of 100 ecopoints, places where one can dispose the waste cooking oil. For EnigmAndroid we used a randomly generated String with $45,000$ characters. In TriRose, the workload for both *Stepwise* and *Batch* models consisted of drawing 1,500 lines on the screen, since the execution times were more similar.

Table IV presents the results. In two cases where we employed the *Stepwise* model on hybrid web apps, it degraded performance and boosted energy consumption. For example, the hybrid web version of TriRose using the *Stepwise* model took 92.87% longer to finish than the hybrid NDK version and consumed 87.92% more energy. This result suggests that unless it is possible to group parts of the work so as to minimize this overhead, building a hybrid web app will not save energy, due to the cost of invoking JavaScript code. On the hybrid NDK apps, we got significant improvements on performance and energy consumption using the *Stepwise* model in AnDOF and got almost identical results to the Java version in TriRose. Invocations to C++ snippets are done using the NDK and incur in minimal overhead for the app as in the cases above. Using the NDK to improve the application may be beneficial even if the developer needs to continually use cross-language functions.

Using the *Batch* model on TriRose, 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 11×10^4 positions. We applied this modification to the original native version, the hybrid web app, and the hybrid NDK app. With the *Batch* version of the app we got negligible improvements in performance and energy consumption in the native and hybrid NDK apps. Nonetheless, using the *Batch* model in the hybrid web app made it two times faster and cut its energy consumption almost in half, when compared to the corresponding *Stepwise* version. Furthermore, the difference in performance between the hybrid web app and the native and hybrid NDK apps using the *Batch* model was only 2.77%, with the hybrid web app consuming 6.04% more energy.

We only apply the *Export* to EnigmAndroid. 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. We also did not use it on anDOF because it creates a potentially unrealistic scenario. The depth of field needs to be informed to the user in real-time. Thus, the aggregated modifications may not be so useful. The results using the *Export* model represent a best case scenario, since the cross-language invocation overhead is almost entirely diluted. In the hybrid approaches, the app runs on Java but the cryptographic processing of the String is made on the other programming languages. In the *Export* model, both hybrid apps improved the performance and energy consumption of the app, with Java version consuming 3.21x more time and 3.12x more energy than the hybrid web app and 2.96x more time and 2.80x more energy than the hybrid NDK app. The hybrid web app was slightly better in performance and energy efficiency than the hybrid NDK app. The latter consumed 11.12% more energy and took 8.39% longer to finish.

Using the Stepwise approach on BioNucleus did not improve or deteriorate the performance or energy efficiency of the hybrid apps. That result suggests that hybridization of IO-

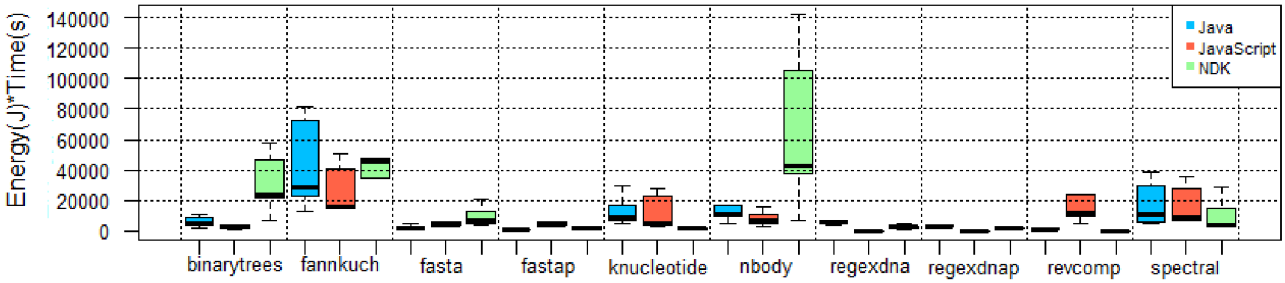

Fig. 3. Energy-delay product (EDP) results of the benchmarks from The Computer Language Benchmark Game (TCLBG) across all devices without outlines.

intensive apps may not result in improvements.

Even though these modifications promoted non-negligible improvements in performance and energy efficiency, they did not require large-scale modifications. For each app, JavaScript, and C++ files had, respectively: AnDOF, 160 and 192 LoC; BioNucleus, 16 and 65 LoC; EnigmAndroid, 173 and 281 LoC; TriRose, 100 and 166 LoC. Changes were relatively simple and represent less than 10% of the total lines of code of each app.

VI. Discussion

In this section we discuss some of the most important lessons learned from this study. We believe these lessons are useful for both researchers, who can investigate each one in more depth, and practitioners, who can leverage them for their app development activities.

There is no overall winner. Analyzing the benchmarks, there was no better approach in all scenarios. It is possible to state that, in general, JavaScript consumed less energy than Java and C++. It had a lower average energy consumption per benchmark and the JavaScript versions of most benchmarks exhibited the lowest energy consumption when compared to Java and, where applicable, C++ versions. Nevertheless, this result was not universal. For example, in the revcomp benchmark, the JavaScript versions had by far the highest energy consumption. In addition, for performance, specially when we look only at the TCLBG benchmarks and the four apps, the results were much more mixed, with Java and C++ versions outperforming JavaScript in most cases.

Both energy and performance are important in practice and most developers would be interested in balancing the two. Thus, we choose to look at the energy-delay product (EDP) [35], the product of the execution time and the energy consumed by a benchmark. This measure highlights the trade-off between energy and execution time (the lower the better). The boxplot in Figure 3 shows the EDP for all benchmarks across all devices. It highlights that some benchmarks perform better in a certain language, e.g., fasta and revcomp in Java and regexdna and regexdnap in JavaScript, whereas others perform worse, e.g., nbody and binarytrees in C++ and revcomp in JavaScript. Furthermore, for some benchmarks the difference between the languages is unclear (fannkuch and spectral). Even though JavaScript usually has lower energy consumption per benchmark, the benefit of running the benchmark faster may overshadow this when looking at an aggregated metric. When considering all the 50 benchmark-device pairs, JavaScript presented a better EDP on 32 out of 50 pairs when compared to Java, and 29 out of 50 when compared with C++. When comparing Java and C++, the latter had a better EDP on 27 out of 50 pairs. This result suggests that even though there is no overall winner, JavaScript exhibits a good trade-off between energy usage and performance, besides consuming less energy in most cases.

These results show that, in certain scenarios, the different approaches may have significant differences in performance and energy consumption. Having that information, developers can try to improve an app in two ways: (i) by experimenting with different approaches, depending on application requirements; and (ii) by exploring combinations of these approaches and building a hybrid app.

The development approaches differ little across devices, in terms of energy usage. For the set of benchmarks, apps, and devices we analyzed, our measurements indicate that a development approach that consumes less energy in one device is likely to do so in other devices as well. A quick look at the bars of Figure 2 or to the Table III highlights this. This finding has a clear practical implication. Even though there are dozens of Android smartphone models in the market, it suggests that optimizing energy consumption for a small and diverse subgroup of these models may be sufficient.

Faster != Greener. The main reason for the good performance of some of the Java and C++ benchmarks is parallelism. While all the JavaScript versions run sequentially, the Java and C++ versions of 5 benchmarks (fannkuch, fastap, knucleotide, regexdnap, and spectral) are capable of leveraging multicore processors to improve performance. All but one (regexdnap) of these versions outperform the corresponding JavaScript version. However, all these benchmarks, with the exception of fastap for Java and knucleotide for C++, consumed more energy. This is consistent with previous work [24], [25] that found out that, for programs capable of benefiting from multi-core processors, performance is often not a proxy for energy consumption. Complementarily, even when a benchmark does not leverage parallelism, the relationship between energy and execution time is unclear. For example, none of the versions of the binarytrees benchmark exploits parallelism. Notwithstanding, Java exhibited the best execution time while JavaScript exhibited the lowest energy consumption.

Parallelism may be a good option anyway. Among the

TABLE V
AVERAGE DATA FROM THE DIVISION BETWEEN THE VALUES FOR THE SEQUENTIAL AND PARALLEL VERSIONS OF THE FASTA AND REGEXDNA BENCHMARKS. A VALUE GREATER THAN 1 IN A CELL MEANS THAT THE SEQUENTIAL VERSION HAD HIGHER EXECUTION TIME, ENERGY CONSUMPTION, OR EDP.

Benchmark	Language	Energy	Speedup	EDP
regexdna	Java	1.20	1.86	2.33
	C++	0.95	1.39	1.32
fasta	Java	1.37	2.52	3.70
	C++	1.02	3.21	3.35

selected benchmarks, five (fannkuch, fastap, spectral, regexdnap and knucleotide) make heavy usage of parallelism. Two of those benchmarks also had a sequential version (regexdna and fasta) that we executed on each device and development approach. We compared the results of these two benchmarks with their parallel versions, and the results are summarized in Table V. From the data, we can see that parallelism is usually better for energy efficiency and performance in Java for regexdna, having a slightly higher energy consumption than the sequential version in C++. The parallel version of fasta in Java exhibited lower energy consumption while the parallel version in C++ had the best speedup. For all benchmarks, the EDP is better using the parallel version. In these cases, using parallelism is better both for energy consumption and performance, and may be considered an improvement.

NDK is a safer bet to improve performance. Analyzing the modified apps, we noticed that using the hybrid NDK approach improved the application performance in two cases (EnigmAndroid and anDOF). When considering the TCLBG benchmarks, C++ exhibited the best performance for most of them (Table III). Therefore, if performance is of utmost importance and energy is a minor concern, using the NDK can be considered a safer bet. It is interesting to note that the NDK exhibited good performance even in hybrid apps using the *Stepwise* model.

VII. THREATS TO VALIDITY

The accuracy of the Google tools for measuring energy have yet to be assessed. However, the most important data is the comparison and relation between the performance and energy consumption for each language, and not their raw values. In addition, preliminary experiments [36] comparing Android's tools for measuring energy consumption with the measurements of an oscilloscope (coarse-grained measurement) indicate that Android's tools are accurate.

We observed that the results we obtained for the performance of the TCLBG benchmarks differ considerably from those reported on the original website. The difference exists both in terms of the absolute values of the measurements and, in some cases, in terms of which development approach exhibits the best performance. This could be an indication that we committed errors in the design, implementation, or execution of our experiments. It could also indicate that the performance of different development approaches in more powerful machines such as desktops and laptops is not a good proxy for the performance of these approaches in a smartphone. To confirm this, we executed all the versions of the TCLBG benchmarks on a desktop computer (Processor Intel i7-4700HQ, 24GB RAM memory using bash on Ubuntu on Windows). Unlike the results we obtained for the mobile devices, the results obtained in this machine were consistent with the ones reported at the TCLBG website.

Benchmarks do not represent the behavior of an application using JavaScript as the main programming language [8], 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 performance gain is measured, by isolating usage pattern behavior. The modifications made in the apps show that even small changes could lead to a non-negligible improvements in energy efficiency.

Some of the devices we employed were not in the same version of Android and that may influence the results. We had some preliminary results comparing an earlier version on Zenfone (5.0) with the current version (6.0.1). Some benchmarks executed faster and used less energy and some executed slower and consumed more energy (the Java version of knucleotide consumed 1.3 times more time and energy in version 5.0 and nqueens took 3 times longer to execute in version 6.0.1). However, the relationships between the development approaches did not change across versions. Furthermore, most devices were in the same version (5.1) and our data shows that the relation between the languages remained roughly the same even across Android versions.

VIII. CONCLUSION

This paper aimed to investigate 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, using Java, Javascript, and C++. We used five devices with different characteristics on two main versions of Android. We found out that there is no clear winner among the development approaches in Android. Each one has its advantages in some scenarios. Our results indicate that JavaScript saves more energy and is slower than the other approaches for benchmarks and that app hybridization may be a solution for app optimization, both in performance and energy consumption. Furthermore, for the set of benchmarks, apps, and devices we analyzed, the relationships between the development approaches remained stable for most benchmarks on every device tested. Among the two options for hybridization, using the NDK is a safer bet for improving performance, but using a web-based approach may have a better outcome when there are few cross-language invocations. The data from this study can be found at https://secaada.github.io/msr2017/

Acknowledgments. We would like to thank the anonymous reviewers for helping to improve this paper. This research was partially funded by CNPq/Brazil (304755/2014-1, 406308/2016-0, 465614/2014-0) and FACEPE/Brazil (APQ-0839-1.03/14, 0388-1.03/14).

REFERENCES

[1] Smartphone battery survey: Battery life considered important. https://aytm.com/blog/daily-survey-results/smartphone-battery-survey/. Accessed: 2017-01-07.

[2] Smartphone os market share. http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Accessed: 2017-03-25.

[3] Number of apps in leading app stores. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Accessed: 2017-03-25.

[4] Google android ndk website. http://developer.android.com/ndk/guides/index.html. Accessed: 2017-01-07.

[5] Rosetta code website. http://rosettacode.org/wiki/Rosetta_Code. Accessed: 2017-03-25.

[6] The computer language benchmark game. http://benchmarksgame.alioth.debian.org. Accessed: 2017-03-25.

[7] Google android developer website. https://source.android.com/devices/tech/power/batterystats.html. Accessed: 2017-02-08.

[8] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 3–3, 2010.

[9] F-droid repository website. http://f-droid.org. Accessed: 2017-03-25.

[10] Instituto nacional de ciłncia e tecnologia para engenharia de software repository. https://github.com/ines-escin/. Accessed: 2017-03-25.

[11] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2:437–445, 1994.

[12] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: a hardware based mining software repositories software energy consumption framework. In *11th Working Conference on Mining Software Repositories*, pages 12–21, May/June 2014.

[13] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, New York, NY, USA, 2013. ACM.

[14] A Hindle. Green mining: A methodology of relating software change to power consumption. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 78–87, June 2012.

[15] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, 2014.

[16] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 345–360, 2014.

[17] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 831–850, 2012.

[18] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[19] Marco Couto, Tiago Carção, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Detecting anomalous energy consumption in android applications. In *Programming Languages - 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings*, pages 77–91, 2014.

[20] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[21] Claas Wilke, Christian Piechnick, Sebastian Richly, Georg Püschel, Sebastian Götz, and Uwe Assmann. Comparing mobile applications' energy consumption. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1177–1179, New York, NY, USA, 2013. ACM.

[22] 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 *Proceedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering*, Klagenfurt, Austria, February 2017.

[23] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80, August 1984.

[24] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 517–528, March 2016.

[25] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. *SIGPLAN Not.*, 49(10):345–360, October 2014.

[26] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 225–236, Austin, USA, May 2016.

[27] Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. A comprehensive study on the energy efficiency of java's thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution*, pages 20–31, Raleigh, USA, October 2016.

[28] Andre Charland and Brian Leroux. Mobile application development: Web vs. native. *Commun. ACM*, 54(5):49–53, May 2011.

[29] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in rosetta code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 778–788, Piscataway, NJ, USA, 2015. IEEE Press.

[30] Wellington Oliveira, Weslley Torres, Fernando Castor, and Bianca H Ximenes. Native or web? a preliminary study on the energy consumption of android development models. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 589–593. IEEE, 2016.

[31] Ibm. swot analysis: Hybrid versus native development in ibm worklight. https://www.Ibm.com/developerworks/community/blogs/mobileblog/entry/swot_analysis_hybrid_versus_native_development_in_ibm_worklight?lang=en. Accessed: 2017-01-07.

[32] Rubén Saborido, Venera Venera Arnaoudova, Giovanni Beltrame, Foutse Khomh, and Giuliano Antoniol. On the impact of sampling frequency on software energy measurements. Technical report, PeerJ PrePrints, 2015.

[33] Monkeyrunner website. https://developer.android.com/studio/test/monkeyrunner/index.html. Accessed: 2017-03-20.

[34] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. An efficient native function interface for java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 35–44. ACM, 2013.

[35] James H. Laros III, Kevin Pedretti, Suzanne M. Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. *Energy Delay Product*, pages 51–55. Springer London, London, 2013.

[36] Quang-Huy Nguyen and Falko Dressler. The accuracy of android energy measurements for offloading computational expensive tasks: Poster. In *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '16, pages 393–394. ACM, June 2016.