Towards using Memoization for Saving Energy in Android *

Rui Rua[13], Marco Couto[13], Adriano Pinto[24],
Jácome Cunha[14], and João Saraiva[14]

[1] Universidade do Minho, Portugal
[2] Universidade NOVA de Lisboa, Portugal
[3] HASLab/INESC TEC, Portugal
[4] NOVA LINCS, Portugal
{rui.a.rua,marco.l.couto}@inesctec.pt ab.pinto@campus.fct.unl.pt
{jacome,saraiva}@di.uminho.pt

Abstract. Over the last few years, the interest in the analysis of the energy consumption of Android applications has been increasing significantly. Indeed, there are a considerable number of studies which aim at analyzing the energy consumption in the Android ecosystem, such as measuring/estimating the energy consumed by an application or block of code, or even detecting energy expensive coding patterns or APIs.
In this paper, we present an initial study of the impact of memoization in the energy consumption of Android applications. We compare implementations of 18 methods from different applications, with and without using memoization, and measure the energy consumption of both of them. The results show that using memoization can be a good approach for saving energy, since 13 of those methods decreased their energy consumption.

Keywords: Android · Energy · Memoization

1 Introduction

The Android ecosystem is evolving at an impressive pace. Since this system can run on a wide variety of devices, from smartphones to tablets of several manufacturers, to wearables, its widespread usage in the last decade is notorious.

This proliferation of Android devices increased the interest in a particular research area: energy consumption analysis of software applications. It has been an intensive research topic in the last few years, greatly motivated not only by the mobile development area, characterized by powerful computing systems, yet energy-harvesting, which run over batteries with limited capacity, but also by the growing interest of developers in knowing more about how to develop software in a energy-saving manner [24].

* Work financed by the ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia, project POCI-01-0145-FEDER-016718.

Several research works focused on detecting or predicting the energy consumption that is triggered by a software artifact [11, 12]. For instance, some works presented techniques for monitoring and classifying energy consumption of blocks of code, such as lines of code [18], methods [7], API calls [20–23], or code patterns [17, 27]. Even the energy consumed at the testing phase can be a concern [19], as well as the display of visual elements in the application views [8].

Nevertheless, the amount of information provided about an application that can be used by developers to reduce its energy consumption at the development phase, is still very low [24]. In this paper we study the influence of incremental computation, obtained via method memoization, on the energy consumption in the context of the Android ecosystem. We present a systematic study that shows the energy gains of applying memoization techniques to Android applications. Memoization is widely used in the context of functional programming to speedup the execution time of programs: calls of a function are cached/memoized, and next calls to that function with the same arguments are obtained from the cache, so that the function is not call again. In order to be able to perform method memoization we consider side-effect free methods, only. That is, methods that implement functions and as consequence are prone to memoization [30]. After computing such methods (i.e., functions) in the application's source code, we refactor it into a semantically equivalent one which memoize/reuses its calls. To study the impact in performance of our memoization refactoring, we monitor the energy consumption while executing the non-incremental and the incremental versions of several Android applications with the same inputs. Our first experimental results show that memoization can greatly improve the energy efficiency of an Android application, considering that such application is prone to it, without threatening its normal functioning and/or efficiency.

To summarize, we intend to answer to the following research question: *Can memoization help reduce energy consumed by Android applications?*

This paper is organized as follows. Section 2 introduces the concept of memoization for Android applications and describes in detail our experimental setting. We present the results of the experiment in Section 3, which are discussed in Section 4, where we also answer our research question. Section 5 presents the threats to the validity of this work. Related approaches are describeded in Section 6. Finally, Section 7 presents our conclusions and future work directions.

2 Experimental Study

In this section, we start by explaining how to decide if a Java/Android method can be memoized. We then describe in detail our memoization technique for Android applications, along with the empirical study used for validation.

2.1 What Can be Memoized?

Yang et al. [29] proposed a set of three pre-conditions to classify a Java method as memoizable:

1. The method must be pure, that is, it must be a function;
2. The arguments must be immutable;
3. The return value cannot depend on static fields, public member fields nor publicly exposed member fields.

In our particular case, we have simplified the application of pre-condition (2) and used only methods that have as arguments primitive types or the *Context* object of an application (since it is immutable).

In the next section we will describe in detail the experimental study we conducted: the methods analyzed, the conditions under which such methods were tested, and the full procedure to actually run the tests and obtain results.

2.2 Experimental Setup

This section describes the study conducted, describing the Android applications and methods used, how methods were refactored, and the experiment setup.

Android applications and methods analysis: We used two Android applications from the MUSE repository[5] (`Pixate Freestyle`[6] and `android-demos`) and another one from F-Droid[7] (`Chanu`[8]). The distribution of methods analyzed from each of these applications is described in Table 1.

`Pixate Freestyle` is a free framework that lets users style his native Android views with stylesheets and is very much based on the graphical component. This application contains 219 classes. `android-demos` is a very specific application from the repository and contains 34 classes. `Chanu` is a well-known application with 538 classes. Basically, it is the code of the 4chan site application where users can browse images of various contents. This is the largest application among the three and so it was where we most applied the technique of memoization.

Since our goal is to test each method prone to memoization individually, we decided that it was best suited to extract such methods from these application and run our experiments in a controlled manner. In order to do so, we created our own application only with such methods which we literally copied from their applications. After this process, we duplicated the application, and in the second version all the 18 methods were using our memoization technique.

Methods Refactoring: We manually analyzed the methods for the three selected applications. After finding memoizable methods, we created an application to encompass all those methods. We have then created two versions of the same application: one with all methods in their original form and the other with the memoized ones.

Our memoization technique consisted of storing the input of the methods as a key in a `HashMap` and its output as the value. So, if the method had already

[5] https://opencatalog.darpa.mil/MUSE.html.
[6] https://github.com/Pixate/pixate-freestyle-android
[7] https://f-droid.org
[8] https://f-droid.org/en/packages/com.chanapps.four.activity

Method	Input	Output	Application
createIntent	Context,String,String	Intent	Chanu
countLines	String	int	Chanu
replyText	Long[]	String	Chanu
join	List<String>, String	String	Chanu
threadSubject	String, String	String	Chanu
textViewFilter	String, boolean	String	Chanu
getUrl	Context, String	String	Chanu
planifyText, quoteText, exifText	String	String	Chanu
getNumeral	String, String	String	Pixate Freestyle
removeLocaleInfoFromFloat, addNegativeSign, addPositiveSign	String	String	Pixate Freestyle
isMobile	String	boolean	android-demos
readableFileSize	long	String	android-demos
dip2px; px2dip	Context,float	int	android-demos

Table 1: Characterization of methods used in experiments.

been called for a particular input, then we just accessed its value on the map and returned it. Otherwise, a new entry was created on the map and the result returned. In cases where the methods only received a single input parameter, this was directly the key of the map. However, there were cases in which more than one argument was passed as parameter. Table 1 presents a characterization of the methods used in our experiments, containing for each method, the respective name, application, input and output parameters. In this table, some of such examples can be observed. For instance, the method name *getURL* from the Chanu application receives two distinct objects as an argument. For the multi-parameter cases, a library called javatuples [29] was used. In this way, the input values were saved in tuples which became the `HashMap` key.

Experiments execution: In order to test the energy impact of memoization in these methods, we defined a usage scenario for them, consisting of invoking a method 50 times with different parameters passed to it. This resulted in inserting 50 entries in the method's map structure. To take advantage of memoization, this procedure was repeated 10, 20, 30, 40, and 50 times, and each of this repetition sets are what we consider our *test cases*.

To obtain a representative number of measurements, each test case was repeated 25 times for each method in both the original and the memoized versions (alternatively), using a Nexus 4 running Android version 5.1.1 - API level 22. In order to perform energy monitoring, we used the Trepn Profiler [9], an accurate Energy profiler developed by Qualcomm. Before running each iteration, we performed the initial set up, where the previously installed version of the application under test was removed, the new version was installed and switched on (as well as the Trepn app), and a warm-up of 5 seconds was considered. After this set up, Trepn started profiling at the same time that the test was executed. When the test finished, the energy results were recorded and both the app and Trepn were switched off.

[9] https://developer.qualcomm.com/software/trepn-power-profiler

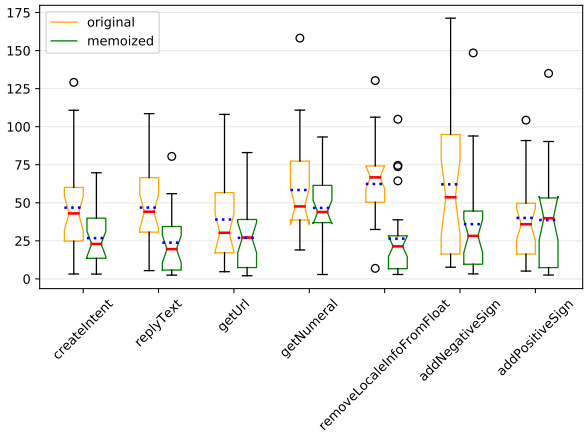

Fig. 1: Methods favoring memoization with consumption between 0 and 175 mJ.

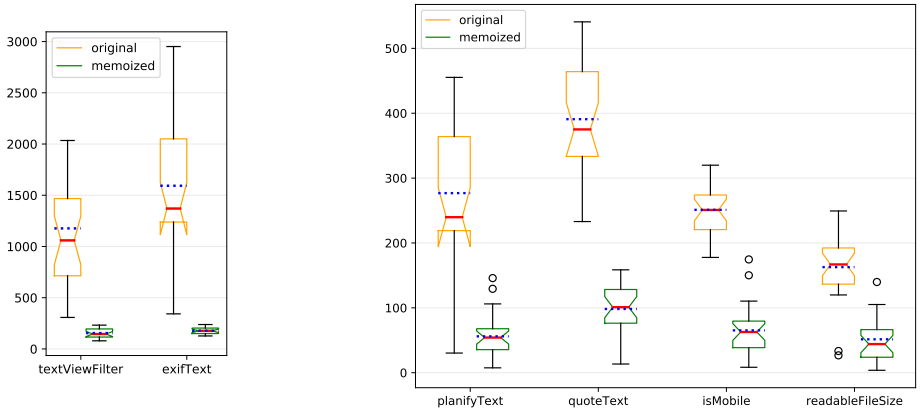

Fig. 2: Methods in favor of memoization with consumption between 0 and 3000 mJ (left hand-side chart) and between 0 and 600 mJ (right hand-side chart).

3 Results

In this section we present in detail the results obtained from the experiment described in the previous section. Figures 1 and 2 display a box plot for each of the 13 methods we could find a statistically significant difference between the energy consumption of the original method and the energy consumption of the memoized one in favor of the memoization, that is, when the memoized method consumed less energy than the original version[10].

[10] We divided the methods in different charts because they have different scales of values and to have them all in the same one would not allow to properly see the details.

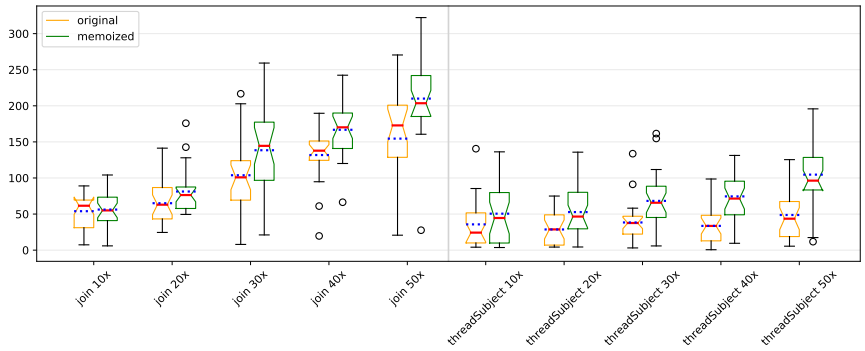

Fig. 3: Box plots representing the energy spent by the original and memoized methods, considering 10-50 executions of the tests, where the original version spent less energy.

Each box plot represents the energy measurements, in millijoules (mJ), of the 25 executions of the corresponding method showing the minimum and maximum values (marked by the whiskers), possible outliers (marked by small circles), the 1st and 3rd quartiles (bottom and top of the box), the median (marked by the red horizontal filled in line), and the average (marked by the blue horizontal dashed line). Each yellow/green pair represents the original/memoized version of the method. We also show the notch because it can be seen as an informal test of the null hypothesis that the medians are equal, that is, if two notches overlap, then it is not possible to reject the null hypothesis with a 95% confidence [4]. In Section 4 we will present formal verification of the statistical differences of the energy measurements.

Note these charts show the energy values for 10 executions of the test suite, the first being with new values, and the remaining with repeated ones. In Section 4 we will detail more the analysis of different numbers of executions.

In Figure 3 we show the box plots with the energy consumption of the only two methods where the energy consumed by the memoized versions is statistically different from the original methods, but in favor of the original ones. Once more, we refer to Section 4 to test executions greater than 10 times.

Figure 4 presents once more box plots, but now for the three methods where no statistical difference was found for original and memoized methods. For these three methods we executed the same test suite, but now running it 10, 20, 30, 40, and 50 times. The goal was to take more advantage of the memoization and understand if this would turn the method better or worst the original one. Each increment of 10 executions only makes the memoized method read the stored values, while the original version needs to run entirely. Indeed it may be the case the overhead of the memoization (another method call, plus the map) makes the gain negligible. However, what we recorded for these methods is that sometimes the memoized version is better, but sometimes the original one also is. We will come back to this discussion in Section 4.

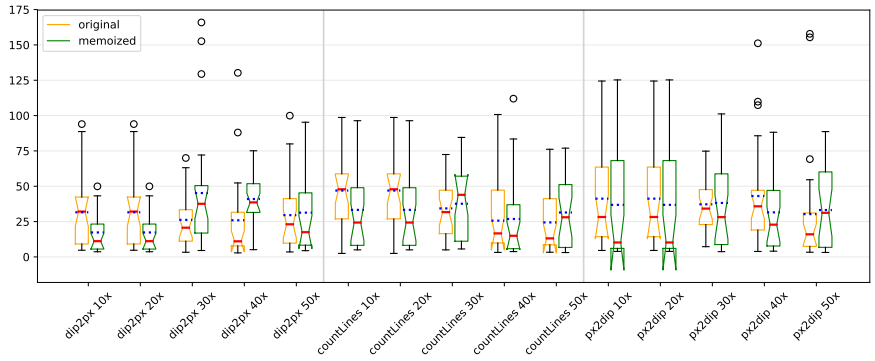

Fig. 4: Box plots representing the energy spent by the original and memoized methods, considering 10 to 50 executions of the tests with no statistical difference.

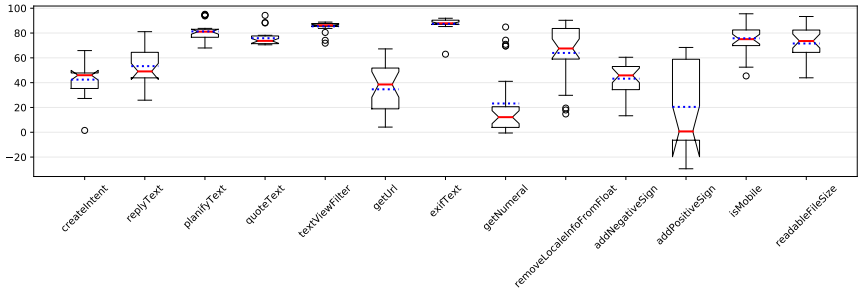

Fig. 5: Energy losses from the original method to the memoized one of the methods that have positive results.

The results presented until now seem to indicate that there is a positive effect in the energy consumption when memoizing method calls. In Figure 5 we present the percentage of losses in the energy spent from the original version of the method to the memoized one, considering the methods where memoization produces positive results (the methods represented in Figures 1 and 2). For each method, we calculated the percentage of energy decrease as follows. For each of the 25 executions, we sorted, independently, the original and the memoized values of energy consumption. We then calculated the losses of energy from the use of the original version to the memoized one using the formula $\frac{original - memoized}{original} \times 100$. Positive values represent the fact that there is indeed a loss of energy from the original to the memoized, thus supporting the use of memoization, and negative values imply an increase in the energy spent, thus pointing situations where memoization does not consumes less energy.

As we can see, the gains are fairly positive. Note once more we are only showing the results for running the test suite 10 times, as this already points to quite positive results. We will discuss these gains further on in the Section 4.

method	10x	20x	30x	40x	50x
exifText	(87, 90)	(92, 95)			
textViewFilter	(85, 88)	(94, 94)			
planifyText	(77, 83)	(86, 89)			
quoteText	(71, 78)	(88, 92)			
isMobile	(70, 82)	(84, 96)			
readableFileSize	(64, 82)	(82, 89)			
removeLocaleInfoFromFloat	(59, 84)	(66, 71)			
replyText	(44, 64)	(-9, 17)	(41, 76)	(41, 71)	(48, 76)
createIntent	(35, 48)	(-21, 6)	(35, 54)	(21, 43)	(49, 66)
addNegativeSign	(34, 53)	(62, 80)	(61, 75)	(68, 94)	(71, 86)
getUrl	(19, 52)	(32, 66)	(45, 57)	(37, 56)	(49, 62)
getNumeral	(4, 21)	(-21, 2)	(9, 30)	(27, 45)	(60, 75)
addPositiveSign	(-6, 59)	(40, 48)	(51, 65)	(36, 57)	(10, 26)
join	(-11, 11)	(-40, -8)	(-46, -18)	(-30, -22)	(-36, -19)
px2dip	(-55, 0)	(2, 60)	(-27, 53)	(4, 52)	(-82, 5)
threadSubject	(-62, 1)	(-99, -54)	(-92, -75)	(-280, -101)	(-219, -102)
countLines	(-84, -7)	(16, 62)	(-17, 6)	(-15, 21)	(-53, 16)
dip2px	(-161, -60)	(24, 47)	(-71, -35)	(-281, -62)	(-21, 17)

Table 2: 1st (on the left hand-side of the column) and 3rd (on the right) quartiles of the energy losses from the use of the original method to the use of the memoized version.

To increase the reading of these results we present in Table 2 the percentage of the energy not spent, comparing the memoized method to the original one of the 1st and 3rd quartiles. This follows from the box plots, complementing the information and giving central values of the losses.

Finally, Table 3 shows the percentage of times the energy spent by the memoized method is lower than the original one, considering the 25 runs of each method, for the 5 test cases. Note the underlined values are the ones where we could not find statistical significant difference between the original and memoized measurements. For all the other methods there is statistical significant difference between the energy consumption of the original and the memoized method. Values in **bold** indicate when more than 50% of the 25 runs spend more energy in the memoized methods than the original ones. The values with no special formatting have statistical significance in favor of the memoized versions.

4 Discussion

The first observation is the most obvious one: memoization has a clear impact on energy consumption in Android applications. In fact, for the majority of situations the impact is fairly positive. Considering the 18 tested methods, the energy consumption for 13 of them has consistently decreased when using memoization (as can be seen in Figures 1 and 2). For those methods, we can observe that the average consumption of the 25 measurements is always lower when using memoization, as well as the minimum and maximum values, and the values for the 1st and 3rd quartile. Figure 2 shows the methods that had the biggest impact.

These values allow us to answer to the research question: can memoization be used to reduce energy consumption in Android applications?

	10x	**20x**	**30x**	**40x**	**50x**
createIntent	72	56	68	68	72
getUrl	60	72	68	76	76
replyText	68	52	84	92	84
getNumeral	56	56	64	68	84
addNegativeSign	56	88	84	100	100
addPositiveSign	56	72	80	84	80
planifyText	100	100	-	-	-
quoteText	100	100	-	-	-
textViewFilter	100	100	-	-	-
exifText	100	100	-	-	-
removeLocaleInfoFromFloat	80	88	-	-	-
isMobile	100	100	-	-	-
ReadableFileSize	88	100	-	-	-
join	48	**32**	28	**28**	**24**
threadSubject	**40**	36	**20**	**12**	**20**
countLines	**36**	68	48	**44**	44
dip2px	**24**	68	**32**	**28**	56
px2dip	**40**	56	48	64	**44**

Table 3: Percentage (of the 25 runs) that the memoized version consumes less energy than the original one (For each method and for each test setting). Underlined values have statistical significance; bold values represent more than 50% of the 25 runs spend more energy in the memoized methods than the original ones.

To make these results more meaningful, we want to know if there is statistical evidence behind these observations, that is, if the energy consumed by the memoized version of these methods is consistently lower than the original one. Thus, we tested the following hypothesis:

$$\boldsymbol{H}_0 : P(A > B) = 0.5$$
$$\boldsymbol{H}_1 : P(A > B) \neq 0.5$$

where B and A represent the act of randomly drawing a value from the set of 25 measurements, with and without using memoization, respectively. Hence, $P(A > B)$ represents, when drawing from both A and B, the probability of getting a value from A (without memoization) larger than the one drawn from B (using memoization). Our null hypothesis is then obtaining a probability of 50%, while obtaining one different than 50% is the alternative hypothesis.

To understand if there is an overall significant relevance between the distributions of A and B, we ran the Wilcoxon signed-rank test, with a two-tail p-value considering $\alpha = 0.01$[11]. The test was repeated for the 13 methods where the use of memoization led to improvements in energy consumption. At the end, the test produced significant relevance, with the p-value < 0.01 for 11 of the 13 cases. The exceptions where `getUrl` and `addPositiveSign`, both with parametric distributions, which we will exclude for now and explain later. To calculate a non-parametric effect size, Field [10] suggests using Rosenthal's formula [25, 26]

[11] All the values obtained for this test, for the 18 methods, are available at https://goo.gl/WHXQeA.

to compute a correlation, and compare the correlation values against Cohen's [5] suggested thresholds: 0.1 for small, 0.3 for medium and 0.5 for large significance.

From the 11 scenarios, 8 were non-parametric, and the values obtained were: 0.4 (medium) for `createIntent` and `replyText` methods, and 0.6 (large) for the remaining methods. For the remaining 3 methods, a D'Agostino and Pearson's test [9] revealed that we were dealing with normal distributions, and so we should calculate the Cohen's d coefficient to determine the magnitude of the effect size. According to Sawilowsky [28], the reference thresholds and their respective effect size should be 0.01 (very small), 0.2 (small), 0.5 (medium), 0.8 (large), 1.2 (very large), and 2 (huge significance).

We obtained values of 0.4 for the `getNumeral`, 0.6 for `addNegativeSign`, and 1.4 for the method `removeLocaleInfoFromFloat`. Considering these reference values, we have statistical support to say that, for these methods, using memoization does lead to energy savings.

The effect size values calculated so far were only for the scenario where each method was called 10 times in each measured test. It may happen that, for methods with lower significance values, repeating the experience with an increasing number of calls could lead to results more or less supportive. Thus, we repeated the experience with 20, 30, 40 and 50 invocations, calculated the significance values and the effect size (see Table 3). We observe that all 13 aforementioned methods continue to have more than 50% of the 25 tests in favor of memoization and, although in some specific cases the statistical experiment resulted in no significance, the percentage of memoization favorable tests kept increasing with the increasing number of calls for all of them, and the same happened with the significance. Such cases were the previously excluded `getUrl` and `addPositiveSign` method. Hence, we categorized these 13 methods as *prone to memoization*.

The use of memoization, however, does not always mean saving energy. In some cases, it is not possible to determine whether memoization results in energy saving or not. In our experiments, we found 3 methods fitting such criteria. The results of such experiments are shown in Figure 4, in which the methods where called 10 times on each test. As we can see by examining the box plots, the values for the original version and the version using memoization are very similar.

By running the same statistical experiment performed for the 13 methods were memoization actually decreased energy consumption, we observed that the results for `dip2px`, `px2dip`, and `countLines` methods were not statistically significant. This means that the reason behind the energy consumption being lower is not related to the use of memoization, neither to the use of the original version of the method. Similarly, the percentage of tests in favor of memoization was mostly around 50%, dropping or increasing a few percentage levels unpredictably when increasing the number of invocations. Therefore, we categorized these methods as *unpredictable*.

For the remaining 2 methods (`threadSubject` and `join`), we observed that the energy consumption actually increased in the majority of cases while using memoization. In fact, if we examine Table 3, we see that for such methods the percentage of times where the memoized version consumes less energy is always

lower than 50%, for all test scenarios. We ran the same statistical test as before to check if the values were consistently worse for the memoized version, and we obtained significant relevance for all of them, except `px2dip`. However, the effect size varied from "small significance" to "medium significance". Thus, these methods are then categorized as *unfit for memoization*.

The most interesting observations can, however, be seen in Table 2. The data presented there shows the calculated gains (positive values) or losses (negative values) when using memoization. To obtain these values, we first sorted, for each method, the energy consumed by the original and memoized version, in order to compare the lower/higher energy consumption values of one version with the lower/higher values of the other. Then, we calculated the gains pairwise and arranged them in a box plot. The first element of the pair in each table cell is the gain calculated for the 1st quartile of the box plots, while the second element is the gain considering the 3rd quartile. With this, we try to show the gains/losses are independent of the energy consumption measured absolute value. Each column contains the gains/loses for the obtained results of running the same experiment, but varying the number of times each method is invoked.

As expected, we see that the majority of the values are positive, that is, memoization is, in the majority of cases, a suitable technique to save energy. In the first 7 methods, the impact is more notorious since in all the 25 measurements the memoized version has a significantly lower energy consumption, and when we repeated the experiment for 20 invocations per test, the impact is even greater. Also, the significance values kept increasing, so we stopped measuring and categorized them as *strongly prone to memoization* methods.

If we look only to the pairs with both positive gains in the column for 10 invocations (methods *prone to memoization*), the gains can go from 3% to 90%. Theses values tend to increase if we increase the number of invocations: for these same methods, considering 20 invocations, if we exclude the ones with negative values, since they were the cases with no statistical significance (see Table 3), they go from 31% to 96%. For the other scenarios (30, 40 or 50 invocations), the values are always positive, as the significance keeps being maintained.

For the *unpredictable* methods, the gains are also unpredictable: the 3rd quartile value is negative (loss), but the 1st is positive (gain). Even so, in that scenario the statistical test showed there was no significance for both the methods, as we increase the number of invocations and tend to decrease proportionally.

5 Threats to Validity

Measuring the energy consumption of a mobile device is complex [3]. This is mostly due to the fact that it is quite difficult to fully isolate the code or application under measurement. To address this issue, we executed our application 25 times, thus giving it enough slack to have in average results that correspond to the truth and with as low side effects as possible. Moreover, we executed the application in a factory-reseted device, in airplane mode and with the lowest brightness level, to ensure the energy consumed by the display was as low as

possible. Because, our memoization reuses the results of previously computed (memoized) method calls, thus, we reduce the computations/work needed by a program. We focus our evaluation in measuring the energy savings of the CPU. However, a detail study on how the energy of the screen and communication mechanisms (two known energy greedy aspects in mobile devices) are also affected by memoization is needed.

To assure the data retrieved were consistent, both the memoized and non-memoized applications were run in the same environment, half of the times starting with the memoized version, and the other half with the non-memoized one. Thus, this gives us confidence the differences we found between the two versions are only due to the usage or not of memoization.

The energy consumption values were obtained by using Trepn, a tool developed by Qualcomm, a manufacturer of processors and chipsets of mobile devices. As Trepn obtains the power values directly from the PMIC (Power Management Integrated Circuit) present in the chipsets, the values obtained through this tool are accurate [14], and has been used in several works [13, 15, 16].

Our experimentation occurred only in one smartphone and one version of the operating system. It is expected to find different absolute values in different smartphones and Android versions, but the differences between the memoized versions and original ones are quite consistent and thus it is not expected to see significant changes in the differences if changing the evaluation settings.

Another possible issue is the executed tests. Indeed, it is usually difficult to find tests for Android applications [6–8]. Thus, we created a test suite for each method we wanted to analyze and execute it a different number of times to understand when using memoization is beneficial. We have shown that for most cases to read 9 times a memoized value is sufficient. If a method is called most of the times with different values, than memoization will not be useful.

6 Related Work

Yang et al. proposed a technique and tool to determine the functional purity of Java methods [29]. In particular, if a method is a pure function, then it can be memoized. In their evaluation they were able to successfully memoize several methods from 3 different Java libraries and to reduce its execution time. As expected the memory usage increased due to the memo table.

A quite similar approach has also been proposed by Agosta et al. [1], since in their work they have also defined which Java methods can be memoized based on their functional purity. They defined a theoretical model to predict the effectiveness of the memoization in terms of energy consumption, but their approach has been applied to a particular set of computations in a desktop computer. In our experiments we have had a similar result. The most important difference is that we have shown that memoization has also a positive impact in the energy consumption in mobile devices.

Banerjee and Roychoudhury proposed a set of guidelines to improve energy consumption of mobile devices related to the use of energy-intensive resources

such as the GPS or the camera [2]. Based on these guidelines they propose a set of refactorings to ensure the guidelines are followed within an application. They have shown that this technique allows to spent between 3% and 29% less energy. Our solution can be combined with this one as they address different issues, possibly achieving even greater gains.

Cruz and Abreu have also proposed a set of refactorings based on performance guidelines to improve the energy consumption of Android applications [8]. They have shown that these refactorings improve the energy consumption, although this does not occur for all applications and refactorings.

7 Conclusions

In this work we have explored the use of memoization in Android applications, focusing on its impact in the energy consumption. We selected 18 methods from 3 different applications and designed an experiment to evaluate the energy consumption of such methods, as well as their memoized versions. The results from this experiment show that indeed the use of memoization promotes energy saving in most cases. These are our first experimental results, and we will further study the impact of these energy savings in the overall consumption of the applications.

The goal of this paper was to show the impact of memoization on energy consumption. However, it will be also interesting to relate these results to the execution time and energy consumption obtained with our memoization refactoring. We will pursue this topic in future work.

References

1. Agosta, G., Bessi, M., Capra, E., Francalanci, C.: Automatic memoization for energy efficiency in financial applications. Sustainable Computing: Informatics and Systems **2**(2), 105–115 (2012), IEEE Int. Green Computing Conference (IGCC'11)
2. Banerjee, A., Roychoudhury, A.: Automated re-factoring of android apps to enhance energy-efficiency. In: Proceedings of the International Conference on Mobile Software Engineering and Systems. pp. 139–150. ACM, New York, NY, USA (2016)
3. Banerjee, A., Roychoudhury, A.: Future of mobile software for smartphones and drones: Energy and performance. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. pp. 1–12. IEEE Press (2017)
4. Chambers, J.M., Cleveland, W.S., Kleiner, B., Tukey, P.A.: Graphical Methods for Data Analysis. Chapman and Hall/CRC (2017)
5. Cohen, J.: Statistical power analysis for the behavioral sciences . hilsdale. NJ: Lawrence Earlbaum Associates **2** (1988)
6. Couto, M., Cunha, J., Fernandes, J.P., Pereira, R., Saraiva, J.: GreenDroid: A tool for analysing power consumption in the android ecosystem. In: Scientific Conf. on Informatics, 2015 IEEE 13th International. pp. 73–78 (Nov 2015)
7. Couto, M., T., C., Cunha, J., Fernandes, J.P., Saraiva, J.: Detecting anomalous energy consumption in android applications. In: Quintão Pereira, F.M. (ed.) Programming Languages, LNCS, vol. 8771, pp. 77–91. Springer Int. Publishing (2014)
8. Cruz, L., Abreu, R.: Performance-based guidelines for energy efficient mobile applications. In: 4th Conf. on Mobile Software Eng. and Systems. pp. 46–57 (2017)

9. D'Agostino, R.B.: An omnibus test of normality for moderate and large size samples. Biometrika **58**(2), 341–348 (1971)
10. Field, A.: Discovering statistics using SPSS. Sage publications (2009)
11. Hao, S., Li, D., Halfond, W.G.J., Govindan, R.: Estimating mobile application energy consumption using program analysis. In: Proc. of 35th Int. Conf. on Software Engineering (ICSE) (May 2013)
12. Hao, S., Li, D., Halfond, W., Govindan, R.: Estimating android applications' cpu energy usage via bytecode profiling. In: Green and Sustainable Software (GREENS), 2012 First Int. Workshop on. pp. 1–7 (June 2012)
13. Hegde, N., Melanson, E.L., Sazonov, E.: Development of a real time activity monitoring android application utilizing smartstep. In: Proceedings of the 38th IEEE Int. Conf. of the Engineering in Medicine and Biology Society. pp. 1886–1889 (2016)
14. Hoque, M.A., Siekkinen, M., Khan, K.N., Xiao, Y., Tarkoma, S.: Modeling, profiling, and debugging the energy consumption of mobile devices. ACM Comput. Surv. **48**(3), 39:1–39:40 (Dec 2015)
15. Hu, Y., Yan, J., Yan, D., Lu, Q., Yan, J.: Lightweight energy consumption analysis and prediction for android applications. Science of Computer Programming (2017)
16. Jabbarvand, R., Sadeghi, A., Garcia, J., Malek, S., Ammann, P.: EcoDroid: An approach for energy-based ranking of android apps. In: Procs. of the 4th Int. Workshop on Green and Sustainable Software. pp. 8–14. IEEE Press (2015)
17. Li, D., Halfond, W.G.J.: An investigation into energy-saving programming practices for android smartphone app development. In: Proc. of 3rd Int. Workshop on Green and Sustainable Software. pp. 46–53. GREENS 2014, ACM (2014)
18. Li, D., Hao, S., Halfond, W.G.J., Govindan, R.: Calculating source line level energy information for android applications. pp. 78–89. ISSTA 2013, ACM (2013)
19. Li, D., Jin, Y., Sahin, C., Clause, J., Halfond, W.G.J.: Integrated energy-directed test suite optimization. pp. 339–350. ISSTA 2014, ACM (2014)
20. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., Poshyvanyk, D.: Mining energy-greedy api usage patterns in android apps:an empirical study. In: 11th Conf. on Mining Software Repositories. pp. 2–11 (2014)
21. Liu, K., Pinto, G., Liu, Y.D.: Data-oriented characterization of application-level energy optimization. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering, LNCS, vol. 9033, pp. 316–331. Springer (2015)
22. Pereira, R., Couto, M., Cunha, J., Fernandes, J.P., Saraiva, J.: The influence of the java collection framework on overall energy consumption. In: Proc. of 5th Int. Workshop on Green and Sustainable Software. pp. 15–21. ACM (2016)
23. Pinto, G., Castor, F.: Characterizing the energy efficiency of java's thread-safe collections in a multi-core environment. In: Proc. of SPLASH'2014 workshop on Software Engineering for Parallel Systems (SEPS), SEPS. vol. 14 (2014)
24. Pinto, G., Castor, F., Liu, Y.D.: Mining questions about software energy consumption. In: 11th Conf. on Mining Software Repositories. pp. 22–31. ACM (2014)
25. Rosenthal, R.: Meta-analytic procedures for social research, vol. 6. Sage (1991)
26. Rosenthal, R., Cooper, H., Hedges, L.: Parametric measures of effect size. The handbook of research synthesis pp. 231–244 (1994)
27. Sahin, C., Pollock, L., Clause, J.: How do code refactorings affect energy usage? In: Symp. on Empirical Sw Engineering and Measurement. pp. 36:1–36:10 (2014)
28. Sawilowsky, S.: New effect size rules of thumb **8**, 597–599 (11 2009)
29. Yang, J., Hotta, K., Higo, Y., Kusumoto, S.: Towards purity-guided refactoring in Java. In: Int. Conf. on Software Maintenance and Evolution. pp. 521–525 (2015)
30. Yang, J., Hotta, K., Higo, Y., Kusumoto, S.: Revealing purity and side effects on functions for reusing Java libraries. pp. 314–329. ICSR 2014, Cham (2014)