Empirical Software Engineering (2018) manuscript No.
(will be inserted by the editor)

Getting the Most from Map Data Structures in Android

Rubén Saborido[1] · Rodrigo Morales[1] · Foutse Khomh[1] · Yann-Gaël Guéhéneuc[1] · Giuliano Antoniol[1]

Received: June 2017 / Accepted: –

Abstract A map is a data structure that is commonly used to store data as key–value pairs and retrieve data as keys, values, or key–value pairs. Although Java offers different `map` implementation classes, Android SDK offers other implementations supposed to be more efficient than `HashMap`: `ArrayMap` and `SparseArray` variants (`SparseArray`, `LongSparseArray`, `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`). Yet, the performance of these implementations in terms of CPU time, memory usage, and energy consumption is lacking in the official Android documentation; although saving CPU, memory, and energy is a major concern of users wanting to increase battery life.

Consequently, we study the use of map implementations by Android developers in two ways. First, we perform an observational study of 5,713 Android apps in GitHub. Second, we conduct a survey to assess developers' perspective on Java and Android map implementations. Then, we perform an experimental study comparing `HashMap`, `ArrayMap`, and `SparseArray` variants map implementations in terms of CPU time, memory usage, and energy consumption. We conclude with guidelines for choosing among the map implementations: `HashMap` is preferable over `ArrayMap` to improve energy efficiency of apps, and `SparseArray` variants should be used instead of `HashMap` and `ArrayMap` when keys are primitive types.

Rubén Saborido
ruben.saborido-infantes@polymtl.ca

Rodrigo Morales
rodrigo.morales@polymtl.ca

Foutse Khomh
foutse.khomh@polymtl.ca

Yann-Gaël Guéhéneuc
yann-gael.gueheneuc@polymtl.ca

Giuliano Antoniol
giulio.antoniol@polymtl.ca

[1] Département de génie informatique et génie logiciel, École Polytechnique de Montréal, Québec, Canada

Keywords Android · Map data structure · Map implementations · CPU usage · Memory usage · Energy consumption

1 Introduction

Android is a popular open-source operating system developed by Google for mobile devices. Android is successful in part due to the availability of hundreds of thousands of apps written using the Android Software Development Kit (SDK) and Java.

Google has recently mentioned, during the Google I/O Developers Festival in May 2017, that there are two billion active Android devices in the world[1]. Developers should manage resources mindfully because emerging markets own a significant share of this installed base; for example, there are more Android users in India than in the United States of America. However, many of the devices sold in emerging markets are resource constrained. To mitigate these factors, Google has also announced Android Go, which is a lightweight version of the operative system that is optimized for low-cost devices with less than one gigabyte of memory. Hundreds of millions of people around the world are making their way on-line for the first time, and Google wants to create a better experience for them. The new Android experience will ship in 2018 for all Android devices that have up to one gigabyte of memory. Google recommends taking a look at the *Building for Billions*[2] to learn about the importance of offering a useful offline state, reducing `apk` size, and minimizing memory and battery usages.

Previous empirical studies indicated that software engineers can help reduce energy consumption by considering the energy impacts of their design and implementation decisions, e.g., using dark colors in their graphical user interfaces (Li et al. (2014)), considering the performance of different data structures (Manotas et al. (2014)), or removing anti-patterns in Android apps (Morales et al. (2017)). Hasan et al. (2016) showed that the Java implementations of various data structures differ significantly in terms of energy consumption, depending on the operations (insertions, iterations, and queries). They made developers aware of the consequences of their decisions. e.g., while `List` and `Set` collections consume about the same energy for the same operations, `HashMap` is the most energy-efficient Java map implementation. A map is a data structure used to store and retrieve data as key–value pairs, each key being unique.

Android SDK offers specialized map implementations and a series of video tutorials discussing performance issues[3]. The Android developers' reference documentation states that *"`ArrayMap` is designed to be more memory efficient than a traditional `HashMap`"*[4]. When keys are defined as integer primitive types, the documentation also states that *"`SparseArray` is designed to be more memory efficient than `HashMap` to map integers to objects"*[5]. The same is stated about `LongSparseArray` and long primitive types used as keys[6]. When keys are defined as integer primitive types and values are defined as integer, long, or boolean primitive types, the documentation also states

[1] https://youtu.be/Y2VF8tmLFHw
[2] https://developer.android.com/topic/billions/index.html
[3] https://developer.android.com/topic/performance/index.html
[4] https://developer.android.com/reference/android/util/ArrayMap.html
[5] https://developer.android.com/reference/android/util/SparseArray.html
[6] https://developer.android.com/reference/android/util/LongSparseArray.html

that `SparseIntArray`[7], `SparseLongArray`[8], and `SparseBooleanArray`[9], respectively, are designed to be more memory efficient than a traditional `HashMap`. In addition to the previous, Android Studio, the official Android integrated development environment (IDE), warns *"Use `new SparseArray` instead `new HashMap<Integer,Object>()` for better performance"* (a similar warning is also given for `SparseArray` variants). Hence, `ArrayMap` and `SparseArray` variants should be preferred over `HashMap`, at least for maps containing up to hundreds of elements according to Android developers' reference documentation. For `ArrayMap` and `SparseArray` variants the documentation claims that *"this implementation is not intended to be appropriate for data structures that may contain large number of items. It is generally slower than a traditional `HashMap`"*.

Yet, the documentation is vague because (1) it does not provide supporting evidence and quantitative information about efficiency and (2) although it discourages their use in maps containing large number of elements, it does not provide more precise numbers (*e.g.,* performance information and–or threshold levels to consider). Consequently, although the current documentation raises the awareness of developers about the advantages and limitations of map implementations, it does not provide concrete evidences that could be used to make informed decisions about the implementations that are the most suitable for their apps. Expressions such as *"large number"* and *"generally slower"* are vague and they do not help developers at all. In addition to the previous, the documentation says nothing about energy consumption and neither about performance for different map-related operations and data sizes.

In this paper, we study the performance of the map implementations `ArrayMap` and `SparseArray` variants provided by Android in comparison to `HasHMap` and provide guidelines for developers to make informed decisions. First, we perform the largest observational study on the use of map implementations in mobile apps. We analyze all of the Android apps hosted on GitHub and available in the official Android marketplace, Google Play[10], as of November 23, 2016 (i.e., 5,713 apps). We report that (1) `HashMap` is the most used map implementation, (2) `ArrayMap` and `SparseArray` variants are rarely used, and (3) `HashMap` is often adopted even if `SparseArray` variants seem to be more appropriated. Second, we survey the Android developers of the analyzed apps to understand their perspective with respect to map implementations. We contacted 656 developers and received 118 (18%) completed surveys. We report that developers are moderately familiar with the implications of different Android map implementations, and they use `HashMap` because it is well-known. However, most developers (86%) would replace `HashMap` if they had more concrete information about the performance of other implementations. Consequently, we perform an empirical study of the CPU and memory usages and energy consumption of `HashMap`, `ArrayMap`, and `SparseArray` variants map implementations. We find that `ArrayMap` is less energy efficient and slower than `HashMap`, although the former is a bit better in terms of memory use than the latter. We estimate that `HashMap` is, on average, 13% faster and consumes 16% less energy than `ArrayMap`. However, `ArrayMap` uses less memory (6%) than `HashMap`. If keys are primitive types, we find that `SparseArray` variants are more efficient than `HashMap` for all performance metrics and most operations. The average incurred cost of using `HashMap` instead of `SparseArray` variants is 25% more CPU time, 62% more memory,

[7] https://developer.android.com/reference/android/util/SparseIntArray.html
[8] https://developer.android.com/reference/android/util/SparseLongArray.html
[9] https://developer.android.com/reference/android/util/SparseBooleanArray.html
[10] https://play.google.com/store/apps

and up to 6% more energy. The incurred cost of using `ArrayMap` instead of `SparseArray` variants follows a similar trend than `HashMap`.

We conclude that Android developers prefer `HashMap` because they are reluctant to adopt Android implementations by speculating on possible gains in efficiency. Although using Android map implementations can really save CPU and memory usages and reduce energy consumption. For most common map-related operations we provide guidelines for choosing among different map implementations taking into account their performance. We thus make the following contributions:

– A quantitative analysis of the use of map implementations in all the 5,713 apps Android apps available in GitHub on December, 2016.
– A survey of 118 Android developers on their knowledge and usage of the different map implementations.
– A quantitative analysis of the impact of using `HashMap`, `ArrayMap`, or `SparseArray` variants on performance metrics.

We ensure the replicability of our study by making available, in a replication package[11], our survey, all Android apps and test cases developed, and scripts and collected data.

In the following, Section 2 discusses Java and Android implementations of map data structures. Section 3 describes our observational study on the usage of map implementations in Android apps. Section 4 presents the survey of Android developers. Section 5 summarizes our experiments to assess the performance of the map implementations. Section 6 offers guidelines to developers. Section 7 warns about threats to the validity of our results. Section 8 summarizes related work. Section 9 concludes with future work.

2 Background

Java offers three general-purpose map implementations in the package `java.util`: `HashMap`[12], `LinkedHashMap`[13], and `TreeMap`[14]. `HashMap` is an array of `HashMap.Entry` instances; i.e., pairs of non-primitive keys and values. When a key–value pair is put in a `HashMap`, a hash-code of the key is calculated and used to obtain the index of the `Entry` in the array. `LinkedHashMap` is a hash table and linked list combined to implement a `Map` with predictable iteration order. Contrary to `HashMap`, `LinkedHashMap` maintains a doubly-linked list of all its entries, which defines the iteration ordering. `TreeMap` is another implementation that uses tree to store key–value pairs in sorted order, allowing rapid retrieval.

Constructors of `HashMap` and `LinkedHashMap` have two optional parameters that affect their performances: capacity and load factor. The default load factor is 0.75 which, regarding the official documentation, offers a "good" trade-off between CPU and memory usages. The default capacity is the number of buckets in the hash table at the time the hash table is created, 16: resizing will occur when $16 \times 0.75 = 13^{th}$ element is inserted. The drawback of these map implementations is in the use of non-primitive types. (Auto-)Boxing is required for primitive types, which creates extra objects at

[11] http://www.ptidej.net/downloads/replications/ese17a/
[12] https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
[13] https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html
[14] https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

insertion. Finally, entries must be rearranged each time the array underlying the maps is compacted or expanded and both implementations must store both keys and their hash-codes to avoid collisions but increasing memory usage.

Android offers its own map implementations. The implementation `ArrayMap` is supposed to be more memory-efficient than `HashMap`. `ArrayMap` keeps its mappings in an array using an integer array of hash codes for each item and an `Object` array of the key–value pairs. Thus, it avoids creating extra objects for every entry put into the map and control the growth of the sizes of its arrays more aggressively (growing them only requires copying the entries in the array, not rebuilding a hash map). Although this implementation was designed to be more memory-efficient than `HashMap`, `ArrayMap` still does not solve the problem of (auto-)boxing given that its `put` method still takes two `Object`s as parameters.

If the key is an integer, Android suggests to use `SparseArray` that maps integers to `Objects`. `SparseArray` is intended to be more memory-efficient than `HashMap`. It avoids (auto-)boxing keys and its implementation does not add the extra overhead of an entry object for each mapping. Android provides different variants of `SparseArray` that map to different primitive types: `LongSparseArray` that maps long type keys to objects, and `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` that map integers to objects for keys and different primitive types to objects for values.

Both `ArrayMap` and `SparseArray` variants are located in the package `android.util` available from the Android API, but `ArrayMap` is only available from the Android API level 19 (KitKat) and higher. An alternative implementation of `ArrayMap` is located in the package `android.support.v4.util`, for older versions of Android. We study both implementations but only report the results of the implementation located in the package `android.util` because apps targeting the Android API 19 and later will run on more than 90% of the devices that are active on the Google Play Store[15].

3 Observational Study

We study usage patterns of Android developers of Java map implementations and the Android map implementations `ArrayMap` and `SparseArray` variants. We conduct this study to analyze the prevalence of map implementations. We define the following research question, which we answer through an observational study of Android apps available on GitHub.

RQ1. *What map implementations do Android developers use?*

3.1 Objects

We selected from GitHub (repository queried and accessed December, 2016) all the projects that advertised to be available in the official Android marketplace as Android apps; i.e., all the projects that contained a link to the Google Play marketplace in their *README.md* file. We chose open-source apps to study the prevalence of map implementations in the source code.

[15] https://developer.android.com/about/dashboards/index.html

3.2 Procedure

We developed a Python script to select and download, automatically, the zip file containing the source code of each existing Android project from GitHub. Thus, we obtained 5,713 Android apps. We also developed a Bash script to process the source code of all the apps looking for occurrences of general-purpose Java map implementations (`HashMap`, `LinkedHashMap`, and `TreeMap`) as well as the Android map implementations (`ArrayMap` and `SparseArray` variants). This script produces a comma-separated values (CSV) file. For each Android app and map implementation, the file contains a boolean value to specify if an app has one or more occurrences of a map implementation in its source code.

3.3 Results

We obtain that, over 5,713 apps, 1,713 (30%) apps have at least one occurrence of any Java map implementation. For `ArrayMap` and `SparseArray` variants, 419 (7%) apps have one or more occurrences of these map implementations. In total, over 5,713 apps, 2,132 (37%) use any of the studied map implementations. From now on, in this section, percentages are given with respect to this number.

We find that `HashMap` is the most used Java map implementation with 1,640 apps (77%) while the others are used less often. We obtain that 282 (13%) apps use `LinkedHashMap` and 179 (8%) use `TreeMap`. Note that different map implementations can be used in the same app. Concerning Android map implementations, we find that `ArrayMap` and `SparseArray` variants are rarely used by Android developers. Only 19 (1%) and 413 (7%) apps use `ArrayMap` or any variant of `SparseArray`, respectively.

Table 1 shows the number and the percentage of apps that have one or more occurrences of any combination of the Java and Android map implementations. The second column shows the number and percentage of apps that have one or more occurrences of `ArrayMap` and one or more occurrences of a Java map implementation. The third column is similar to the previous one but for `SparseArray` variants. The last column shows the number and percentage of apps that have one or more occurrences of both `ArrayMap` and any variant of `SparseArray` map implementations and one or more occurrences of a Java map implementation. We observe that `ArrayMap` and `SparseArray` variants are used in combination with `HashMap`. This is expected for `SparseArray` variants, because these map implementations are used when keys are primitive types while `HashMap` can be used to store non-primitive types as keys and–or values. However, `ArrayMap` could be used as a replacement for `HashMap` but we find that only five (0.23% over 2,132 or 0.30% over the 1,640 apps using `HashMap`) apps use exclusively `ArrayMap` instead of `HashMap`.

The Android documentation claims that `SparseArray` variants have a better memory performance than `HashMap` and, for this reason, Android Studio suggests to replace `HashMap` by `SparseArray` variants. Because we find that `HashMap` is the most used map implementation we also study whether Android developers adopt the `HashMap` implementation when keys are defined as primitive types. We develop another script to process the source code of apps looking for usages of the `HashMap` implementation using primitive types as keys. This script produces a CSV file containing for each Android app a boolean value that indicates whether an app uses this map implementation. From the 1,640 apps using the `HashMap` implementation, 332 (20%) of

Table 1 Number and percentage of Android apps having one or more occurrences of any combination of the Java and Android map implementations.

	Android map implementation		
Java map implementation	`ArrayMap`	`SparseArray` variants	Both
`TreeMap`	3 (<1%)	46 (2%)	2 (<1%)
`LinkedHashMap`	4 (<1%)	100 (5%)	3 (<1%)
`HashMap`	14 (<1%)	314 (15%)	8 (<1%)

these apps use integers as keys, 64 (4%) use longs as keys, 89 (5%) use integers as keys and values, 12 (<1%) use integers as keys and longs as values, and 13 (<1%) use integers as keys and booleans as values. However, in these cases, Android recommends to replace `HashMap` with `SparseArray`, `LongSparseArray`, `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, respectively, for better memory performance.

3.4 Discussion of the Observational Study

From the observational study we answer **RQ1** (*What map implementations do Android developers use?*) concluding the following:

– `HashMap` is the most used Java map implementation in Android apps.
– `ArrayMap` and `SparseArray` variants map implementations are rarely used in Android apps.
– Although `ArrayMap` is designed by Android to be more memory efficient than a traditional `HashMap`, less than 1% of Android apps use `ArrayMap` as a replacement for `HashMap`.
– Although the Android documentation and Android Studio highly suggest to replace `HashMap` by `SparseArray` variants when keys are primitive types, Android developers are not following this practice.

We think that `HaspMap` is the most used map implementation since Android apps are mainly usually written in Java and most Android developers are likely used to this programming language. Because of this, we believe that Android specific map implementations are not always well-known and taken into account. We also think that `SparseArray` variants are much more used than `ArrayMap` in Android apps because Android Studio warns about using variants of `SparseArray` instead of `HashMap` with primitive type keys. As we show in Section 4, most developers of the analyzed apps use Android Studio as IDE.

4 Developers' Perspective

We are interested to know why developers mostly select the Java map implementation `HashMap` instead of `ArrayMap` and `SparseArray`. Particularly, in the cases where the Android documentation advises the opposite. We assume that this lack of use is due to developers' reluctance to try new implementations. But it is also probably due to their lack of knowledge about the possible advantages in terms of performance of switching to `ArrayMap` or `SparseArray`. We define the following research question, which we answer by conducting an on-line survey. All questions were optional and the survey was anonymous to encourage developers to participate (Tyagi (1989)).

RQ2. *What is the developers' perspective with respect to map implementations?*

4.1 Subjects

We considered the 1,744 apps that use `HashMap`, `ArrayMap`, and–or any variant of `SparseArray`. We contacted those project's owners who that made their email address publicly available in GitHub. The total amount of emails sent was 656. We surveyed these 656 developers and 118 (18%) responded to our survey. This rate is considerably larger than the typical 5% answer rate obtained in questionnaire-based software engineering surveys (Singer et al. (2008)). The survey was available on-line from December 2016 to January 2017.

Participants in our survey were from 35 different countries around the world. The majority of them was from USA (17, 14%), India (15, 13%), and Spain (seven, 6%). Of all 118 participants: 93 (79%) had up to four years of experience developing mobile apps, 101 (86%) participants declared to use Java as primary programming language, and 97 (82%) participants declared to use Android Studio as IDE.

4.2 Instrument

The on-line survey contained 14 questions: four on the usage of map implementations, five on the participants' familiarity with Android map implementations, one about the importance of performance metrics, and four on the participants' background and experience.

All the questions had a closed set of answers from which a participant selects, while two of them included an additional field for open comments. None of the questions were mandatory and participants were allowed to drop out at any time.

The survey consisted of four different sections asking about the use of map implementations, the familiarity with map implementations offered by Android, the importance of different performance metrics, and participants' profile. We accept answers using a dichotomous scale (*yes*/*no*) or Likert scales with values from 1 (*never*) to 5 (*every time*). When the question was related to importance magnitude we used a similar scale with values from 1 (*not important*) to 5 (*very important*). For questions about familiarity we used the same scale with values from 1 (*not at all familiar*) to 5 (*extremely familiar*).

The survey was designed to be completed within approximately 5-8 minutes. In order to automatically collect the answers, we hosted the survey using the web app Google Forms. We allowed developers to complete the questionnaire in multiple rounds.

4.3 Results

Demographic and experience information about participants was reported previously but results for the other three sections are presented next. For Likert scale questions we use diverging stacked bar charts to show the frequency of responses (in percentage). Replies are positioned horizontally, so "low" responses are stacked to the left of a vertical baseline and "high" responses are stacked to the right of this baseline. We consider as baseline or "neutral response" the midpoint of the scale (value 3). For each stacked bar we also add the percentage of low, neutral, and high responses. In addition, on the right, we also show the total number of participants who answered each of the questions. In all the questions we obtained more than 110 responses (over 118 participants).

Use of Map Implementations

We asked participants the frequency of usage of the Java map implementations `TreeMap`, `LinkedHashMap`, and `HashMap`. Figure 1 summarizes the participants' responses. As it is shown, `HashMap` is the most often used Java map implementation (80% of participants responded *almost every time* or *every time*).

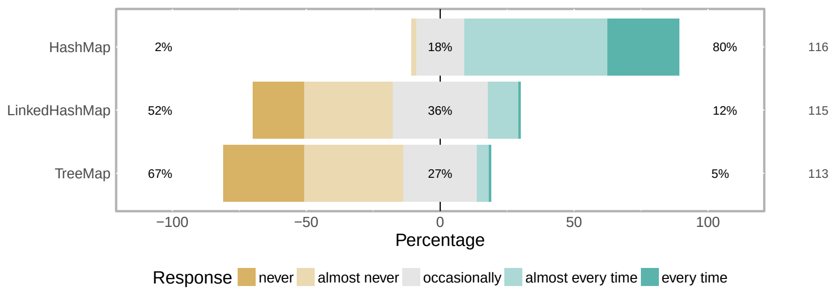

Fig. 1 Participants' responses about the usage of most popular Java map implementations.

Concerning map-related operations (insertion, iteration, random query, and deletion), we asked participants to rate them according to their importance in their codes. Figure 2 summarizes the participants' responses. For insertion, iteration, and random query operations, more than 70% of the participants responded *moderately important* or *very important*. For the deletion operation, 52% of the participants considered it an important operation while 27% of the participants had a neutral opinion about it.

Regarding the iteration operation we asked developers about the way of iterating through map structures. We received 118 responses from participants: 94 (80%) of these participants selected *for-Each loop* instead of *the Iteration pattern*, which was selected by 24 (20%) participants. In addition to this, we asked if the iteration is used over the *key–value pairs*, the *set of keys*, or the *set of values*. We received 118 responses: 59 (50%) participants selected *key–value pairs*, 41 (35%) chose the *set of keys* because they get the value mapped to each key using the `get` method, and 18 (15%) selected the *set of values* because they are not usually interested in keys.

Familiarity with Android Map Implementations

In the second section of the survey, we asked about the familiarity of developers with the Android map implementations `ArrayMap` and `SparseArray`. We focused on

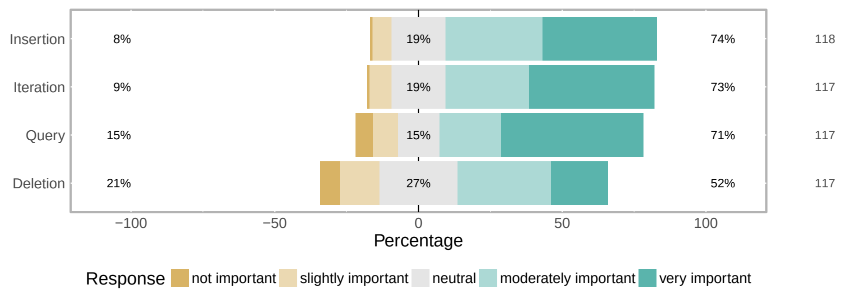

Fig. 2 Participants' responses about the importance of map-related operations.

`SparseArray` and not on the other variants because they are used less often. Figure 3 summarizes the participants' responses. For `ArrayMap`, 55% of participants responded *moderate familiar* or *extremely familiar*. For `SparseArray`, 27% of participants responded *moderate familiar* or *extremely familiar*. Half of the participants responded that they were *not at all familiar* or *slightly familiar* with `SparseArray`.

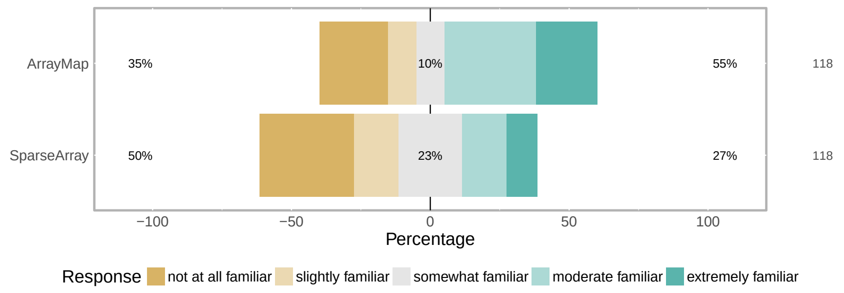

Fig. 3 Participants' responses about familiarity with Android map implementations.

Since `ArrayMap` was introduced in the Android API 19, developers from early versions of Android or even those who want to provide compatibility backwards may opt for not switching to this implementation. To counter effect this issue, Android offers a utility class `android.support.v4.util` that allows to use `ArrayMap` in early Android versions. We asked developers to share whether they were aware of this class. 118 participants answered this question, from whom 49 (42%) were aware. The rest, 69 participants (58%), could hesitate to use it if they want to bring compatibility back to previous versions, like one respondent said: "*My app is an uncommon case, I support all the way back to API level 8... I do however use SparseArray map, as it is available since API 1*".

With respect to `SparseArray`, we asked participants if they knew that it is designed to be more efficient than `HashMap` when keys are integers. 118 participants answered this question, from whom 52 (44%) confirmed this fact. The rest, 66 participants (56%), answered *no*.

We asked participants if they were willing to replace `HashMap` with any of the map implementations provided by Android, if they offer better performance. 118 participants answered this question, from whom 102 (86%) answered *yes*. To the 16 (14%) remaining participants, we asked why they chose *no*. In general, they answered that they use what they know and they fear learning or incorrectly using new structures. Another respondent said that (s)he is concerned by the size of the app's `apk`, because adding the Android library would increase its size. In addition, another respondent was worried about portability of code when Android libraries are included. We were more concrete and we also asked if they were willing to substitute `HashMap` with `SparseArray` when integers are used as key in map data structures. 118 participants answered this question, from whom 107 (91%) said *yes* and 11 (9%) *no*. When we asked about the reason, one participant said that "*the replacement effort may be huge because they have different interfaces*".

Importance of Performance Metrics

Finally, we asked participants to rank performance metrics (CPU time, memory usage, and energy consumption) according to the weight that they give when choosing a map implementation. Figure 4 summarizes the participants' responses. As it is shown, more than 70% of the participants responded *moderately important* or *very important* for CPU time and memory usage. About energy consumption, 43% of the participants responded that this performance metric is important while 27% of the participants had a neutral opinion about it.

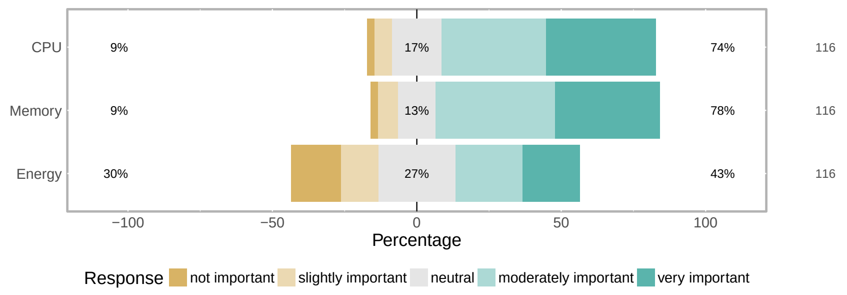

Fig. 4 Participants' responses about importance of performance metrics when choosing a map implementation.

4.4 Discussion about Developer's Perspective

Based on the results obtained from the survey, we answer **RQ2** (*What is the developers' perspective with respect to map implementations?*) concluding the following:

– We confirm what we found in our observational study (Section 3): `HashMap` is the most popular map implementation and developers are moderately familiar with

Android map implementations. In addition, we also confirm that most Android developers use Java as primary programming language.
- Developers are not aware of the overhead incurred when selecting an inappropriate map implementation and that is why they stick with the well-known `HashMap`.
- Participants accepted to be more familiar with `ArrayMap` than with respect to `SparseArray`. However, in our observational study (Section 3) we found that the Android map implementation `SparseArray` was used in more apps than `ArrayMap`. According to our survey, most developers use Android Studio as IDE. Hence, we could suggest that `ArrayMap` is not typically used by them because Android Studio does not suggest it as an alternative to `HashMap`. However, Android Studio warns about using `SparseArray` instead of `HashMap` with integer keys.
- The most important map-related operations for Android developers are insertion, iteration, and random access.
- Most developers iterate through map data structures over key–value pairs, and they prefer to do that using a `for-Each` loop.
- When Android apps developers choose a map implementation, CPU time and memory usage are more important metrics than energy consumption. Even if battery life is a main concern for mobile device users, this metric is less important for developers than CPU time and memory usage. Energy is more difficult to measure because specific knowledge and software tools or–and specific hardware could be required. We believe that CPU time and memory usage are more important performance metrics for developers because they can easily measure both of them.

5 Experimental Study

In our observational study we found that `HashMap` is the most used map implementation. From the survey we concluded that developers are not aware of the cost of selecting map implementations. For this reason, developers are reluctant to use new map implementations and they use what they know that works. We believe that if developers are provided with concrete results about the performance of maps in terms of critical performance metrics (e.g., CPU time, memory usage, and energy consumption), they will be able to make informed decisions. Consequently, we define the following research question.

RQ3. *What is the performance of `HashMap` and `ArrayMap`?*

We answer this question through an experimental study about the performance of `HashMap`, the most popular Java map implementation, and `ArrayMap`, a map implementation proposed by Android as a replacement for `HashMap`.

In addition, as we found in our observational study, 20% of the apps using `HashMap` use integers as keys, 4% use longs as keys, 5% use integers as keys and values, <1% use integers as keys and longs as values, and <1% use integers as keys and booleans as values. For these cases, Android map implementations `SparseArray`, `LongSparseArray`, `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` are suggested by Android to be more efficient map implementations. This is the rational that led us to define the following research question.

RQ4. *What is the cost of adopting `HashMap` instead of `SparseArray` variants?*

We answer this research question by studying the performance of the `SparseArray` variants map implementation.

Although Android proposes `SparseArray` variants as a replacement for `HashMap`, nothing is said about the usage of `ArrayMap` for primitive types. For this reason we also define the following research question.

RQ5. *Is `ArrayMap` as efficient as `SparseArray` variants?*

Therefore, we answer these three research questions by analyzing the performance of `HashMap`, `ArrayMap`, and `SparseArray` variants in terms of CPU time, memory usage, and energy consumption. We analyze these map implementations for different data sizes and for the four most important operations: insertion, iteration, random query, and deletion.

5.1 Design

We ran experiments with `HashMap`, `ArrayMap`, and `SparseArray` variants considering 30 different data sizes in the range $[1, 80000]$. The Android documentation says that `ArrayMap` and `SparseArray` variants are not intended to be appropriate for data structures that may contain large numbers of items. Because "large" is a vague quantity, we use as upper bound for data size a quantity 15 times larger than the one used by Hasan et al. (2016). Thus, we want to know how the performance of map implementations is affected when up to 80,000 elements are used in map data structures. For all the map implementations we used the default initial capacity and the default load factor value of 0.75 for `HashMap`, as it is proposed in the Java documentation.

Because `SparseArray` variants are designed to be used with specific primitive type keys, we compare them with respect to `HashMap` and `ArrayMap` setting the same primitive types. Additionally, we also compare `HashMap` and `ArrayMap` implementations using objects (strings) as keys, instead of primitive types. Thus, we analyze 17 different map implementations. They are shown in Table 2. The first column contains the specific types used as keys and values for the map implementations shown in the second column. The third column contains the syntax for the declaration of each map implementation.

In our experiments, we used four different operations over these map implementations. For each of the 30 data sizes used, map implementation, and operation, we collected performance metrics. We now explain in detail each of the four operations:

- *Insertion.* We created the data structure and we filled it by inserting the number of elements desired. The insertion was done using the `put` method of the map implementations.
- *Iteration.* We created the data structure and we filled it by inserting the number of elements desired. After that, we iterated, with a `for-Each` loop, over each `Entry` using the `entrySet` method of the `HashMap` and `ArrayMap` implementations. We used this approach to iterate over the elements of map data structures because, as observed in our survey, it is extensively used by developers. However, `SparseArray` variants do not offer an `entrySet` method. We iterated over these implementations by modifying the index between zero and the number of elements. We obtained the key and value from each indexed key–value mapping using the methods `keyAt` and `valueAt` of `SparseArray` variants, respectively.

Table 2 Subject map implementations for the experimental study.

Types for keys and values	Map implementation	Declaration
String keys and integer values	`HashMap`	`HashMap<String,Integer>()`
	`ArrayMap`	`ArrayMap<String,Integer>()`
Long keys and integer values	`HashMap`	`HashMap<Long,Integer>()`
	`ArrayMap`	`ArrayMap<Long,Integer>()`
	`LongSparseArray`	`LongSparseArray<Integer>()`
Integer keys and integer values	`HashMap`	`HashMap<Integer,Integer>()`
	`ArrayMap`	`ArrayMap<Integer,Integer>()`
	`SparseArray`	`SparseArray<Integer>()`
	`SparseIntArray`	`SparseIntArray()`
Integer keys and long values	`HashMap`	`HashMap<Integer,Long>()`
	`ArrayMap`	`ArrayMap<Integer,Long>()`
	`SparseArray`	`SparseArray<Integer,Long>()`
	`SparseLongArray`	`SparseLongArray()`
Integer keys and boolean values	`HashMap`	`HashMap<Integer,Boolean>()`
	`ArrayMap`	`ArrayMap<Integer,Boolean>()`
	`SparseArray`	`SparseArray<Integer,Boolean>()`
	`SparseBooleanArray`	`SparseBooleanArray()`

– *Random query.* We created the data structure and we filled it by inserting the number of elements desired. Then, using the `get` method of the map implementations, we returned the values to which the specified key of N random elements was mapped. Here, N is the data size of the data structure. To make a fair comparison, the same seed was used for all the data structures. Therefore, the same sequence of random numbers was always generated.
– *Deletion.* We created the data structure and we filled it by inserting the number of elements desired. Then, we iterated over the data structure removing one element at time. We removed elements (accessing by key) using the `remove` method of the `HashMap` and `ArrayMap` map implementations, and the `delete` method of the `SparseArray` variants.

Next, we explain the way CPU time, memory usage, and energy consumption were collected. For each of these performance metrics we used a different approach and various scripts developed by us. We used a LG Nexus 4 Android phone equipped with a quad-core Krait CPU@1500MHz, a 4.7-inch screen, and running Android Lollipop (version 5.1.1, Build number LMY47V). We believe that this phone is representative of the current generation of Android mobile phones. Nexus mobile phones are pure Android devices designed by Google and manufactured by one of the most important mobile companies to provide the best user experience.

CPU Time

We created an Android app for each map implementation which runs the four operations while it collects execution traces using the Android profiler. Execution traces were used to get the CPU time associated to each operation. We ran the experiments in an automatic way using a Python script. This script uses as input a text file specifying, on each line, the map implementation and the data size to use. Considering the first parameter, the map implementation, the script runs the corresponding Android app, which receives as a parameter the data size. When a tap event is detected on the screen, the Android app runs the insertion, iteration, random query, and deletion operations while the app is profiled using the Android debugger. After these actions are completed, the resulting execution traces are saved on the phone and then transferred to a server for backup and processing. Algorithm 1 shows the pseudo-code of our approach to measure CPU time of map implementations. Because we were analyzing 17 map implementations and 30 data sizes, we ran 510 (17 × 30) experiments to get CPU measurements.

```
forall map implementation and data sizes in input file do
    Install app of the current map implementation (using adb).
    Start app passing the data size as parameter (using adb).
    Wait to load the app completely.
    Touch the screen to run the experiment (using adb).
    Wait until experiment is finished.
    Download execution traces from the phone (using adb).
    Remove execution traces from the phone (using adb).
    Stop the app (using adb).
    Clean the app data (using adb).
    Uninstall the app (using adb).
end
```

Algorithm 1: Collection of CPU time for different map implementations, operations, and data sizes.

For each data size and map implementation, we obtained one execution trace per operation (insertion, iteration, random query, and deletion). Using a Bash script and the Android `dmtracedump` command, we processed execution traces to generate a CSV file containing the CPU time of each independent experiment. Execution traces generated by the Android profiler show both the inclusive and exclusive CPU times (as well as the percentage of the total time). Exclusive time is the time spent in the method. Inclusive time is the time spent in the method plus the time spent in any called functions. We use inclusive CPU time as CPU usage.

Memory Usage

We created an Android app for each map implementation to measure memory usage. Each app runs insertion operations over the corresponding map data structure and reports the memory difference before and after the insertion of elements. The Android app does the following: (1) get the amount of memory (in bytes) used by the Java Virtual Machine, (2) create and fill the corresponding data structure, (3) get the current amount of memory (in bytes) used by the Java Virtual Machine (using the methods `freeMemory` and `totalMemory` offered by the class `Runtime`), (4) calculate the difference between both memory values, and (5) save the resulting amount of memory in a text file on the phone. Thus, the generated file contains the memory used by the data structure, expressed in bytes. We used a Python script to collect memory usage of the studied map implementations automatically. This script uses as input a text file specifying, in each line, the map implementation to use and the data size. Considering the first parameter, the map implementation, the script runs the corresponding Android app that receives as parameter the data size of the map data structure. Algorithm 2 shows the pseudo-code of our approach to measure memory usage of map implementations.

We were analyzing 17 map implementations and 30 data sizes, so we ran 510 (17×30) experiments.

```
forall map implementation and data sizes in input file do
    Install app of the current map implementation (using adb).
    Start app passing data size as a parameter (using adb).
    Wait to load the app completely.
    Touch the screen to run the experiment (using adb).
    Wait until experiment is finished.
    Download the file with information about memory usage (using adb).
    Stop the app (using adb).
    Clean the app data (using adb).
    Uninstall the app (using adb).
end
```

Algorithm 2: Collection of memory usage for different map implementations and data sizes.

Energy Consumption

We designed a parametrized Android test suite for each map implementation with four Android test cases, one for each operation. The data size was considered as a parameter of each test case. We measured energy consumption in a real phone while we ran these test cases. Using Android test cases allowed us run experiments turning the screen off (which removes the impact on energy consumption from the screen) and stop the measurement process automatically when a test case finished. We ran automatically these Android test cases using a Python script. This script reads an input text file specifying, on each line, a test case to be run. A test case is defined by the name of a map implementation, the operation to run, and the data size. Thus, a line "`ArrayMap` INSERTION 1000" means that the test case inserts 1,000 elements in an `ArrayMap` data structure. We measured energy consumption using a digital oscilloscope TiePie Handyscope HS5[16]. TiePie offers the LibTiePie SDK, a cross platform library for using TiePie engineering USB oscilloscopes through third party software. This SDK allowed us to communicate with the oscilloscope in our script. Algorithm 3 shows the pseudo-code of our approach to measure energy consumption of map implementations. We analyzed 17 different map implementations, four different operations, and we used 30 different data sizes. In total we ran 2,040 ($17 \times 4 \times 30$) different test cases.

```
forall Android test cases in input file do
    Compose test case name.
    Start oscilloscope to measure energy consumption.
    Run Android test case (using adb).
    Stop oscilloscope.
end
```

Algorithm 3: Collection of energy consumption for different map implementations, operations, and data sizes.

[16] http://www.tiepie.com/en/products/Oscilloscopes/Handyscope_HS5

The mobile phone was powered by a power supply. Between both we connected, in series, a uCurrent[17] device. It is a precision current adapter for multimeters that converts the input current I proportionally to the output voltage V_{out}. Knowing I and the voltage supplied by the power supply V_{sup}, we used Ohm's law to calculate the power usage P as $P = V_{sup} \cdot I$. The resolution of the oscilloscope was set up to 16 bits and the frequency to $125kHz$; therefore, a measure was taken each eight microseconds. We calculated the energy associated to each sample as $E = P \cdot T = P \cdot (8 \cdot 10^{-6})s$, where P is the power of the smart-phone and T is the period sampling in seconds. The total energy consumption is the sum of the energy associated to each sample.

We connected the phone to an external power supplier that was connected to the phone's motherboard to avoid any kind of interference with the phone battery in our measurements. Figure 5 shows the connection diagram.

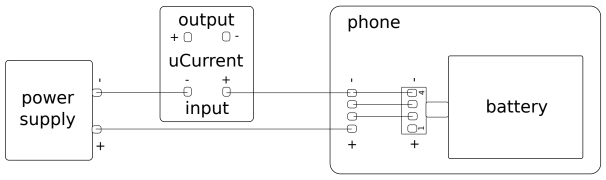

Fig. 5 Connection between the power supply, the uCurrent device, and the LG Nexus 4 phone.

The phone was connected via USB to the PC to send and receive data. However, using an Android app that we developed, we disabled[18] the USB charging on the device to avoid any interference in our measurements. This application is free and it is available for downloading[19] in Google Play.

5.2 Data analysis

Experiments were run 20 times to obtain statistical confidence. Hence, we ran $20,400$ $(510 \times 20 \times 2)$ experiments for collecting both CPU time and memory usage. We ran $40,800$ $(2,040 \times 20)$ Android test cases for energy measurements. Overall, the collection of performance metrics took around 800 hours (over five weeks) of continuous execution time and resulted in over 600 GB of raw data.

A Wilcoxon rank sum test was carried out to check if the difference observed between the values of the performance metrics was significant. In this case, the null hypothesis was that the distribution of performance metrics of the `HashMap` implementation and performance metrics of `ArrayMap` or `SparseArray` variants differed by a location shift of μ (the average value). We considered the difference to be significant if the obtained p-value was lower than $\alpha = 0.05$. In addition, when the comparison was significant, we computed the effect size using Cliff's Delta function from the R software[20].

[17] http://www.eevblog.com/projects/ucurrent/
[18] The mobile phone has to be rooted first.
[19] https://play.google.com/store/apps/details?id=ruben.nexus4usbcharging
[20] https://cran.r-project.org/web/packages/effsize/

5.3 Results

Figure 6 shows the distribution of values of each performance metric for each map implementation and map-related operation. CPU time is expressed in milliseconds (ms), memory usage in kilobytes (kB), and energy consumption in Joules (J). The deletion operation is omitted because performance metric values are much higher for this operation and some map implementations. Later we discuss this fact. Given a performance metric and a map-related operation, we compute for each data size the median value of each performance metric over the 20 runs. Figures 7, 8, 9, 10, and 11 show the median CPU time, memory usage, and energy consumption for each map implementation, data size, and operation. In addition, for each pair of map implementations we compute the average difference of the median values previously computed for each data size. We do this for each performance metric and map-related operation. We use all of this information to answer **RQ3**, **RQ4**, and **RQ5**.

RQ3. What is the performance of `HashMap` and `ArrayMap`?
To answer this research question, first, we compare both map implementations with object keys and primitive type values. Then, we compare `HashMap` and `ArrayMap` with primitive type keys and primitive type values.

Object keys and primitive type value

`HashMap` is faster than `ArrayMap` for iteration, random query operations, and deletion operations, but `ArrayMap` seems a bit faster than `HashMap` for insertion operations. We observe that `ArrayMap` is, on average, 31 ms (1%) faster for insertion operations. However, `HashMap` is, on average, 246 ms (13%), 422 ms (4%), and 5,516 ms (22%) faster than `ArrayMap` for iteration, random query, and deletion operations, respectively.

Concerning memory usage, `ArrayMap` is a bit more efficient than `HashMap`. We find that `ArrayMap` uses, on average, less memory (6%) than `HashMap`.

In terms of energy efficiency, `HashMap` consumes less energy than `ArrayMap` for all the operations (on average 7%, 6%, 6%, and 45%, for insertion, iteration, random query, and deletion operations, respectively).

The Wilcoxon statistical test concludes that differences in CPU time and memory usage are significant and the effect size is large for any data size. Differences in energy consumption are significant and the effect size is large when the data size is greater than or equal to 2,000 for insertion, iteration, and random query operations. For deletion operations, differences in energy consumption are significant when the data size is greater than 1,000 elements.

Primitive type keys and values

`ArrayMap` is faster than `HashMap` for insertion operations, but `ArrayMap` is slower for iteration, random query, and deletion operations. We observe that `ArrayMap` is, on average, 648 ms (19%) faster for insertion operations. However, `HashMap` is, on average, 239 ms (12%), 434 ms (12%), and 5,518 ms (47%) faster than `ArrayMap` for iteration, random query, and deletion operations, respectively.

Concerning memory usage, there is no clear trend and we consider that both implementations are memory efficient. However, `ArrayMap` uses, on average, less memory (5%) than `HashMap`.

In terms of energy efficiency, `ArrayMap` consumes less energy than `HashMap` for all the operations (on average 4%, 4%, and 2%, for insertion, iteration, and random query

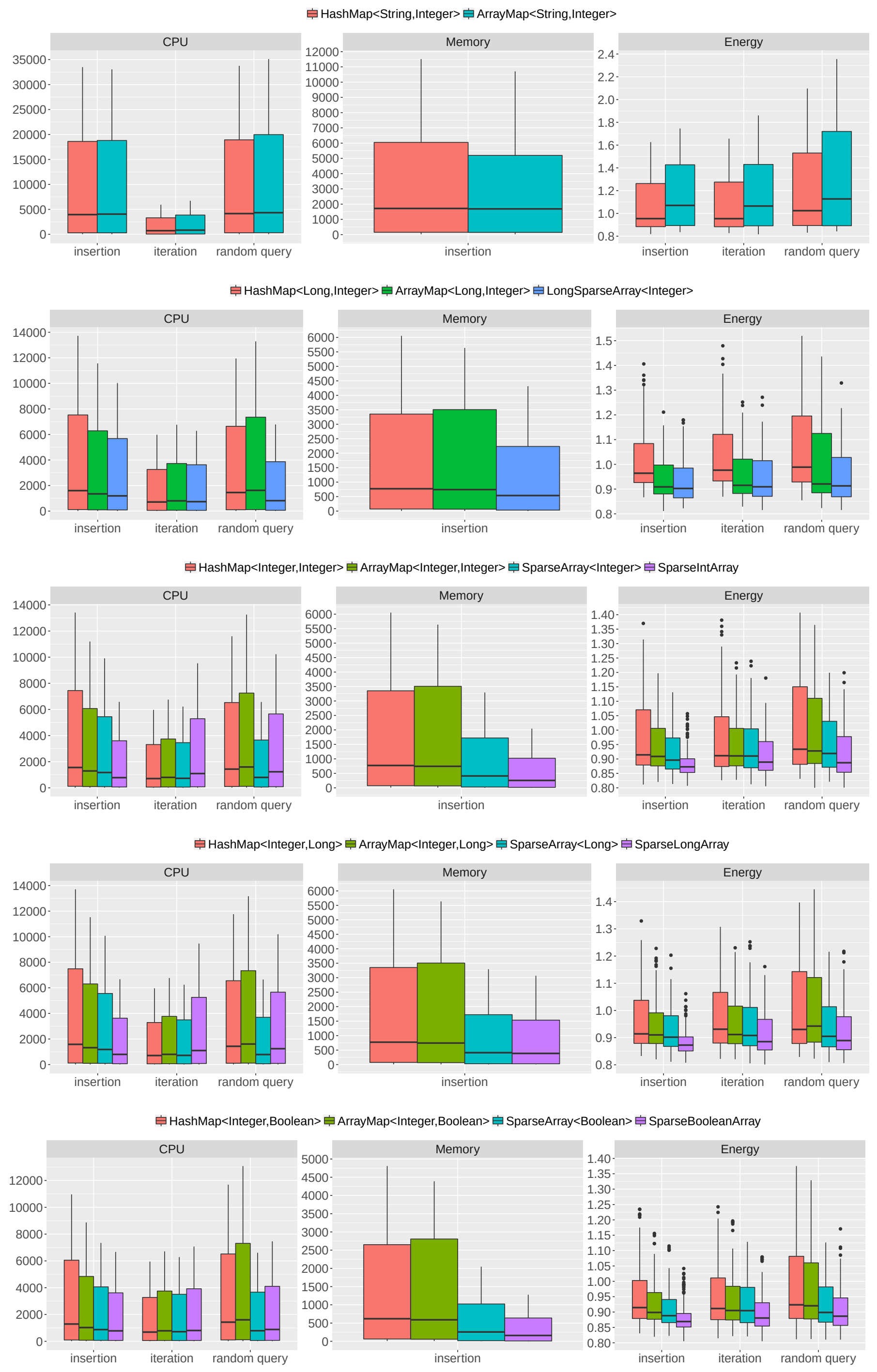

Fig. 6 Performance metrics of map implementations and map-related operations over 20 runs. CPU time, memory usage, and energy consumption are in ms, kB, and J, respectively.

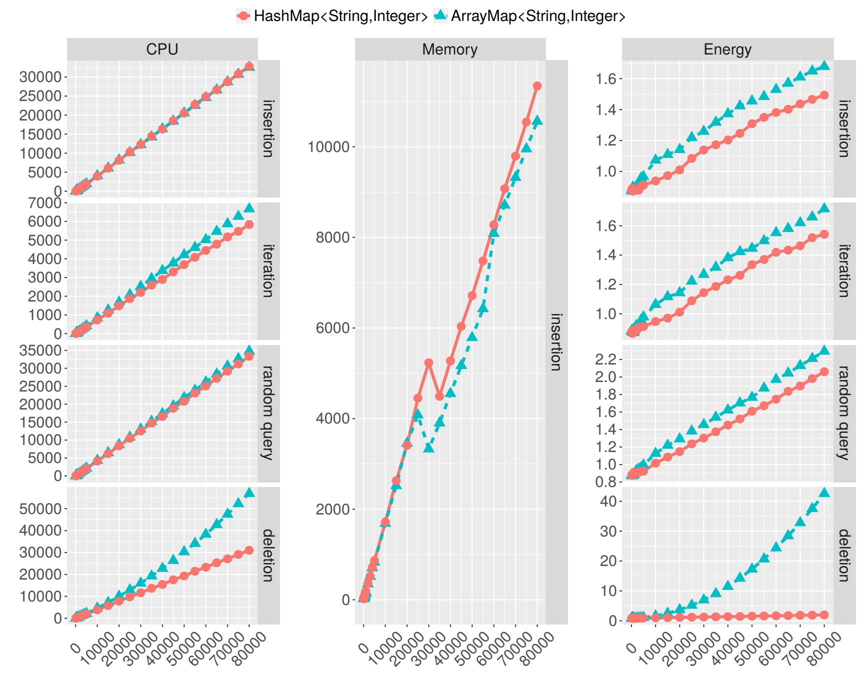

Fig. 7 Performance metrics of map implementations using string keys and integer values, by map-related operation and data size. CPU time, memory usage, and energy consumption are in ms, kB, and J, respectively.

operations, respectively). However, `ArrayMap` consumes, on average, much more energy (45%) than `HashMap` for deletion operations.

The Wilcoxon statistical test concludes that differences in CPU time and memory usage are significant and the effect size is large when the data size is greater than or equal to 100. Differences in energy consumption are significant and the effect size is large when the data size is greater than or equal to 20,000, for insertion and iteration operations. For random query operations, the statistical test reports that differences in terms of energy consumption are significant when the data size is greater than or equal to 30,000. For deletion operations, differences in energy consumption are significant when the data size is greater than 1,000 elements.

RQ4. What is the cost of adopting `HashMap` instead of `SparseArray` variants?

To answer this research question, we focus on the cost of adopting `HashMap` with primitive type keys instead of using `SparseArray` variants.

We find that `SparseArray` variants are faster than `HashMap` for insertion and random query operations. This trend also applies for deletion operations and `SparseArray` and `LongSparseArray`. Regarding iteration operations, `HashMap` is a bit faster than any `SparseArray` variant. We observe that `SparseArray` and `LongSparseArray` are, on average, 1,040 ms (27%), 1,503 ms (42%), and 1,746 ms (63%) faster for insertion, random query, and deletion operations, respectively. For iteration operations, `HashMap` is, on average, 79 ms (<1%) faster than `SparseArray` and `LongSparseArray`. Concerning `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, they are, on average,

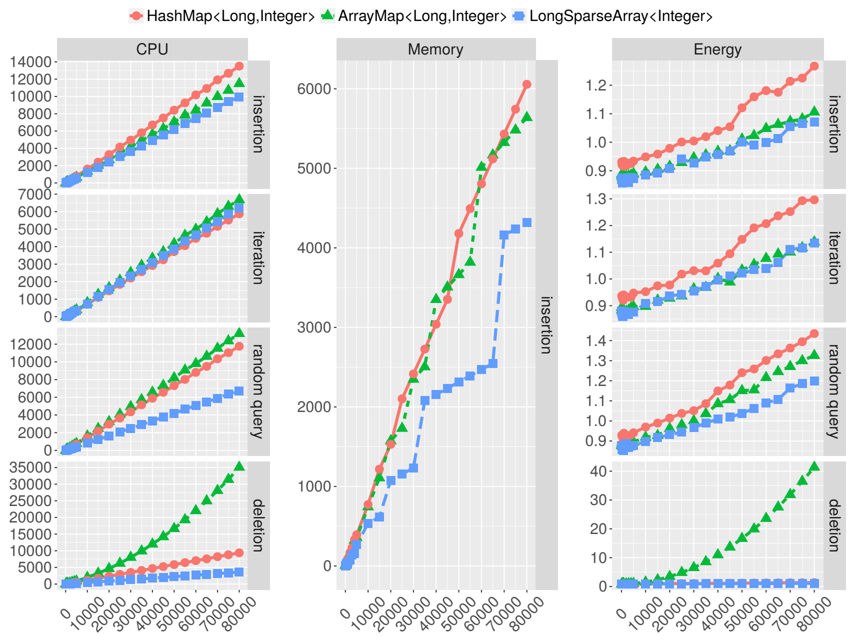

Fig. 8 Performance metrics of map implementations using long keys and integer values, by map-related operation and data size. CPU time, memory usage, and energy consumption are in ms, kB, and J, respectively.

1,768 ms (45%) and 725 ms (21%) faster for insertion and random query operations, respectively. However, HashMap is 782 ms (25%) faster for iteration operations. Regarding the deletion operation, we observe that SparseIntArray, SparseLongArray, and SparseBooleanArray are, on average, 63 ms (28%) faster than HashMap for data sizes lower than 20,000 elements. From 20,000 elements, HashMap is, on average, 5,569 ms (40%) faster for deletion operations.

In terms of memory usage, SparseArray variants are more efficient than HashMap for all data sizes. They use, on average, 1,025 kB (62%) less than HashMap.

Regarding energy consumption, SparseArray variants consume less energy than HashMap for insertion, iteration, and random query operations. This trend also applies for SparseArray and SparseLongArray and deletion operations. We find that SparseArray and LongSparseArray consume, on average, 6%, 4%, 7%, and 6% less energy than HashMap for insertion, iteration, random query, and deletion operations, respectively. Concerning SparseIntArray, SparseLongArray, and SparseBooleanArray, they consume, on average, 8%, 6%, and 8% less energy than HashMap for insertion, iteration, and random query operations. However, HashMap consumes 42% less energy than SparseIntArray, SparseLongArray, and SparseBooleanArray for deletion operations. Regarding the deletion operation, we observe that SparseIntArray, SparseLongArray, and SparseBooleanArray consume, on average, 2% less energy than HashMap for data sizes lower than 2,000 elements. However, for 2,000 or more elements, HashMap consumes, on average, 66% less energy for deletion operations.

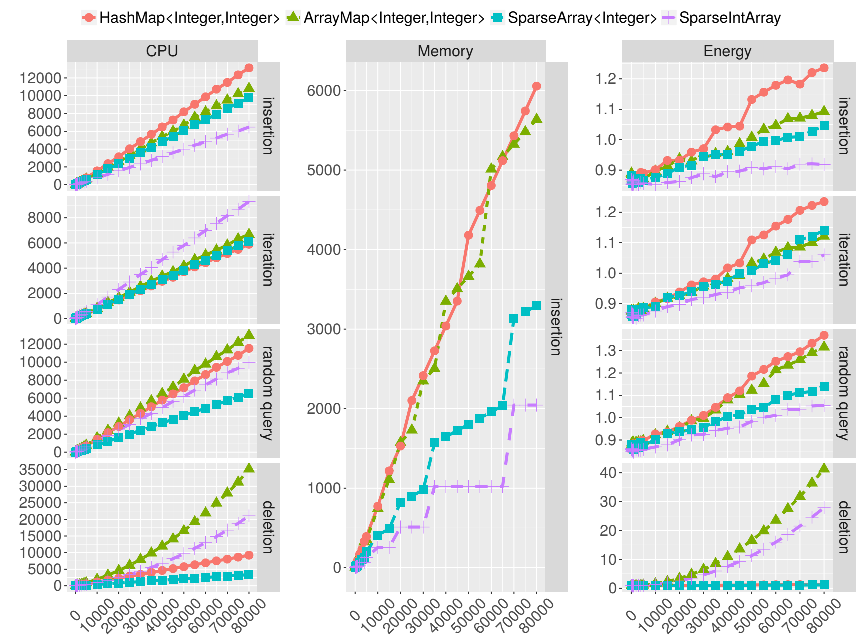

Fig. 9 Performance metrics of map implementations using integer keys and integer values, by map-related operation and data size. CPU time, memory usage, and energy consumption are in ms, kB, and J, respectively.

The Wilcoxon statistical test concludes that differences in CPU time and memory usage are significant and the effect size is large for all data sizes. Differences in energy consumption are significant and the effect size is large for most data sizes.

RQ5. Is `ArrayMap` as efficient as `SparseArray` variants?

To answer this research question we evaluate the cost of adopting `ArrayMap` with primitive type keys instead of using `SparseArray` variants.

We find that `SparseArray` variants are faster than `ArrayMap` for insertion, random query, and deletion operations. This also holds for iteration operations and `SparseArray` and `LongSparseArray`. We obtain that `SparseArray` and `LongSparseArray` are, on average, 393 ms (10%), 160 ms (13%), 1,938 (49%), and 7,264 ms (80%) faster than `ArrayMap` for insertion, iteration, random query, and deletion operations, respectively. Concerning `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, they are, on average, 1,118 ms (32%), 1,165 ms (31%), and 3,112 ms (47%) faster than `ArrayMap` for insertion, random query, and deletion operations, respectively. However, `ArrayMap` is 543 ms (14%) faster for iteration operations.

In terms of memory usage, `SparseArray` variants are more efficient than `ArrayMap` for all data sizes. They use, on average, 952 kB (60%) less than `ArrayMap`.

Regarding energy consumption, `SparseArray` variants consume less energy than `ArrayMap` for all the operations. We find that `SparseArray` variants consume, on average, 4%, 2%, 6%, and 32% less energy than `ArrayMap` for insertion, iteration, random query, and deletion operations, respectively.

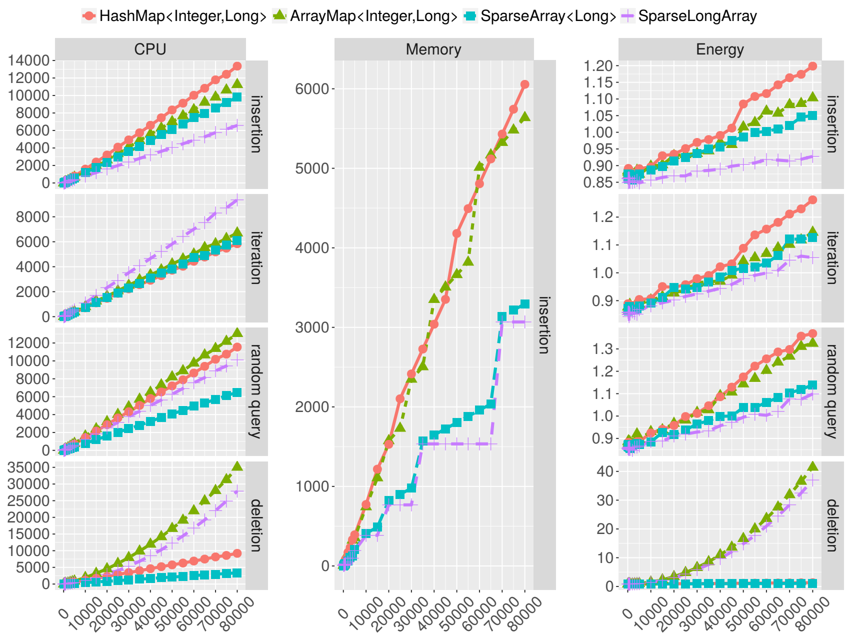

Fig. 10 Performance metrics of map implementations using integer keys and long values, by map-related operation and data size. CPU time, memory usage, and energy consumption are in ms, kB, and J, respectively.

The Wilcoxon statistical test concludes that differences in CPU time and memory usage are significant and the effect size is large for all operations and data sizes. Differences in energy consumption between `ArrayMap` and `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` are also significant and the effect size is large for all operations and most data sizes. For `ArrayMap` and `SparseArray` and `LongSparseArray`, differences in energy consumption are significant for most data sizes for random query and deletion operations. However, for insertion and iteration operations, differences are not always significant.

5.4 Discussion about the Experimental Study

We partially agree with the Android developers' reference documentation: `ArrayMap` is generally slower than `HashMap` since lookups require a binary search. However, it is not true for insertion operations for which we found that `ArrayMap` is faster no matter the number of elements. We also confirm that `ArrayMap` consumes, on average, less memory than `HashMap` (up to 6%). Although `ArrayMap` is more energy efficient than `HashMap` when keys are primitive types, we found that `ArrayMap` consumes more energy than `HashMap` for all the operations when keys are objects. The larger the number of elements, the larger the difference. We also observed that `ArrayMap` is highly inefficient for deletion operations. As it was shown in Figures 7, 8, 9, 10, and 11, while for insertion, iteration, and random query operations CPU time and energy consumption grew up linearly with respect to data size, for the deletion operation the growth was

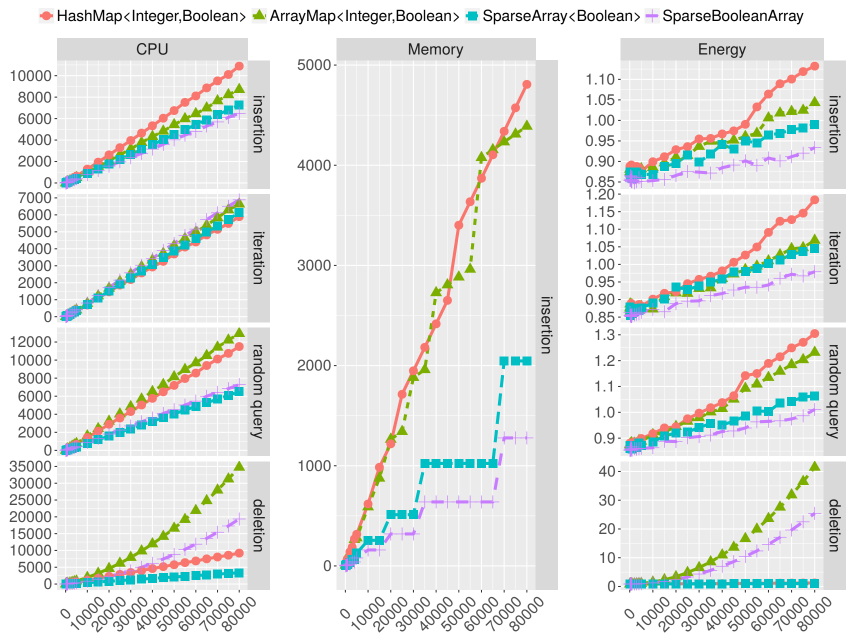

Fig. 11 Performance metrics of map implementations using integer keys and boolean values, by map-related operation and data size. CPU time, memory usage, and energy consumption are in ms, kB, and J, respectively.

exponential. The reason is that ArrayMap shrinks its array as items are removed from it. However, when an element is removed in a HashMap, the corresponding node in the entry set table is set to null. This means that HashMap does not shrink its entry set in deletion operations. This aggressive shrinking behavior for deletion operations in ArrayMap is justified in the Android developers' reference documentation to better balance memory use. Even if this fact is true, the impact of this decision on CPU time and energy consumption is not negligible.

We recommend to use HashMap instead of ArrayMap when keys are objects to improve the energy efficiency of Android apps.

If keys are primitive types, Android proposes SparseArray variants. SparseArray and LongSparseArray are faster than SparseIntArray, SparseLongArray, and SparseBooleanArray for iteration, random query, and deletion operations. The formers are also more energy efficient for deletion operations. However, SparseIntArray, SparseLongArray, and SparseBooleanArray use less memory than SparseArray and LongSparseArray, they are faster for insertion operations, and they are more energy efficient for insertion, iteration, and random query operations. Concerning deletion operations, we found that SparseIntArray, SparseLongArray, and SparseBooleanArray are inefficient. As it was shown in Figures 8, 9, 10, and 11, while for insertion, iteration, and random query operations CPU time and energy consumption grew up linearly with respect to data size, for the deletion operation the growth was exponential. The reason is that deletions require removing entries in the array of these map implementations. Conversely, `SparseArray` and `LongSparseArray` include an optimization when removing keys to help with performance. Instead of compacting its corresponding array immediately, it leaves the removed entry marked as deleted. The entry can then be re-used for the same key, or compacted later in a single garbage collection step of all removed entries. Considering our experiments, this optimization seems to make the difference between these map implementations for the deletion operation. However, we think that Google made this decision prioritizing memory usage over CPU time and energy consumption.

We advice to modify the implementation of `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` to include the same optimization used by `SparseArray` and `LongSparseArray`, to improve their performance when deleting elements.

We confirm what is claimed by the Android developers' reference documentation when primitive types are used as keys: `SparseArray` and `LongSparseArray` are more efficient than `HashMap` in terms of memory usage. But we also extend this fact to energy consumption. Contrary to the documentation, we found that `HashMap` is faster than `SparseArray` and `LongSparseArray` but only for iteration operations. For insertion and random query operations, they are faster than `HashMap` no matter the number of elements. We also observed that the higher the numbers of elements, the higher the differences in the three performance metrics in favor of `SparseArray` and `LongSparseArray` (except for the iteration operation for which `HashMap` is slightly faster). The same holds for `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` if values are also primitive types, excepting that these map implementations are faster than `HashMap` for iteration operations and they are less efficient than `HashMap` for deletion operations.

We recommend `SparseArray` and `LongSparseArray` over `HashMap` when keys are primitive types to improve the performance of Android apps. If values are also primitive types, and deletion operations are not usual, we recommend `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray` as a more efficient alternative to `SparseArray` and `LongSparseArray`.

Android proposes `SparseArray` variants as a replacement for `HashMap` when keys are primitive types for better performance. We confirmed that this proposition also holds for `ArrayMap`. Thus, `ArrayMap` should be replaced by `SparseArray` variants if primitive types are used as keys.

If primitive types are used as keys, we discourage the use of `ArrayMap` because `SparseArray` variants are more efficient. We strongly recommend that Android Studio suggests replacing `ArrayMap` by `SparseArray` variants when keys are primitive types.

To know whether performance differences are perceivable by end users when a more efficient map implementation is used, we conducted a new experiment. We randomly selected an app from our subject apps which contained occurrences of `HashMap` with primitive types as keys and values. Our guidelines suggest to use `SparseIntArray` as a more energy efficient alternative to `HashMap`. We selected the app *SudokuIsFun* because we could compile and run it on our phone. This is a simple Sudoku game with a simple user interface. We carried out an experiment comparing, for 30 independent runs, the energy consumption of the original version and the refactored one (by replacing `HashMap<Integer,Integer>` with `SparseIntArray`). We used a Python script to collect energy measurements automatically while we played a scenario that simulated the user interaction with the app. For each run, our script launched the app, played a scenario, and stopped the app. We used the `monkeyrunner`[21] Android tool to define the scenario that taps "NEW GAME", taps "Easy Puzzle #1", introduces five numbers in the first square of the Sudoku board, taps "Menu", and taps "Solve Puzzle*". Because both versions use exactly the same GUI, we consider that the delta in our energy measurements is due to the existing difference between these two versions of the app: the usage of `SparseIntArray` instead of `HashMap`. In 30 runs of this simple scenario that introduced some values and solved the Sudoku, the refactored version consumed less energy (median of 0.38 mJ). Regarding the voltage (3.8 V) and electric charge (2100 mAh) of the Nexus 4 phone battery, this reduction means that, if the battery is fully charged and a user repeats this scenario until the battery is over, the refactored version allows users to play two minutes and forty-nine seconds more. Therefore, replacing `HashMap` with `SparseIntArray` extended battery life by 0.81%. Although the improvement may seem rather marginal, it might be important enough for some developers to rethink their choices.

6 Guidelines

In our study, we observed that `HashMap` is the most used map implementation since most Android developers came from a Java developing background. Hence, Android specific map implementations are not always well-known and, because of that, they are not taken into account by developers. However, most developers are willing to replace `HashMap` with any of the map implementations provided by Android if they offer better performance. Figure 12 is a choice matrix summarizing our findings to help developers choose a map implementation. Green shades identify improvements in an operation and performance metric while yellow and orange shades mean worsening. We consider that an improvement/worsening is high/low if differences between map implementations and values of a performance metric are, on average, greater or equal/lower than 25%. A row with stronger shades of green is likely to be more efficient on average. We also specify a threshold in data size indicating whether our findings hold for more elements than a specific threshold. The symbol – indicates that, even if on average a map implementation is better than the other, there is no threshold for which this is always true.

[21] https://developer.android.com/studio/test/monkeyrunner/index.html

High improvement (≥25%) 🟩 Low improvement (<25%) 🟢 Low worsening (<25%) 🟨 High worsening (≥25%) 🟧

	CPU				Memory	Energy			
	Insertion	Iteration	Query	Deletion	Insertion	Insertion	Iteration	Query	Deletion
HashMap (objects)	🟨	🟢 any size	🟢 any size	🟢 any size	🟨	🟢 >= 2,000	🟢 >= 2,000	🟢 >= 2,000	🟩 >= 1,000
ArrayMap (objects)	🟢 any size	🟨	🟨	🟨	🟢 any size	🟨	🟨	🟨	🟧
HashMap (primitive types)	🟨	🟢 any size	🟢 any size	🟩 any size	🟨	🟨	🟨	🟨	🟩 any size
ArrayMap (primitive types)	🟢 any size	🟨	🟨	🟧	🟢 --	🟢 >= 15,000	🟢 >= 20,000	🟢 >= 30,000	🟧
HashMap (primitive types)	🟧	🟢 any size	🟧	🟧	🟧	🟨	🟨	🟨	🟨
SparseArray and LongSparseArray	🟩 any size	🟨	🟩 any size	🟩 any size	🟩 any size	🟢 any size	🟢 any size	🟢 any size	🟢 any size
HashMap (primitive types)	🟧	🟢 any size	🟨	🟩 >= 20,000	🟧	🟢	🟢	🟢	🟩 >= 2,000
SparseIntArray SparseLongArray SparseBooleanArray	🟩 any size	🟨	🟢 any size	🟧 >= 20,000	🟩 any size	🟢 any size	🟢 any size	🟢 any size	🟧 >= 2,000
ArrayMap (primitive types)	🟨	🟨	🟧	🟧	🟧	🟨	🟨	🟨	🟧
SparseArray and LongSparseArray	🟢 any size	🟢 any size	🟩 any size	🟩 any size	🟩 any size	🟢 --	🟢 --	🟢 --	🟩 > 600
ArrayMap (primitive types)	🟧	🟢 any size	🟧	🟧	🟧	🟨	🟨	🟨	🟧
SparseIntArray SparseLongArray SparseBooleanArray	🟩 any size	🟨	🟩 any size	🟩 any size	🟩 any size	🟢 any size	🟢 any size	🟢 any size	🟩 any size

Fig. 12 Color map showing the comparison between each pair of map implementations, operation, and performance metric. Green colors identify more efficient implementations. The greener the color, the better.

7 Threats to Validity

Threats to construct validity concern relationship between theory and observation and the extent to which the measures represent real values. For the observational study, we selected apps from GitHub because it is one of the most popular repositories with more than 14 million users and millions of open source projects. We focused on open source projects available in GitHub which had a link in the readme file to the official Android marketplace Google Play. We did that to select real Android apps. For the experimental study, we used a Nexus 4 phone which was used in other papers (Linares-Vásquez et al. (2014); Sahin et al. (2014); Huang et al. (2016); Saborido et al. (2016); Sahin et al. (2016)). Concerning CPU time, results were obtained using the execution traces generated by the Android profiler. As it is said in the Android documentation[22], because interpreted code runs more slowly when profiling is enabled, it is not correct to generate absolute timings from the profiler results. The times are only useful in relation to other profile outputs, and this is what we did for CPU time in our experiments. We chose the profiler rather than simple timers because we were not interested in real time but CPU time. Real time is the wall clock time, which would include time spent doing I/O and also time of other threads. CPU time is the time where a function is actually running. It does not include waiting on I/O or the time used by other threads that got CPU quantum. We used the Android profiler to record precisely CPU time. Regarding memory usage, we believe that our approach of measuring is precise. Although we

[22] https://developer.android.com/studio/profile/traceview.html

cannot control the garbage collector (GC) invocation, we think that the impact of this on our measurements is low since: (1) we ran 20 times and we took the median value and (2) the execution time of each experiment was small and it was unlikely the execution of the GC. In terms of energy, our measurement environment offered a higher or the same number of sampling bits as previous studies. In addition, our sampling frequency was one order of magnitude higher than past studies.

Threats to internal validity concern factors, internal to our study, that could have influenced our results. For the observational study, a Bash script that uses the `grep` command to search for occurrences of the map implementations under study. We manually validated the results and we found four different types of false positives: (1) apps that defined a data structure named `ArrayMap` which implements the `Map` interface, (2) apps that used a third-party library named `ArrayMap`, (3) apps that contained one or more strings containing text matching our patterns, and (4) apps that contained comments with text matching our patterns. We have manually checked true positives (TP), false positives (FP), and false negatives (FN) for each map implementation and projects under study. We obtained that, on average, the precision $\left(\frac{TP}{TP+FP}\right)$ was 93.46% and the recall $\left(\frac{TP}{TP+FN}\right)$ 100%. Thus, we consider that our Bash script performed well for our observational study, concerning the apps under study. Regarding the survey, it operated on a self-selection principle. This means that the results might be skewed towards developers who were willing to answer the survey, but avoiding the self-selection principle is not feasible in practice. Question-wording effect might have biased respondents towards one object if there was insufficient context when comparing different objects under the same conditions. To counteract the possible question-wording effect, we took care to make questions as specific and concrete as possible and used votes among the authors of this work to discard leading, loaded, or double-barreled questions. Considering the experimental study, we computed performance metrics using well-known techniques. In addition, we replicated several times our measures to ensure statistical validity.

Threats to external validity concern the generalization of our findings. For the observational study, we analyzed a big data set of Android apps but we limited our study to open source apps. Because of this, our findings could have been biased given that developers of selected apps could not be representative enough of Android developers. In order to increase the generality of our results, we encourage the research community to replicate this study regarding close source Android apps. This could be done processing their `apk` files and extracting and analyzing their bytecode. Concerning the survey, our selection of participants was constrained to the owner of the Android projects existing in GitHub. However, more than one developer could be involved in the development of an app. All of them could also be considered for surveyed to get a full view of the picture.

Threats to conclusion validity concern the relationship between experimentation and outcome. Analyses related to the use of map implementations in real Android apps and about performance metrics are supported by appropriate statistical procedures. However, our findings are based on the apps analyzed and on the data collected, which were limited to open source apps, from a unique source (GitHub), to the Nexus 4 phone, and to the Android version Lollipop[23]. We chose this Android version because it introduced one of the most significant changes in Android: the shift to the relatively new way of executing apps called Android Runtime (ART) that improves the

[23] https://developer.android.com/about/versions/android-5.0-changes.html

CPU usage and energy consumption of Android apps (Chen and Zong (2016)). From Android Lollipop, ART is the default runtime environment. Starting from Android Marshmallow[24], Android introduced two power-saving features that extend battery life for users by managing how apps behave when a device is not connected to a power source: (i) Doze, which reduces energy consumption by deferring background CPU and network activity for apps when the device is unused for long periods of time and (ii) App Standby, which defers background network activity for apps with which the user has not recently interacted. Android Nougat[25] brought further enhancements to Doze by applying a subset of CPU and network restrictions while the device is unplugged with the screen turned off. Android Oreo[26] limits certain behaviors by apps that are not running in the foreground to improve device performance. From the observations, we conclude that most performance optimizations in Android Marshmallow and later versions are oriented towards background tasks and, usually, when the device is unused for long periods of time. Yet, we cannot guarantee the generalization of our results for all Android versions although we have no reason to believe that our findings and claims would not be valid for Android Marshmallow, Android Nougat, and Android Oreo. Considering the information provided by Android Studio, by targeting Android Lollipop and later versions, apps will run on approximately 71.3% of the devices that are active on the Google Play Store (as of January 2018). For all the previous reasons, we think that our findings are valid for most of active devices. However, further validation on different phones is desirable to make our findings more generic. But there exist different factors (each one with several possible levels) to control, due to the fragmentation problem of Android. There are hundreds of different devices with different hardware configurations, more than 10 different versions of Android OS, and different ways of executing apps (Dalvik vs. ART).

Chen and Zong (2016) observed that Android apps developed in Java run much faster in ART and also consume much less energy than in Dalvik. Based on this, our claims for CPU time and energy consumption of map implementations could be considered as a lower bound in Dalvik. However, in that work, nothing is said about the impact of Dalvik and ART on memory usage. Consequently, we carried out additional experiments on a Samsung Galaxy S2 phone (running Android Jelly Bean and Dalvik) and measured the memory usage of map implementations for 10 runs. These experiments allowed us to compare the memory usage of map implementations for the Nexus 4 phone (running Android Lollipop and ART) and the Samsung Galaxy S2. We observed the same trend for both devices but differences were more notifiable, in favor of the Android map implementations, for the Samsung Galaxy S2. In general, `ArrayMap` used, on average, less memory than `HashMap` for any data type and data size. We concluded that our findings and claims are also applicable for older devices that use Dalvik instead of ART. Nevertheless, to complete the study, we should do a factorial experiment design taking into account all possible combinations of these levels across all factors, which seems unfeasible.

[24] https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html
[25] https://developer.android.com/about/versions/nougat/android-7.0-changes.html
[26] https://developer.android.com/about/versions/oreo/android-8.0-changes.html

8 Related Work

There are several works on the energy consumption of mobile apps. Liu et al. (2015) empirically studied the optimization space of application-level energy management from the data-oriented perspective. The energy optimization space is explored through multiple dimensions, ranging from data access pattern, data organization and representation, data precision, and data input/output intensity. Banerjee et al. (2014) defined an automated test generation framework that detects energy hotspots in Android apps. Each test input captures a sequence of user interactions that leads to a scenario that causes smart-phones to consume abnormally high amounts of energy in an app. Developers can focus on these hotspots to improve the energy-efficiency of their apps. Cuervo et al. (2010) proposed an approach that enables fine-grained energy-aware offload of mobile code to the infrastructure. Their approach decides at runtime which methods should be remotely executed, driven by an optimization engine that achieves the best energy savings possible under the mobile device's current connectivity constraints. Other works aimed at understanding the impact of users' choices on energy consumption (Zhang et al. (2014)) or energy consumption and network usage (Saborido et al. (2016)).

In addition to energy consumption, other performance metrics have also been studied. Chen and Zong (2016) conducted a comprehensive study on the impact of languages, compilers, Android runtimes, and implementations on the performance of Android apps in terms of energy consumption and CPU usage. Some studies also quantified the impact of ads on energy consumption, but also on CPU, memory, and network usages on Android apps (Gui et al. (2015); Saborido et al. (2017)).

Hasan et al. (2016) created detailed profiles of the energy consumed by common operations done on Java list, map, and set implementations. They also explored the memory usage of list implementations but they did not analyze the CPU usage. Other works studied the collection API of Java. Chameleon (Shacham et al. (2009)) is an automatic tool that assists developers in choosing the appropriate collection implementation for their apps. During program execution, it collects traces and computes heap-based metrics on collection classes to generate a list of suggested collection adaptation strategies. It is focused on CPU and memory usages. SEEDS (Manotas et al. (2014)) is the first known framework for helping developers make decisions regarding the energy consumption of their apps. It automatically optimizes Java apps by selecting the most energy efficient library implementations for Java's Collections API.

Although performance metrics of Java collections have been studied, none of the previous works has considered Android map implementations. We drew inspiration from this to perform an observational study and conduct a survey about Android map implementations. From this we concluded that Android map implementations are rarely used because of a lack of information about performance metrics for these implementations. This fact motivated our empirical study and, therefore, this research.

9 Conclusion

The Android API offers different implementations of the map data structure. These implementations are supposed to be improvements over the Java implementation `HashMap`. However, the Android developers' reference documentation does not provide precise information of the improvements (if any) in terms of CPU time, memory usage, and–or energy consumption.

We performed an observational study, conducted a survey, and carried out an experimental study. First, we analyzed 5,713 Android apps in GitHub and available in Google Play. We did that to study the use of Java and Android map implementations on real Android apps. Second, we conducted a survey to assess developers' perspective about Java and Android map implementations. Finally, we performed an experimental study on the CPU time, memory usage, and energy consumption of `HashMap`, `ArrayMap`, and `SparseArray` variants, for different data sizes and operations.

`HashMap` is the most used map implementation while `ArrayMap` and `SparseArray` variants are rarely used. Developers are moderately familiar with Android map implementations. However, they use `HashMap` because they are more familiar with it and because there is a lack of information about the Android map implementations and their performance.

We partially agree with the Android developers' reference documentation that `ArrayMap` is more memory efficient and generally slower than `HashMap`. However, for insertion operations, `ArrayMap` is faster than `HashMap` no matter the number of elements. Nevertheless, `ArrayMap` is less efficient than `HashMap` in terms of energy consumption. We recommend to use `HashMap` instead of `ArrayMap` to improve the energy efficiency of Android apps.

`HashMap` is often adopted with primitive type keys. In that cases, the official Android IDE, Android Studio, warns about replacing `HashMap` with `SparseArray` variants for better performance. `HashMap` with primitive type keys is less efficient than `SparseArray` variants in terms of memory usage. This fact is already claimed by the Android developers' reference documentation. However, we extend this to CPU time and energy consumption.

`ArrayMap` with primitive type keys is also less efficient than `SparseArray` variants. Thus, we recommend that Android Studio also warns about replacing `ArrayMap` by `SparseArray` variants when keys are primitive keys.

Although `SparseArray` variants are efficient alternatives to `HashMap` and `ArrayMap`, we advise to review the implementation of `SparseIntArray`, `SparseLongArray`, and `SparseBooleanArray`, which are highly inefficient when removing elements in comparison to `SparseArray` and `LongSparseArray`.

Overall, developers fail to follow Android recommendations due to the lack of precise information about performance. We suggest Google to complement the documentation of `ArrayMap` and `SparseArray` variants including more precise information about their CPU time, memory usage, and energy consumption for different operations. This will allow Android developers to make informed decisions about the map implementations that are the most suitable for their apps. We also encourage Android developers to follow our guidelines to adopt Android map implementations in their apps.

Future work includes assessing the feasibility of proposing semi-automated refactoring tools to detect uses of `HashMap` and–or `ArrayMap` implementations replacing them by `SparseArray` variants. We plan to study the impact of parameters capacity (of `HashMap`, `ArrayMap`, and `SparseArray` variants) and load factor (of `HashMap`) on performance metrics. We also want to use our guidelines to adapt Android apps, allowing them to dynamically decide which implementation to use depending on available resources (CPU speed, memory, and–or battery).

References

Banerjee A, Chong LK, Chattopadhyay S, Roychoudhury A (2014) Detecting Energy Bugs and Hotspots in Mobile Apps. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, FSE 2014, pp 588–598, DOI 10.1145/2635868.2635871, URL http://doi.acm.org/10.1145/2635868.2635871

Chen X, Zong Z (2016) Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation. In: 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom), pp 485–492, DOI 10.1109/BDCloud-SocialCom-SustainCom.2016.77

Cuervo E, Balasubramanian A, Cho Dk, Wolman A, Saroiu S, Chandra R, Bahl P (2010) MAUI: Making Smartphones Last Longer with Code Offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, ACM, New York, NY, USA, MobiSys '10, pp 49–62, DOI 10.1145/1814433.1814441, URL http://doi.acm.org/10.1145/1814433.1814441

Gui J, Mcilroy S, Nagappan M, Halfond WGJ (2015) Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers. In: Proceedings of the 37th International Conference on Software Engineering (ICSE)

Hasan S, King Z, Hafiz M, Sayagh M, Adams B, Hindle A (2016) Energy Profiles of Java Collections Classes. In: Proceedings of the 38th International Conference on Software Engineering (ICSE), Austin, TX, US, pp 225–236

Huang P, Xu T, Jin X, Zhou Y (2016) DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. In: Proceedings of the The 14th ACM International Conference on Mobile Systems, Applications, and Services, Singapore, Singapore, DOI 10.1145/2906388.2906419, URL http://doi.acm.org/10.1145/2906388.2906419

Li D, Hao S, Gui J, Halfond WGJ (2014) An Empirical Study of the Energy Consumption of Android Applications. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)

Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Oliveto R, Di Penta M, Poshyvanyk D (2014) Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR 2014, pp 2–11, DOI 10.1145/2597073.2597085, URL http://doi.acm.org/10.1145/2597073.2597085

Liu K, Pinto G, Liu YD (2015) Data-Oriented Characterization of Application-Level Energy Optimization. In: Egyed A, Schaefer I (eds) Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 316–331, URL https://doi.org/10.1007/978-3-662-46675-9_21, dOI: 10.1007/978-3-662-46675-9_21

Manotas I, Pollock L, Clause J (2014) SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp 503–514, DOI 10.1145/2568225.2568297, URL http://doi.acm.org/10.1145/2568225.2568297

Morales R, Saborido R, Khomh F, Chicano F, Antoniol G (2017) Earmo: An energy-aware refactoring approach for mobile apps. IEEE Transactions on Software Engineering PP(99):1–1, DOI 10.1109/TSE.2017.2757486

Saborido R, Beltrame G, Khomh F, Alba E, Antoniol G (2016) Optimizing User Experience in Choosing Android Applications. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)

Saborido R, Khomh F, Antoniol G, Guéhéneuc YG (2017) Comprehension of Ads-supported and Paid Android Applications: Are They Different? In: Proceedings of the 25th International Conference on Program Comprehension (ICPC), IEEE, Buenos Aires, Argentina, pp 143–153, DOI 10.1109/ICPC.2017.25

Sahin C, Tornquist P, Mckenna R, Pearson Z, Clause J (2014) How Does Code Obfuscation Impact Energy Usage? In: ICSME'14, pp 131–140

Sahin C, Pollock L, Clause J (2016) From Benchmarks to Real Apps: Exploring the Energy Impacts of Performance-directed Changes. Journal of Systems and Software pp –, DOI http://dx.doi.org/10.1016/j.jss.2016.03.031, URL `http://www.sciencedirect.com/science/article/pii/S0164121216000893`

Shacham O, Vechev M, Yahav E (2009) Chameleon: Adaptive selection of collections. SIGPLAN Not 44(6):408–418, DOI 10.1145/1543135.1542522, URL `http://doi.acm.org/10.1145/1543135.1542522`

Singer J, Sim SE, Lethbridge TC (2008) Software engineering data collection for field studies. In: Guide to Advanced Empirical Software Engineering, Springer, pp 9–34

Tyagi PK (1989) The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. Journal of the Academy of Marketing Science 17(3):235–241, DOI 10.1007/bf02729815, URL `http://dx.doi.org/10.1007/BF02729815`

Zhang C, Hindle A, German DM (2014) The Impact of User Choice on Energy Consumption. IEEE Software 31(3):69–75, DOI http://doi.ieeecomputersociety.org/10.1109/MS.2014.27

Rubén Saborido received his BS. degree in Computer Engineering and his MS. in Software Engineering and Artificial Intelligence from University of Malaga (Spain), where he worked for three years as a researcher. In 2017, he received a Ph.D. in Computer Engineering from Polytechnique Montréal and his thesis was nominated for best thesis award. Rubén research focuses on search based software engineering applied to performance and energy optimization of mobile devices. He is also interested in the use of metaheuristics to solve complex multi-objective optimization problems and in the design of algorithms to approximate a part of the whole Pareto optimal front taking into account user preferences. He has published seven papers in ISI indexed journals, and conference papers in MCDM, SANER, and ICPC. He co-organized the International Conference on Multiple Criteria Decision Making, in 2013.

Rodrigo Morales earned his BS. degree in Computer Science in 2005 from Polytechnic of Mexico. In 2008, he earned his MS. in Computer Technology from the same university, where he also worked as a professor in the Computer Science Department for five years. He has also worked in the bank industry as a software developer for more than three years (2009-2012). He received a Ph.D. in Computer Engineering from Polytechnique Montréal, and his thesis was nominated for best thesis award 2017. His research interests include software design quality, anti-patterns, and automated-refactoring, mobile software engineering. He has published several papers in ISI indexed journals and different conference papers. He is on the program committees of ICPC 2018 and ICSME 2018.

Foutse Khomh is an associate professor at Polytechnique Montréal, where he heads the SWAT Lab on software analytics and cloud engineering research (http://swat.polymtl.ca/). He received a Ph.D. in Software Engineering from the University of Montréal with the award of excellence. His research interests include software maintenance and evolution, cloud engineering, empirical software engineering, and software analytic. He has published more than 100 papers in international conferences and journals, and his work has received one Most Influential Paper Award, three Best Paper Awards and multiple nominations for Best Paper Awards. He has served on the program committees of several international conferences and reviewed for top international journals such as EMSE, TSE and TOSEM. He is on the Review Board of EMSE. He is program chair for Satellite Events at SANER 2015, program co-chair of SCAM 2015 and ICSME 2018, and general chair of ICPC 2018. He is one of the organizers of the RELENG workshop series (http://releng.polymtl.ca) and has been guest editor for special issues in the IEEE Software magazine and JSEP.

Yann-Gaël Guéhéneuc is full professor at the Department of Computer Science and Software Engineering of Concordia University since 2017, where he leads the Ptidej team on evaluating and enhancing the quality of the software systems, focusing on the Internet of Things and researching new theories, methods, and tools to understand, evaluate, and improve the development, release, testing, and security of such systems. Prior, he was faculty member at Polytechnique Montréal and Université de Montréal, where he started as assistant professor in 2003. In 2014, he was awarded the NSERC Research Chair Tier II on Patterns in Mixed-language Systems. In 2013-2014, he visited KAIST, Yonsei U., and Seoul National University, in Korea, as well as the National Institute of Informatics, in Japan, during his sabbatical year. In 2010, he became IEEE Senior Member. In 2009, he obtained the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. In 2003, he received a Ph.D. in Software Engineering from University of Nantes, France, under Professor Pierre Cointe's supervision. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM Ottawa Labs.), where he worked in 1999 and 2000. In 1998, he graduated as engineer from École des Mines of Nantes. His research interests are program understanding and program quality, in particular through the use and the identification of recurring patterns. He was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published papers in international conferences and journals, including IEEE TSE, Springer EMSE, ACM/IEEE ICSE, IEEE ICSME, and IEEE SANER. He was the program co-chair and general chair of several events, including IEEE SANER'15, APSEC'14, and IEEE ICSM'13.

Giuliano Antoniol received his Laurea degree in electronic engineering from the Universita' di Padova, Italy, in 1982. In 2004 he received his Ph.D. in Electrical Engineering at Polytechnique Montréal. He worked in companies, research institutions and universities. In 2005 he was awarded the Canada Research Chair Tier I in Software Change and Evolution. He has participated in the program and organization committees of numerous IEEE-sponsored international conferences. He served as program chair, industrial chair, tutorial, and general chair of international conferences and workshops. He is a member of the editorial boards of four journals: the Journal of Software Testing Verification & Reliability, the Journal of Empirical Software Engineering and the Software Quality Journal and the Journal of Software Maintenance and Evolution: Research and Practice. Dr Giuliano Antoniol served as Deputy Chair of the Steering Committee for the IEEE International Conference on Software Maintenance. He contributed to the program committees of more than 30 IEEE and ACM conferences and workshops, and he acts as referee for all major software engineering journals. He is currently Full Professor at Polytechnique Montréal, where he works in the area of software evolution, software traceability, search based software engineering, software testing and software maintenance.