An Empirical Study of the Energy Consumption of Android Applications

Ding Li, Shuai Hao, Jiaping Gui, William G.J. Halfond
Department of Computer Science
University of Southern California
Los Angeles, California 90089-0781
Email: {dingli, shuaihao, jgui, halfond}@usc.edu

***Abstract*—Energy is a critical resource for smartphones. However, developers who create apps for these platforms lack quantitative and objective information about the behavior of apps with respect to energy consumption. In this paper, we describe the results of our source-line level energy consumption study of 405 real-world market applications. Based on our study, we discover several interesting observations. For example, we find on average apps spend 61% of their energy in idle states, network is the most energy consuming component, and only a few APIs dominate non-idle energy consumption. The results of this study provide developers with objective information about how energy is consumed by a broad sample of mobile applications and can guide them in their efforts of improving the energy efficiency of their applications.**

***Keywords*—Mobile applications, Energy, Empirical study**

I. Introduction

Mobile devices, such as smartphones and tablets, have become incredibly popular and widely used in the daily lives of millions of users. Their popularity has been driven, in part, by the growth of a vibrant application ecosystem. Smartphone applications combine data from the Internet with a myriad of sensors to provide innovative and useful capabilities to end users. However, mobile devices are energy constrained as they rely on limited battery power supply. Although advances in battery technology, hardware components, and operating systems have helped to alleviate this limitation, these improvements cannot prevent applications from inefficiently or needlessly consuming a device's battery. Therefore, improving developers' ability to engineer energy efficient applications is essential to reduce the impact of smartphones' energy constraints.

Despite the importance of energy information, developers lack guidance on how energy is typically consumed by mobile applications. Our own research on energy consumption shows that there is a lack of objective and actionable information for developers. For example, developer blogs, such as the official Android developers' blog, offer "performance" tips on execution speed, which, as we show in this and prior work [1], [2], is not equal to energy consumption. Research work on energy has tended to focus on consumption patterns at the hardware level, which is beyond the control of most software engineers. Other works, which are more software systems oriented, focus very narrowly on specific problems, such as wake-lock bugs [3], distributed programming abstractions [4], energy bug comparison [5], and configuration [6]. Unsurprisingly, as a result, developers' success in writing energy efficient applications has been mixed. Those that have been unsuccessful are frequently the target of complaints about battery consumption in marketplace reviews [7]. These complaints can have serious ramifications for developers, as they can influence users' app purchasing decisions. Therefore, energy consumption has become an important quality metric for marketplace applications.

To better understand energy consumption in mobile devices, we performed an extensive study of the energy related behavior of smartphone applications. In this study we measured and analyzed the energy consumption of over 400 real-world marketplace applications downloaded from the Google Play market. Furthermore, we leveraged state of the art energy measurements tools to provide measurements at different levels of granularity, from the whole application level to the source line and instruction level. To the best of our knowledge, our work is the first to combine such a large sample size with detailed source line level information.

Our study reveals several interesting observations that provide actionable guidance for software engineers and motivate areas for future research work in the software engineering community. First, we find that most applications spend more than 60% of their energy in an idle state, which is the state where no code of the current application is running, so only optimizing the code of applications is not sufficient to reduce the overall app energy consumption. Second, we find that the network is the most energy consuming component in Android applications. In particular, making an HTTP request is the most energy consuming operation of the network. Third, we find that the energy consumption of applications is dominated by that of system APIs. Thus, developers should focus on the optimization of the use of system APIs. Fourth, we find that, despite the large number of APIs used in applications, only a few are significant in terms of energy consumption. Hence, developers could narrow the range of APIs that need to be optimized to a small set. Fifth, we find that instructions in loops typically consume over 40% of the total non-idle energy. Lastly, we find that data manipulation operations are the most energy consuming among all types of bytecodes. Taken together, these results provide useful information about the energy consumption of Android applications to developers and may help them design better schemes for energy optimization.

In addition to energy consumption insights, we also evaluate the impact on the accuracy of three typical energy measurement practices that have appeared in related work. These practices are: (1) using time as an approximation for

energy; (2) using millisecond level measurements; and (3) neglecting to account for idle state energy. Our analysis results show that these practices can lead to inaccurate measurements in energy consumption studies.

The rest of this paper is organized as follows. In Section II, we motivate our list of research questions. In Section III, we introduce the energy experiment protocol of our measurements. In Section IV, we report and discuss the results of our measurements. In Section V, we discuss the threats to the validity of our experiments. In Section VI, we describe the related work. Finally, we conclude the paper in Section VII.

II. Research Questions

We focus on two aspects of the energy consumption in Android applications. The first aspect is how energy is consumed. This aspect includes several research questions, such as what kind of applications are the most energy consuming, how much energy is actually consumed by the applications, which are the most energy consuming components, etc. We answer these research questions in a top-down order. We first answer the research questions on the application level, and then move to the component level, and finally move to the source line level for APIs and instructions. The second aspect is how to measure the energy consumption of applications. Measurement is the basis for energy optimization. Inaccurate measurements can undermine the validity of results. Thus, we analyze the impact of three typical practices in energy studies on the accuracy of energy consumption results.

For the first aspect, the most high-level research question on the application level is **RQ 1: How much energy is consumed by individual applications?** This research question gives a high level picture of how energy is consumed by different applications and what kinds of applications are the most energy consuming. This information provides developers with a general overview of the energy consumption of a broad sample of applications.

Next, we break down the energy consumption of each application. The second research question is **RQ 2: How much energy is consumed by the idle state of an application?** Android applications are event based, multithreaded, and sensor intensive. Applications may be suspended at times to wait for user input, sensor data, or thread synchronization. However, the system will still consume energy during this idle time. The amount of energy consumed in an idle versus non-idle state is important as most code-oriented optimization will not affect energy consumed when an app is idle.

The last research question on the application level is **RQ 3: Which code consumes more energy: system APIs or developer-written code?** In this research question, we break down the non-idle energy even further. Android applications are composed of system APIs and developer created code. So, the ratio of the energy spent in APIs or developer created code will influence the energy optimization strategy. If the system APIs dominate the non-idle energy, the energy optimization scheme should focus on the usage of system APIs. Otherwise, developers should focus on the optimization of their code.

On the component level, our research question is **RQ 4: How much energy is consumed by the different components of a smartphone?** Current Android smartphones have multiple software and hardware components that provide different functionalities. These components include UI, network, IO, Sqlite, location, camera, media, and sensors. The applications can use a subset of these software and hardware components to accomplish their tasks. By knowing how energy is distributed in different components, developers can be better informed in designing their apps and optimizing energy usage.

On the source line level, the first question we answer is **RQ 5: Which APIs are significant in terms of energy consumption?** Some of the APIs may consume a lot of energy and dominate the total energy consumption, while other APIs may have trivial energy consumption. To address this question, we answer three subquestions. The first one is *How many APIs are significant in energy consumption?*. This subquestion provides information about the quantity of energy significant APIs. The second one is *How similar are the top ten most energy consuming APIs across different applications?*. This subquestion asks if there are common patterns in the energy significant APIs across different applications. If so, developers can design specific optimization schemes for these patterns. The third one is *Which APIs are the most likely to be in the top ten most energy consuming APIs?*. This subquestion asks which APIs are likely to be energy significant. Based on this study, developers are better informed when they consider which API usages should be optimized.

Besides the energy distribution in APIs, the second research question at the source line level is **RQ 6: How much energy is consumed by code in loops?** Loops are common code structures to do repeating tasks. It is useful to know whether they also play an important role in energy consumption. Specifically, we want to know whether there are often energy consuming APIs in loops. If so, these loops may be good targets for optimization.

The third research question at the source line level is **RQ 7: How much energy is consumed by the different types of bytecodes?** Besides invocations of system APIs, the developer-written code is another major part of an Android application. However, it is not clear what aspects of this code are the most energy consuming. With better knowledge of the energy consumption of the different underlying instructions, developers could better plan their design and create more energy efficient applications.

In the second aspect, we study three typical practices in energy measurement studies. These three methods are commonly used but have limitations that could introduce inaccuracy into the measurement. We evaluate how much inaccuracy they can introduce and help developers decide what method of measurement they need.

Our first research question on energy measurement is **RQ 8: Is time equal to energy?** Traditionally, developers use the execution time of methods to optimize the performance of applications. However, it may not be sufficient for energy optimizations and may be misleading. Knowing how accurately time can approximate energy would allow developers to determine what kind of information they need to use to carry out the most effective optimizations.

The second research question on energy measurement is **RQ 9: What granularity of measurement is sufficient?**

Today, developers have many energy measurement tools to help diagnose energy issues. However, many of them are only on the millisecond level. The processors of modern smartphones are generally in the GHz range, which means millions of operations can be executed in one millisecond. Simply using the millisecond level measurement may miss a lot of details and introduce inaccuracy into the energy diagnosis. The information about how much inaccuracy millisecond level measurements could cause can help developers make the appropriate decision for energy measurement and optimization.

Our last research question on energy measurement is **RQ 10: Is it necessary to account for idle state energy?** In many recent measurement studies, developers do not consider the energy consumption during idle states, where no code of the current application is running. This method can inaccurately inflate the results of measurements because it includes energy that is not consumed by the code of an application. If the idle state of applications consumes a significant amount of energy, the measurement results could be misleading. Knowing whether the idle state energy is a problem can also help developers measure and optimize the energy of their applications correctly.

III. General Experiment Protocol

In this section, we describe our protocol for obtaining the data used to address the research questions posed in Section II. These questions require two key capabilities: obtain source line level energy information of mobile applications and automate the interaction with the subject applications.

To obtain source line level information, we leveraged the capabilities provided in our prior work, *vLens* [8]. The *vLens* method combines hardware-based energy measurements with program analysis and statistical modeling techniques to provide source line level energy information. At a high level, the basic intuition of *vLens* is as follows: while measuring the energy consumption of a smartphone, *vLens* uses efficient path profiling (EPP) to identify which paths in the application are being executed and correlates these paths with the measured energy samples. Then, *vLens* statically analyzes the paths to identify and adjust for high-energy events, such as thread switching, before applying robust regression analysis [9] to calculate each source line's energy consumption. Finally, *vLens* provides a mapping of the energy consumption to the source code of the application.

In this study, we modified the power measurement platform (PMP) of *vLens*. The PMP was originally based on an x86 based Android platform [10]. In order to run non-x86 applications, we built a new PMP based on the Monsoon power meter [11]. The Monsoon power meter samples the energy consumption of the smartphone at a frequency of 5KHz and synchronizes every energy sample with the standard Unix time.

In order to get the nanosecond level energy information, we instrumented with both millisecond (`System.currentTimeMillis`) and nanosecond level timestamps (`System.nanoTime`). We used the millisecond level timetamps, which are synchronized to the Unix time, to match executed code blocks to their corresponding energy measurements. If a code block was executed over a long enough time duration, we summed up the energy samples between the start and end millisecond level timestamps of the code block and used that value as its energy consumption. If the code block was executed in a small enough period of time that could not be accurately captured by millisecond level timestamps, we used the nanosecond level timestamps (`System.nanoTime`) to measure its time span and calculated its energy consumption with function $P * \Delta t$, where P is the power of the smartphone when the target code block is executed and Δt is its execution time measured by the nanosecond level time stamps.

For the purpose of this study, the Monsoon based PMP not only provides the same capabilities as the LEAP based PMP, but also allows *vLens* to work with any modern Android platform, and by extension, a broader set of mobile applications. In this study, we used the Samsung Galaxy SII smartphone with a rooted Android 4.3 operating system.

To facilitate the automated interaction with the applications, we leveraged a workload generation tool, Monkey [12], from the Android SDK. Monkey is analogous to a traditional web crawler, but interacts with the GUI of Android applications, and generates a pseudo random stream of user events and inputs, such as touch and scrolling, for the target application. Monkey runs on a controller desktop and pushes UI events to the smartphone through the network. For each of our applications, Monkey generated five UI events per second and 500 events in total. Due to the subject pool size of our study, an important feature of this tool was that it could be completely automated to interact with the application. Also the tool uses pseudo random numbers to generate each user input, thus its workload was repeatable in every experiment for a single application with the same random number seed.

To build a pool of subject applications, we started by collecting 9,644 free applications from the Google Play market using the Google Play Crawler [13]. These applications came from over 23 different categories, but did not include games because they typically require non-deterministic and complex operations that could not be generated automatically. The *vLens* tool only supports instrumenting bytecode for path profiling, so we also excluded any applications that required native libraries. We then randomly selected 412 applications for our experiment. Finally, we removed the applications for which Monkey had generated statement coverage of less than 50% or that crashed during the automated interaction. This left us with 405 applications. The statistics and categories of these 405 applications are shown in Fig. 1. In this figure, we list the eight categories with the highest number of applications and all other categories are reported as "others." Each category in Fig. 1 has at least 28 applications and none of them have more than 74 applications.

IV. Evaluation

A. RQ 1: How much energy is consumed by individual applications?

To address this research question, we measured the total energy consumed by each of our subject applications during its execution. The total energy of an application is calculated as the sum of all energy consumed between the earliest timestamp and the latest timestamp associated with the application's

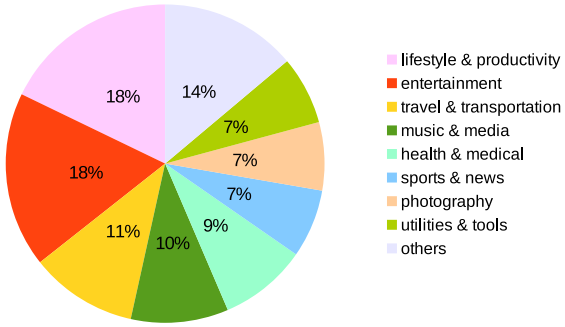

Fig. 1: Categorization of the subject applications.

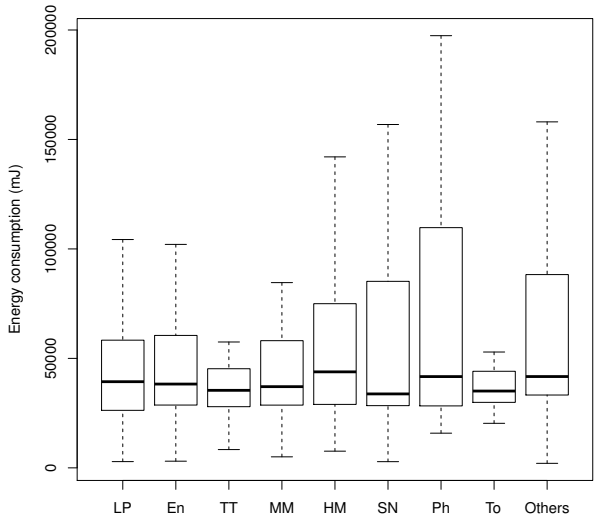

Fig. 2: Average application energy consumption by category.

execution. We analyze the results of all the applications as a group and then by the categories shown in Figure 1.

On average, each application consumes 57,977 mJ of energy during its execution. 81% of the applications fall into the range of 10,000 mJ to 100,000 mJ. The standard deviation of all applications is 62,416 mJ, and the difference between the highest and the lowest energy consumption is several orders of magnitude. The energy consumption by categories is shown in Fig. 2: lifestyle & productivity (LP), entertainment (En), travel & transportation (TT), music & media (MM), health & medical (HM), sports and news (SN), photography (Ph), tools (To), and other (Other).

These results show that energy consumption patterns are not homogeneous. The energy of different applications in the same category can vary by several orders of magnitude. Further, the category of an application does not have a strong correlation with its energy consumption. The average energy of applications in different categories are at most 30% different. Compared with the difference in the energy consumption of applications within each group, which can be as large as several hundreds times, this amount of difference is not significant.

B. RQ 2: How much energy is consumed by the idle state of an application?

To answer this research question, we divide the general energy usage of applications into three categories, PureIdle, APIIdle, NonIdle, and compare their energy consumption. PureIdle is the energy that is consumed with no code of the application running. For example, the running application may be suspended to wait for user input or asynchronous sensor data. During this time, there is no code of the application being executed but the system still consumes energy. APIIdle is the energy consumed by the sleeping APIs, such as `java.lang.Thread.sleep` and `java.lang.Object.wait`. These APIs set the running application to idle state, but unlike the PureIdle, these APIs are a part of the application code. Finally, NonIdle is the amount of energy that is actually consumed by the application code. This amount of energy includes all energy consumed by the non-sleeping APIs and all the user code of the running application.

We calculate each of these three categories as follows: First, we calculate APIIdle as the sum of the energy for all `java.lang.Thread.sleep` and `java.lang.Object.wait` series of APIs. The energy of an API is the sum of energy samples between its starting and ending timestamps. Second, we calculate the NonIdle energy as the sum of the energy of all execution paths minus the APIIdle energy. The energy of a path is the sum of all energy samples between the entry and exit timestamps of the path. This includes the energy of sleeping APIs called along the path so we subtract their energy from the summed value to get NonIdle. Lastly, we calculate PureIdle as the total energy minus APIIdle and NonIdle, where the total energy for an application is the same as in Section IV-A. PureIdle represents all the energy that has been consumed while the application is running but not caused by any code of the application.

On average the PureIdle, APIIdle, and NonIdle consume 36.6%, 25.0%, and 38.4% respectively. Their standard deviations are 37.8%, 30.0%, and 30.8% respectively. These numbers indicate that, for many applications, the code does not play a dominant role in energy consumption. In our study, in half of the applications, code consumes less than 31.1% of the total application energy. Most of the energy is spent as idle energy, either as PureIdle or APIIdle. Thus, simply optimizing the energy consumption of user code is insufficient; developers also need to reduce the energy consumed during idle states.

One possible way to save energy is to design an energy efficient color scheme for mobile applications. Many current smartphones, such as the Samsung Galaxy series, use OLED screens as a display device. For these OLED based smartphones, a well designed energy efficient color scheme can reduce the energy consumption of mobile applications in the idle state. According to previous research [14], a well designed color scheme could save up to 72% of the OLED screen power compared to energy inefficient color schemes. In our previous work [15], we developed a technique to automatically transform apps to use a more energy efficient color scheme,

which resulted in power savings averaging 40% per app. Combining the fact that half of the applications spend more than 69% of their energy in the idle state, developers should first optimize their design of colors before diving into the optimization of their code.

C. RQ 3: Which code consumes more energy: system APIs or developer-written code?

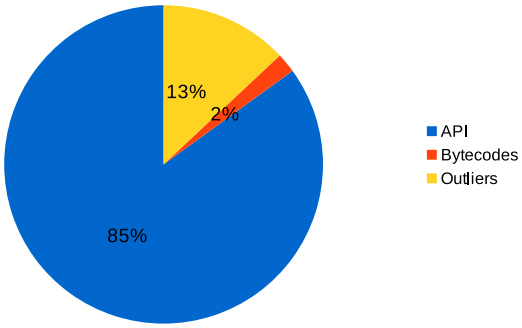

Fig. 3: The breakdown of non-idle energy

To answer this question, we break down the NonIdle energy in Section IV-B into three categories: API, Bytecodes, and Outliers. API is the energy of invoking any API in the Android SDK, such as *android.bluetooth.BluetoothA2dp.finalize()*. Bytecodes represents the energy consumption of normal user code, such as branch instructions and arithmetic instructions. Outliers is the energy introduced by system events, such as garbage collection and process switching. During the execution of an application, the Android system may interrupt the current application to schedule a garbage collection or another process. Events like garbage collection and external process switching are controlled by the system and the application has no knowledge as to when they will happen. If the garbage collection and external process switching interrupt an execution path, their energy will be included in the measured energy of the execution path. According to our former research [8], the cost of a garbage collection or external process switching is 10 to 10,000 times larger than a normal path. Thus, if a path is interrupted by a garbage collection or external process switching, its energy consumption will be abnormally large.

In our measurement, we calculate the API energy as the sum of the energy of all API calls except for calls to the sleeping APIs, which were identified in Section IV-B. Similar to Section IV-B, the energy of each API is the sum of all energy samples between its starting and ending time stamps. We calculate the energy consumption of each bytecode with the robust regression techniques introduced in *vLens* [8]. The Bytecode energy is the total number of bytecodes multiplied with the per-bytecode cost identified via the robust regression. The energy of Outliers is the NonIdle energy minus the API energy and the Bytecode energy.

The results of this analysis are shown in Fig. 3. As shown in the figure, 75% of applications consume more than 82.2% of their energy via API invocations, and 91.4% of applications consume more than 60% of their total energy via API invocations. These results indicate that the system APIs dominate non-idle application energy for most applications. The user code does not consume a lot of energy. Thus, developers should focus on optimizing their usage of APIs to reduce the energy of their applications.

D. RQ 4: How much energy is consumed by the different components of a smartphone?

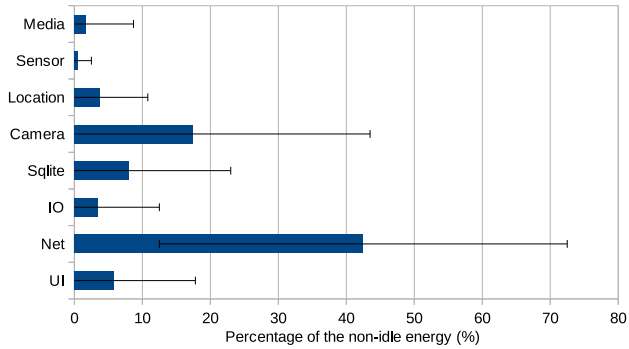

Fig. 4: The component level energy usage

To address this question, we report the ratio of the energy consumed by eight components of a smartphone to the total non-idle energy consumption of the test applications. These eight components are UI, network (Net), I/O operations (IO), sqlite queries (Sqlite), camera related operations (Camera), location information (Location), sensor accessing (Sensors), and multimedia (Media). They represent the major functionalities of Android applications. For example, an Android application may read some data from the sensors and then display it on the screen through the UI or share it through the network. Sqlite, Camera, Location, and Media are also commonly used in Android applications. Since the hardware and software components of Android smartphones have to be accessed through APIs, we use the energy consumption of APIs that access a component as the energy consumption of the component. In our measurement, each of the eight components includes one or more packages of Android APIs, for example, the Net includes packages such as *android.net.** and *java.net.**.

The result of this study is depicted in Fig. 4, where the x-axis is the average percentage of the component level energy to the total non-idle energy. The average value is calculated only with the applications that use a certain component. If one application does not use a certain component, it will be excluded from the statistics of the component. For example, there are 330 applications using the network, so we calculate the average energy of Net only for these 330 applications.

The data indicates that network is the most energy consuming and frequently used component in our measurement. Thus, the usage of network operations should be the optimization priority for reducing an app's energy consumption. Other components have less energy consumption because they are not used as frequently as the network. Another observation is that the standard deviations are significant compared with the average value, which means the energy consumption of each component varies widely from application to application. Thus, despite the low average energy consumption of some components, they can still be significant in certain applications.

Breakdown of Net: Since Net is the most energy consuming category of APIs in most applications, we go one step further to break down the energy of Net. There are three mechanisms in Android to use the network, HTTP requests, Socket connections, and Webkit to display web pages. Thus, we divide the energy of Net into three categories that represent each one of the methods to access network. The average ratio of HTTP, Socket, and Webkit are 80%, 0.27%, 10%, respectively. Furthermore, 75% of the application spend more than 89% of their network energy in HTTP. This result indicates that making HTTP requests is the most energy consuming network operation. Thus, when considering the network, operations on HTTP requests should be a primary focus.

E. RQ 5: Which APIs are significant in terms of energy consumption?

We answer this research question with three subquestions: (1) How many APIs are significant in energy consumption? (2) How similar are the top ten most energy consuming APIs across different applications? and (3) Which APIs are the most likely to be in the top ten most energy consuming APIs?

1) How many APIs are significant in energy consumption?: To answer this subquestion, we calculated the distribution of the average ratio of the energy consumed by each API to the total non-idle energy of all applications (Ratio) and the ratio of the top ten most energy consuming APIs to the total non-idle energy of each application (Top10). Ratio is a metric for each API that we use to evaluate how many APIs are significant across all applications. However, the ratio of an API can be distorted by the frequency of the API used in different applications. For example, an API may be the most energy consuming API when it is invoked, but if it is only used in a few applications, its Ratio may be very low. To give a complete view of this research question, for each application, we also calculate its Top10 to evaluate how many APIs are significant in the application. Top10 is the metric for each individual application and will not be distorted by the invocation frequency of APIs in different applications.

For an API API_i, its Ratio is given by the equation: $Ratio_i = \frac{\sum_{j=1}^{N} RA_j^i}{N}$, where RA_j^i is the ratio of the energy of API_i to the total API energy of the jth application. N is the number of applications, which is 405 in our experiment. For an application App_i, its Top10 value is given by the equation: $Top10_i = \frac{\sum_{j=1}^{10} EA_j^i}{E_i}$, where EA_1 to EA_{10} are the energy consumption of the top 10 most energy consuming APIs in App_i and E_i is the non-idle energy of App_i.

There are 7,784 unique APIs used in our 405 applications. On average, each application invokes 292 different APIs. In our measurement, 98.4% unique APIs have a Ratio value below 0.1%. There are 11 APIs that have a Ratio larger than 1% and three APIs that have a Ratio larger than 4%. For the Top10 value of each application, the average is 76.4%. Furthermore, in 91.1% applications, the Top10 value is at least 50%.

This data indicates that only a few APIs are significant in the energy consumption. Across all applications, the Ratio shows that, on average, only 2% of APIs consume more than 0.1% of the non-idle energy. For each application, the Top10 value shows that the top ten most energy consuming APIs out of the average 292 APIs per each app consume more than 3/4 of the app's total non-idle energy. Thus, developers can focus on a small set of APIs when they optimize the energy consumption of their applications.

2) How similar are the top ten most energy consuming APIs across different applications?: To answer this subquestion, we measured the overlapping of the top ten most energy consuming APIs in each pair of applications. We calculated the size of the intersection of the top ten most energy consuming APIs for each pair of applications. Since we have 405 applications, there are $405 * (405 - 1)/2 = 81810$ pairs of applications. On average, the size of the intersection of the top ten most energy consuming APIs is 1.093 and the median is 1. This number indicates that different applications consume energy in a different manner. There is not a general pattern among the top ten most energy consuming APIs across all applications. So, there is not a universal approach to optimizing the energy consumption for all applications. Developers have to design specific energy efficient schemes for each applications.

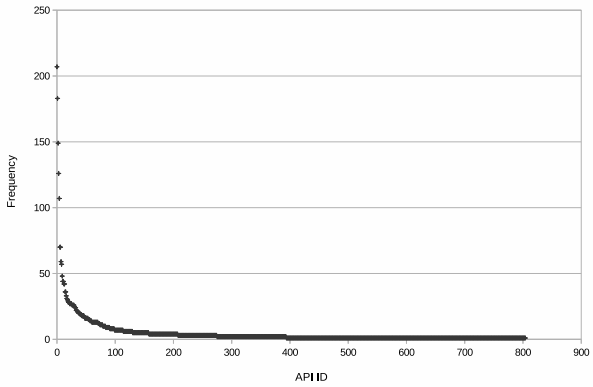

Fig. 5: The frequency distribution of the top ten most energy consuming APIs

3) Which APIs are the most likely to be in the top ten most energy consuming APIs?: To answer this subquestion, we plot the frequency distribution of each API that falls into the set of the top ten most energy consuming APIs. To be more specific, we calculate, for each API, how many times it falls into the top ten most energy consuming APIs of the test applications. The result is shown in Fig. 5, where the x-axis is the API ID and the y-axis is the frequency of a certain API. There are five APIs that are in the top 10 energy consuming APIs in more than 100 applications. Among these five APIs, four are related to making HTTP requests or viewing web contents and one is used to synchronize shared files between threads. These results indicate that most APIs do not frequently appear in the set of the top ten most energy consuming APIs. Only a few APIs, such as the APIs that make HTTP requests and share files between threads, are most likely to be the most energy consuming in different applications.

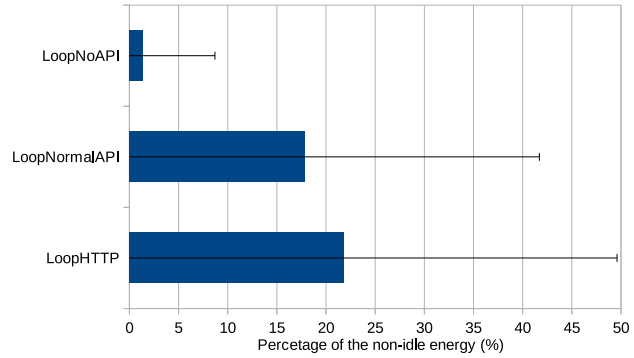

Fig. 6: The distribution of loop energy consumption

F. RQ 6: How much energy is consumed by code in loops?

To address this research question, we studied three different types of loops: loops with HTTP requests (LoopHTTP), loops with other APIs (LoopNormalAPI), and loops with no APIs (LoopNoAPI). We studied LoopNormalAPI and LoopHTTP separately from other loops because APIs often dominate the energy consumption of an application and making HTTP requests often dominates most of the API energy. The result is plotted in Fig. 6.

From these results, we find that loops are significant in terms of consuming energy in applications. On average, instructions in loops consume 41.1% of total non-idle energy. Among all loops, those with HTTP related APIs are the most energy consuming. This result is consistent with the conclusion of Section IV-D — making HTTP requests is the most energy expensive operation. Another observation is that the standard deviation of each type of loop is larger than the average. This indicates that the energy consumption of loops varies significantly across different applications. This is because the functionalities of different applications are different and they may use loops in very different ways as well.

G. RQ 7: How much energy is consumed by the different types of bytecodes?

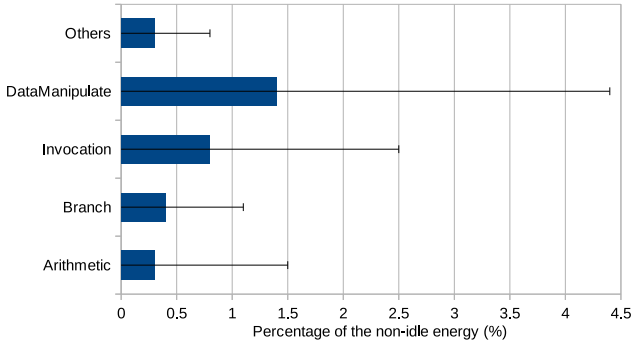

Fig. 7: The distribution of bytecode energy

To answer this question, we plot the average ratio of the energy consumption of each bytecode to the total non-idle energy consumption of applications. The average ratio of a bytecode B^i is given by the formula: $\frac{\sum EB^i_j/E_i}{N}$, where EB^i_j is the energy consumption of the bytecode B^i in the jth application, E_j is the total non-idle energy of the jth application, and N is the number of applications. We measure the energy consumption of Java bytecodes which are generated from Dalvik bytecodes of Android executable files by reverse engineering. These two sets of bytecodes are different in the format of operation and the way they manipulate data. Although Java bytecodes and Dalvik bytecodes are not equal to each other, the types of their operations are similar. For example, both of them have instructions for arithmetic calculations, branching, and invocation. Therefore, instead of reporting the energy consumption of each bytecode, we categorize bytecodes into five categories which represent the main operations in Java and Dalvik bytecodes. Since Java bytecodes are generated from Dalvik bytecodes, the functionalities of corresponding bytecodes are not changed. Thus, the energy consumption of a category in Java bytecodes will reflect the energy consumption of the category in Dalvik bytecodes. These five categories are Arithmetic, Branch, Invocation, Data Manipulation and Others. Arithmetic is the instructions for arithmetic calculations, such as *add* and *multiply*. Branch is control flow instructions, such as *conditional jump*. Invocation is instructions that call a method, such as *invokevirtual* in Java. The category Data Manipulation in Java represents the bytecodes to manipulate the stack, such as *iload*. In Dalvik, it represents the bytecodes of manipulating the registers, such as *move*. The category Others represents all other bytecodes that are not in the above four categories. Examples of Others include *monitor-enter*, *instanceof*, *length*, etc. The result of this analysis is shown in Fig. 7.

From the results, we find that the energy consumption of data manipulating bytecodes is 1.7-5.4 times larger than other categories. This is because data manipulating operations are used more frequently than other operations. Before the execution of any arithmetic or invocation instructions, the arguments have to be properly set by the data manipulating instructions. For example, the operands have to be loaded into correct registers before arithmetic operations.

H. RQ 8: Is time equal to energy?

To address this question, we measured two metrics. First, we calculated the size of the intersection of the top ten most energy consuming APIs and the top ten most time consuming APIs for each app. Second, we mapped the ranking of the top ten most time consuming APIs in descending order to that of the top ten most energy consuming APIs in descending order. To be more specific, for each application, we calculated $\sum_{i=0}^{10} O_i$, where $O_i = 1$ if the i-th most energy consuming API is also the i-th most time consuming API, otherwise $O_i = 0$. We chose this metric because if time could accurately reflect the energy consumption, the ranking of API calls with respect to time should be the same as that of APIs with respect to energy.

In our study, the average size of the intersection of the top ten most energy consuming APIs and the top ten most time consuming APIs is 9.1 with a standard deviation of 0.8. The average number of APIs that are ranked the same is 4.6 with a standard deviation of 2.0. These results show that although the top ten most time consuming APIs are roughly the same

as the top ten most energy consuming APIs, the ranking of each API may be significantly different. In practice, optimizing the energy consumption of an API is not trivial; an inaccurate ranking may lead developers to incorrectly prioritize their work and increase the amount of effort needed to achieve energy optimization.

I. RQ 9: What granularity of measurement is sufficient?

We answer this question by measuring the difference between our nanosecond and millisecond level measurement. To get the millisecond value, we first replace the nanosecond level fine grained time stamps of *vLens* with the normal millisecond level time stamps from *System.currentTimeMillis()*. Then we repeat the steps in Section III with the rough time stamps to get the millisecond level results. We compare the energy measurement on the millisecond level to that on the nanosecond level. The error ratio is given by the equation $|\frac{milli-nano}{nano}|$, where $milli$ and $nano$ are the results from the millisecond and nanosecond level measurement, respectively.

In our measurement, the mean error rate of the non-idle energy is 64.1% and the largest error rate is over 2500%! Such a number indicates that using only millisecond level is not sufficient and is likely to give misleading results. The use of nanosecond level time stamp can improve the accuracy of measurement.

We argue that nanosecond level measurement is sufficient because it could capture all API calls and methods. The clock frequencies of current processors are normally at the GHz level, so each instruction is executed at the nanosecond level. Since methods and APIs in general have several instructions, the nanosecond level time stamps are able to capture all methods and API calls.

J. RQ 10: Is it necessary to account for idle state energy?

To answer this research question, we report the measurement error of total energy caused by neglecting the idle state energy of an application. In traditional measurements, developers assume that energy consumed during the execution of an application belongs completely to the code of the application. However, this assumption is not valid. As we discussed in Section IV-B, the system keeps consuming energy while the running application is in the idle state. Thus, assigning the energy consumed during the idle state to the code of the application may be misleading.

In our measurement, we calculate the energy consumption of an application without subtracting the idle state energy (IdleKept), which sums up all the energy samples between the first time stamp and the last time stamp during the execution of the application. The energy consumption during the non-idle state of an application (IdleSubtract) is obtained by summing up the energy consumption of all the execution paths of the application. In the result, the average difference between IdleKept and IdleSubtract is 36% and the largest error is over 99%.

This result indicates that assuming the energy consumed during the execution of an application is the total energy can introduce inaccurate measurements and misleading results. Developers may incorrectly optimize the energy consumption of their applications. For example, developers may put a lot of effort into optimizing the energy consumption of the code of an application, even though most of the energy is consumed during the idle state of the application. In this case, the efforts will be wasted since the idle state energy consumption of the application cannot be reduced by optimizing the application code.

V. Threats to Validity

External Validity To ensure that our applications are representative of real world applications, we randomly selected 405 applications from the Google Play market. These applications were from 23 categories and had various functionalities. Their size ranged from 1.6KB to 18 MB and their number of instructions ranged from 1,507 to 1,866,692.

To ensure the workload generated by Monkey was representative of real use cases of the applications, we first removed all games since we could not generate deterministic workloads for them. We then removed all applications with statement coverage less than 50%. Therefore, the workloads we used in our experiments covered most of the statements of our test applications. Finally, we removed applications from the subject pool that crashed during execution. Monkey did not generate any text for user input, such as login information. However, if the user input was critical to an application, it was not likely to reach the statement coverage of 50% and would be removed.

Internal Validity In Section IV-C, the energy consumption of Bytecodes and Outliers were estimated with the method from our previous work [8]. The estimation error for our technique was 19.2%. Since the Bytecode energy was only 3.2% of the total non-idle energy, a 19.2% estimation error in Bytecode energy would not influence our final conclusion. For the accuracy of detecting Outliers, we previously showed [8] that our method could accurately detect process switching and garbage collection. In Section IV-G, the energy of bytecodes was also estimated through the robust regression. Thus, it had the same estimation error as we reported in Section IV-C. Since the energy consumption of each category differed by at least 50%, here again, a 19.2% estimation error would not change our conclusion.

In our study, we measured the energy consumption once for each application, which may introduce bias to the result of each individual application due to the randomness of the workload and the environment, such as network conditions. However, since we made our conclusions based on the whole set of our test applications, this bias was mitigated by the large number of applications in our experiment.

Lastly, in our study Monkey was running on the controller desktop and pushed UI events to the smartphone through the network. So Monkey did not introduce any extra energy consumption except the network energy consumption to transmit UI events. We checked this increment of network energy on a subset of our test applications and found that it was not noticeable compared to the total energy consumption.

Construct Validity In Section IV-D, the average energy consumption of a component may be distorted by the popularity of the component in the applications. One component may consume a lot of energy when it is used, but may

have low average energy consumption due to low usage rates across all applications. To avoid this distortion, we reported the effective average energy for each component, which excludes the applications that did not use a component from the statistics of the component. In Section IV-E, the average energy distribution of an API across all applications may be distorted by the frequency of the API being called in all applications. Some APIs may have more average energy consumption at the application level than others because they were used in more applications. To avoid this bias, for each application, we also reported how much energy was consumed by the top ten most energy consuming APIs in the application. In Section IV-G, our robust regression model estimated the energy consumption of JVM bytecode. Although the JVM bytecode was generated from the DVM bytecode by reverse engineering, it was not the same as DVM bytecodes. To address the difference between the JVM bytecodes and the DVM bytecodes, we reported the energy of categories of bytecodes instead of the energy of each specific bytecode. Since there was a mapping of functionalities from the JVM bytecodes to the DVM bytecodes, the categories were consistent for both of them.

VI. Related Work

In previous work [2], we conducted a small-scale empirical evaluation of energy-saving programming practices, and provided some concrete guidelines to develop more energy efficient code. For example, using certain coding practices for reading array length information and accessing class fields reduce energy consumption. However, we experimented on small programs instead of real-world mobile apps and did not analyze the energy usage patterns at a high level. *Eprof* [7] provides insight into the energy usage of certain APIs. Their results show that 65%-75% of an app's energy is consumed by third-party advertisement APIs and is concentrated in I/O. Other energy usage patterns were not discussed. Additionally, the results in this paper used a significantly larger pool of subject applications (390 more), which provides a broader picture of the energy usage patterns of mobile apps.

Linares-Vásquez and colleagues [16] conducted an empirical study on analyzing API methods and mining API usage patterns in Android apps. As in our work, they utilized the same power monitor Monsoon [11]. The authors provided practical advice or actionable knowledge for developers. Our work differs from Linares-Vásquez and colleagues' in several aspects. First, they only identified energy-greedy API related usage patterns; whereas our work examined application, package, loop, and bytecode level energy usage. Second, over 400 apps from different domains were tested in our work compared to 55 mobile apps in their work. The higher number of apps helps to provide a more comprehensive and general picture of the energy usage of mobile apps. Third, due to the use of the Traceview of Android SDK, they could not report idle and non-idle energy as in our work. Fourth, compared to nanosecond time stamps, they measured the millisecond level energy usage, which as we show can lead to inaccuracies in the reported measurements.

Previous studies of mobile apps have also focused on app usage. Böhmer and colleagues [17] collected information, such as average session, time, and location, from 22,626 apps based on 4,125 users. Froehlich and colleagues [18] designed MyExperience to log objective traces and subjective feedback of real usage data. McMillan and colleagues [19] proposed a method to collect user feedback. They used the collected feedback to improve the user experience. Shepard and colleagues [20] proposed LiveLab to collect the wireless usage of smartphones. They discovered that only a few apps were heavily executed in daily usage. Developers could make use of this app usage information to improve app performance. However, none of the works above studied the energy usage of mobile apps.

Wilke and colleagues [21] studied the significance of the energy efficiency-issues for mobile apps by analyzing a large set of user comments extracted from the Google Play market place. Kwon and colleagues [4] reported the impact of distributed programming abstractions on application energy consumption and presented a set of practical guidelines for the development of distributed applications. Zhang and colleagues [5] compared power models and energy bugs across different smartphone platforms and discovered some useful bug patterns related to energy consumption. Nevertheless, all of these approaches focus very narrowly on specific problems or research on energy usage from a different perspective.

Another group of related work estimated or measured energy consumption. Different approaches were proposed to achieve different levels of granularity. Our previous work, *vLens* [8] and *eLens* [1], [22], used fine grained instrumentation to achieve source line level energy measurement. In *Aneprof* [23], the energy consumption could be estimated at the method level based on the proposed model. Dong and colleagues [24] studied a self-modeling solution to calculate per-application energy. Cycle-accurate simulators, such as *SimPanalyzer* [25] and *Wattch* [26], estimated the energy consumption of embedded systems by building the energy model of circuits. Seo and colleagues [6] presented a framework for estimating the energy consumption of Java-based software systems. Other models [27], [28], [29], [30] obtained the software energy at the instruction level. All of the above approaches or models focused on the estimation of the energy consumption, some of which provided meaningful feedback to developers. Nonetheless, energy usage patterns within apps were not investigated or targeted, and the amount of tested apps was limited to dozens, which in general was not enough to reach more general conclusions.

VII. Conclusions

In this paper, we presented the results of an extensive empirical study we performed on the energy consumption of Android based mobile applications. For this study, we examined over 400 apps from the Google Play market and tracked energy at different levels of granularity, ranging from the whole application level to the source line level. Our study revealed several interesting trends and observations. First, we found that most applications consumed over 60% of their total energy in idle states (RQ2). Second, we found that network was the most energy consuming component among all of a smartphone's hardware components (RQ4). Third, we found that only a few system APIs dominated the energy consumption of the non-idle state of Android applications (RQ5). Finally, we found that data manipulation operations were the most energy consuming among all operations (RQ7). In addition to these

energy patterns, we also analyzed three common practices in energy related studies: using time as an approximation; using millisecond level measurement; and neglecting idle energy. Our analysis of these practices showed that they could lead to large inaccuracies in the observed data and could undermine the validity of statistical studies. These findings could have broad implications for past and future energy studies, as they demonstrate the necessity of certain types of measurements and practices. Overall, our results provided both interesting insights for developers and useful guidance on effective practices to employ for energy related studies. Taken together, we expect these findings to help improve developers' understanding of the patterns of energy consumption in mobile applications.

VIII. Acknowledgments

This work was supported by National Science Foundation grant CCF-1321141.

References

[1] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, May 2013.

[2] D. Li and W. G. Halfond, "An investigation into energy-saving programming practices for android smartphone app development." in *Proceedings of the 3rd International Workshop on Green and Sustainable Software (GREENS)*, 2014.

[3] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys)*, 2012.

[4] Y.-W. Kwon and E. Tilevich, "The impact of distributed programming abstractions on application energy consumption," *Information and Software Technology*, 2013.

[5] J. Zhang, A. Musa, and W. Le, "A comparison of energy bugs for smartphone platforms," in *MOBS*. IEEE, 2013, pp. 25–30.

[6] C. Seo, S. Malek, and N. Medvidovic, "An energy consumption framework for distributed java-based systems," in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering (ASE)*. ACM, 2007, pp. 421–424.

[7] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys)*. ACM, 2012, pp. 29–42.

[8] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, July 2013.

[9] P. J. Rousseeuw and A. M. Leroy, *Robust regression and outlier detection*. John Wiley & Sons, 2005, vol. 589.

[10] D. McIntire, K. Ho, B. Yip, A. Singh, W. Wu, and W. J. Kaiser, "The low power energy aware processing (leap)embedded networked sensor system," in *IPSN*, 2006.

[11] "Monsoon solutions. power monitor. http://goo.gl/w2NSqW."

[12] "Android monkey. http://goo.gl/wSlG0b."

[13] "Google play crawler http://goo.gl/0yDL5w."

[14] M. Dong and L. Zhong, "Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays," in *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys)*, 2011.

[15] D. Li, H. Tran, Angelica, and G. J. Halfond, William, "Making web applications more energy efficient for oled smartphones," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[16] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014.

[17] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI)*, 2011.

[18] J. Froehlich, M. Y. Chen, S. Consolvo, B. Harrison, and J. A. Landay, "Myexperience: A system for in situ tracing and capturing of user feedback on mobile phones," in *Proceedings of the 5th international conference on Mobile systems, applications and services (MobiSys)*, 2007.

[19] D. McMillan, A. Morrison, O. Brown, M. Hall, and M. Chalmers, "Further into the wild: Running worldwide trials of mobile systems," in *Pervasive*, 2010.

[20] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: Measuring wireless networks and smartphone users in the field," *SIGMETRICS Perform. Eval. Rev.*, 2011.

[21] C. Wilke, S. Richly, S. Gotz, C. Piechnick, and U. Assmann, "Energy consumption and efficiency in mobile applications: A user feedback study," in *GreenCom, iThings/CPSCom, CPSCom*, 2013.

[22] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating android applications' cpu energy usage via bytecode profiling," in *Proceedings on the first international workshop on green and sustainable software (GREENS)*, May 2012.

[23] Y.-F. Chung, C.-Y. Lin, and C.-T. King, "Aneprof: Energy profiling for android java virtual machine and applications," in *Proceedings of the 17th International Conference on Parallel and Distributed System (ICPADS)*, 2011.

[24] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys)*, 2011.

[25] T. Mudge, T. Austin, and D. Grunwald, "The Reference Manual for the Sim-Panalyzer Version 2.0."

[26] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2000.

[27] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Trans. Very Large Scale Integr. Syst.*, 1994.

[28] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, "Instruction Level Power Analysis and Optimization of Software," in *Proceedings on 9th International Conference on VLSI Design (VLSID)*, 1996.

[29] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking energy in networked embedded systems." in *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[30] T. Li and L. K. John, "Run-time modeling and estimation of operating system power consumption," *ACM SIGMETRICS Performance Evaluation Review*, 2003.