2016 10th International Symposium on Theoretical Aspects of Software Engineering

Lightweight Method-level Energy Consumption Estimation for Android Applications

Qiong Lu[1,3], Tianyong Wu[2,3], Jiwei Yan[2,3], Jun Yan[†1,2], Feifei Ma[†2], Fan Zhang[4]
[1]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[3]University of Chinese Academy of Sciences
[4]Beijing University of Technology
Email: {luqiong13,yanjun}@otcaix.iscas.ac.cn, {wuty, yanjw, maff}@ios.ac.cn

***Abstract*—The energy consumption problem is a hot topic in Android communities. The high energy cost caused by improper development brings lots of complaints from users. An effective and efficient energy consumption analysis technique can guide the developers to improve the energy efficiency of their apps. Existing researches on this problem focus on either system entity level that gives the energy consumption of the hardware, or source line level that calculates the energy cost of source codes. With the consideration of accuracy and cost of analysis, this paper proposes a lightweight and automatic approach to estimate the method-level energy consumption for Android apps. We construct a statistical model from a set of energy values obtained by Dalvik bytecode based instrumentation and software-based measurement, to predict the energy consumption of execution sequences of methods. The experiments on several real-world apps show that the proposed techniques have low overhead while persisting acceptable accuracy.**

I. Introduction

Android application market is expanding rapidly. This expansion is mainly driven by the development of mobile apps, which provide users with more interesting and diverse functionality. However, one of the most frustrating things is that these applications often require a large amount of battery power, which drains the battery life quickly and brings a lot of complaints from users. Besides, millions of apps were often developed in an energy oblivious manner, so optimizing the energy consumption of apps is of critical importance.

To reduce Android applications' energy consumption, researches have explored some techniques on the platform layer [20]. However, their improvements are not sufficient enough for reducing the energy cost since a poorly written app will still die the battery life out. Hence, it is also significant to optimize the energy cost of Android apps, in which how to analyze the energy consumption distribution in an app is a considerable problem. It can help developers to know how the battery is utilized inside an app and guide them to improve the energy efficiency.

For the application level, several techniques were put forward to estimate the energy consumption on the components and source lines. Some researches focus on the energy cost of program entities, including WIFI, GPS, etc. [7]. Although their approaches can provide a high-level cognition to developers, the information seems not enough to map to the code. The developers can not easily find the root causes of energy hotspots [6] to optimize apps via these types of techniques. To address this issue, D. Li et al. focus on estimating the energy consumption at the source line level [13] that is a fine-grained work and need the testers to design a set of use cases. However, due to the complexity of real-world apps, it is expensive to build a set of representative input scenarios with high coverage of source lines.

In addition, through communicating with developers in several Internet companies in China, we find that they also focus on the energy consumption related to the execution sequences that consist of methods, which can mirror the app's behavior and explain how the energy flows inside an app. Therefore, we believe that the method (the basic unit in the program development) energy estimation appears to be more reasonable. Furthermore, most developers only care about the energy hole methods, rather than the accurate energy consumption values that are difficult to obtain. Hence, it motivates us to propose our approach, a new lightweight approach estimating Android applications' runtime energy consumption at the method level, providing developers with helpful hints for apps' understanding and tuning.

In this paper, we combine the software-based measurement with statistical modeling to estimate the Android applications' energy consumption. Compared to the hardware-based energy measurement (e.g., [12], [21]) that records the energy consumption of the whole device, the software-based one can profile the app under test with small background noise. In order to obtain the information about execution sequences, we design a Dalvik bytecode based instrumentation technique to mark all the methods. After we execute the app under the monitoring by an energy measurement software, we can get the energy data along with the corresponding runtime sequences. Next, we employ a linear regression analysis to generate a prediction model that maps the energy consumption to the methods, and refine it by taking a feedback way to improve the prediction accuracy. Finally, we figure out the method energy consumption distribution and high cost parts in apps.

In our experiments, we have applied our approach to several

Supported in part by National Natural Science Foundation of China (Grant No. 91418206) and the National Key Basic Research (973) Program of China (Grant No. 2014CB340701).

† Corresponding author.

Android applications and our results reveal that the proposed techniques are able to estimate and predict the energy cost of each method and execution sequences. Our results also suggest that the granularity level of our model is reasonable, where the overhead brought by our instrumentation technique is less than 3% that has little influence on the measurement values, and the average error of prediction values is no more than the source line level.

The rest of this paper is organized as follows. In Section II, we briefly introduce some related concepts and definitions. Details of our approach are discussed in Section III. Then we present some experiments and evaluate our approach in Section IV. The related work and conclusion are in Section V and VI, respectively.

II. Background

In this section, we describe some background knowledge that will be used in the following parts. Firstly, we introduce some definitions related to the energy consumption. In order to get runtime information, we propose a Dalvik bytecode instrumentation technique. Thus, in the second subsection, we explain some backgrounds about Dalvik. At last, we introduce the linear regression used in our energy model construction.

A. Method Energy Consumption

There are many trivial factors and external environment influencing the energy consumption of Android applications, such as the user operations, temperature, running on different devices, etc. So the energy consumption measurement values of the same method sequence may be different in two executions. Thus, we need to ignore these minor factors and just give a rough estimation among methods, which is more interesting to developers. We assume that the outside environments (e.g., the temperature, the location, the battery power voltage, etc.) are stable, the screen brightness is set fixed and the users' operations are normal with no deliberate delay.

An application on the same Android device executes the same sequence of methods that may have different energy consumption because of the differences of execution logic (different program paths) inside methods. In our approach, we define the method energy consumption as average of all the possible executions. Formally speaking, a **method energy consumption** $w = (m, ins)$ denotes average energy consumption by method m which contains the instruction set ins. It only involves basic instructions but not the nested invocation methods. For example, as shown in Fig. 1, The energy consumption of method `a()` involves the energy consumpiton of the three instructions in `a()` excluding that of method `b()`. Furthermore, we define the **method sequence energy consumption** $e = (s, M_s)$ denotes average energy consumption by method sequence s which contains the set of methods M_s. Without consideration of measurement errors, we have $e = \sum_{m \in M_s} w$.

B. Dalvik

The Android application is actually an Android Package (apk) file, which is a type of archive file with .apk as the

```
public void a(){
    int x;
    x = 3;
    b();
}
private void b(){
    // other codes
}
```

Fig. 1. A `Java` Code Snippet

filename extension. The source codes of an Android app are commonly compiled to Dalvik bytecodes, that is a register-based instruction set. Our instrumentation is based on `smali`, a dex format of Dalvik bytecodes. The `smali` file is composed of several statements, which comply with a series of rules, with the keyword `.class`, `.locals`, and so on. We only care about the methods, starting with `.method` and ending with `.end method`, between which the instructions form a method block. There are two kinds of registers used by `smali`, the local register and the parameter register. In a method, `.locals` represents the number of used local registers and `.param` represents that of parameter registers.

The code snippet in Fig. 2 is an example of a simplified `smali` method. As shown in Fig. 3, the local variables in the example are stored at v_0 and v_1, which are the local registers, and the method parameter is stored at v_2, which is a parameter register. By the way, the parameter variables are always stored at the last registers.

```
.method protected onCreate(Landroid/os/Bundle;)V
        .locals 2
        .param p1, "savedInstanceState"
        // other bytecodes
.end method
```

Fig. 2. A `Smali` Code Snippet

v0	Local Register 1
v1	Local Register 2
v2	Parameter Register 1

Fig. 3. Register Distribution

Note that the straightforward instrumentation may result in apps crash at runtime due to the collision of registers. Therefore, we apply several more local registers to store the variables of instrumentation to avoid using the local registers that have been occupied. Moreover, it is necessary to distinguish the local registers and the parameter ones, since the parameter registers always rank in the final.

C. Linear Regression Analysis

Let $\vec{x} = (x_1, x_2, \ldots, x_n)$ be arguments of a system and for a pair $(\vec{x}, y)$ where y is the dependent variable, a linear regression model assumes that the relationship between y and $\vec{x}$ is linear. Assuming that $A = (a_1, a_2, \ldots, a_n)$ denote the coefficients, the model takes the form $y = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \vec{x}A^T$. We often have multiple pairs of experimental data. Let $X = (\vec{x_1}, \vec{x_2}, \ldots, \vec{x_m})^T$ and $Y = (y_1, y_2, \ldots, y_m)^T$ be m pairs of arguments and dependent variables, respectively. The vector form of the above equation can be written as

$$Y = XA^T.$$

A solution to fit a proper assignment of A is the simulated annealing (SA) algorithm [11], which is a probabilistic technique for approximating the global optimum of a given function.

III. Approach

In this paper, we aim to estimate the energy consumption at the method level for Android applications. We propose a lightweight and automatic approach to generate a prediction model to map the energy consumption to the methods. The model can also be used to predict the energy cost of a specific execution sequence.

Fig. 4 shows the architecture of our approach. There are five main modules : (1) APK Instrumentation, (2) Runtime Monitoring, (3) Data Preprocessing, (4) Energy Modeling, and (5) Feedback.

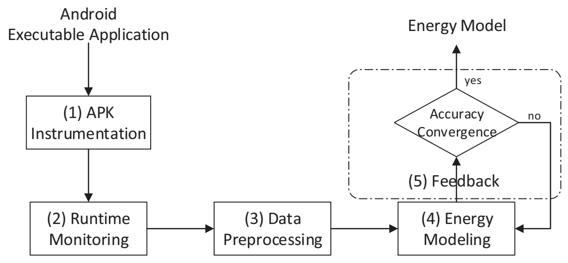


Fig. 4. Architecture of our approach

Our approach takes an Android executable application as input. First of all, we instrument probes into the apk file to track the executed methods. Then, we run the instrumented application on the Android device to get the runtime path information and energy measurement data, which are stored in the log files and database, respectively. Next, we use these raw data to count the corresponding methods' invocation times and calculate the energy consumption in each run, which are the inputs to the energy modeling component. We use linear regression analysis to model the energy consumption that can predict the energy cost of a given execution sequence. A feedback approach that is an optional step, is used to optimize the effect of the energy model by adjusting the training data from a small reserved set. If the accuracy of the model is acceptable, we use it to predict energy consumption. Or else, we feed the results back to the energy model to optimize it. After several rounds, the model will be stable and output the feasible predicted values.

A. APK Instrumentation

The input of the first step is an executable Android app. To obtain the information of the runtime execution sequences, we need to insert probes into the original Android app to collect messages about how the app executes from the log file.

Our approach is based on the Dalvik bytecodes instead of the source codes, which may be not available for analyzers sometimes. Though there are some tools that can decompile the bytecodes to source codes, they cannot work well on all apps. Therefore, instrumentation on the bytecodes is a more reasonable and generic choice.

The main difficulty lies in the register-based Dalvik instruction set for the straightforward instrumentation approach may result in apps crash at runtime. We provided an effective and lightweight method for the Dalvik bytecode instrumentation, that is applicable to real world apps. We only modify the `smali` files, that are decompiled from the Dalvik bytecodes, without changing any other things of the original application.

Fig. 5 shows the process of apk instrumentation. Since our approach is at the method level, we take method as an instrumentation unit. Firstly, we need to unpack and decompile the Android application to extract its `smali` and resource files. To collect information about the methods executed when the app is running, we need to insert probes about method information to the `smali` files. By traversing the `smali` files, we identify all the methods excluding the system ones (original Android APIs) and the `R` file. Then under the premise of not violating the register rules, we insert our probes at the entry points of the methods. The probes are a set of tuples in the form of $\{c, m\}$, where c denotes the class name, and m denotes the method name. After this is done, we repackage all the `smali` and resource files to an executable Android application, which is embedded with the path information.

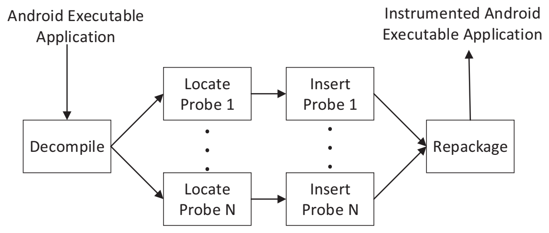


Fig. 5. APK Instrumentation

We illustrate how we insert probes into the example code in Fig. 2. Assume that the probes we need to insert would occupy two local registers, we should apply two more new registers. Just add the value after `.locals` to 2 if the total number of registers does not exceed 16. However, if not, we use the instruction `/range` to expand the registers' index scope and then map each register identification to the new one. As shown in Fig. 7, since the parameter variable should be stored in the last register, it will occupy `v4` instead of `v2`. So when we refer to the new registers, we should use `v2` and `v3`, rather than `v3` and `v4`. Fig. 6 shows the method after instrumentation.

```
.method protected onCreate(Landroid/os/Bundle;)V
        . locals 4
        .param p1, `` savedInstanceState ''
        // My Instrumentation
        // other bytecodes
.end method
```

Fig. 6. `Smali` Codes After Instrumentation

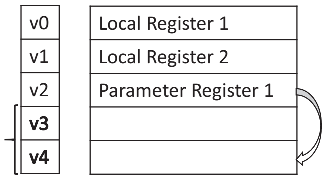


Fig. 7. Register Identification Mapping Process

After this step, we obtain a modified Android app carrying the path information we need. It will write the executed method sequences into the log files for further analysis when the app runs on the real device.

B. Runtime Monitoring

At the first step, we have got an instrumented application with the path information tracing instructions. In this step, we execute the instrumented app and measure the energy consumption of each run.

Fig. 8 shows the framework of this step, which is composed of the runtime controller and the runtime measurement. The controller is built on the computer, while the measurement is on the phone. We install the instrumented apk file on the Android device under the control of the computer, which will monitor its entire execution and capture the method sequences information. At the same time, we execute the application on the cell phone and measure the energy consumption. When the app stops running, a log file recording the runtime path information and a database recording the energy information will be generated. The whole process is totally automatic without any user interactions.

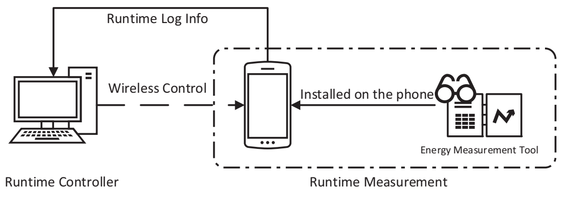


Fig. 8. Runtime Monitoring

Note that the cell phone is controlled via wireless connection instead of cable connection, which could charge the battery and disturb the measurement of power consumption of the application under test.

We employ a random testing tool Monkey, a popular command line tool to test Android apps, as the controller. Monkey will automatically generate pseudo-random stream of events and send them to the phone, simulating user input like touch, gesture, and so on. For the runtime measurement, we employ another tool, Trepn profiler 5.1 [5] provided by Qualcomm. Trepn uses an interface of the Linux kernel to take a closer look at the power management IC on a supported chip to measure energy consumption. Compared with other energy measurement apps, Trepn can profile not only the energy consumption of Android system, but also that of the individual Android app, which helps us get rid of some of the background influence to a certain extent.

As shown in Fig. 8, after this step finishes, we'll get the executed method sequences information in the log files and the energy consumption information in Trepn's database. We will perform some further analysis based on these raw data in the following steps.

C. Data Preprocessing

Since the raw data collected in the second step contains a large amount of irrelevant information brought by Monkey and Trepn for our processing, we need to do some arrangements. Specifically, in the log files, the generated runtime information has several categories, including our instrumentation path information and some others. We count the executed method invocation times at each run and format them as a set of features that can be used in the next step. In Trepn's database, we extract the average battery power measurement, and multiply by time as the energy consumption of one test, ignoring data of memory, network, etc. With these preprocessed data, we construct a training data set and a small reserved data set for modeling in the next step.

D. Energy Modeling

In this step, we employ a linear regression analysis to produce a mapping of each method's energy cost.

After the data preprocessing step, we get a number of features. Let $x_{i,j}$ denote the invocation times of the jth $(1 \leq j \leq n)$ method in the ith test and e_i denote the energy consumption for the same test. We can define the feature of the ith test as $\{x_{i,1}, x_{i,2}, \ldots, x_{i,n}, e_i\}$. With m executions, let $X = (x_{i,j})_{m \times n}$ be the method invocation times matrix, $E = (e_1, e_2, \ldots, e_m)^T$ be the average energy measurement values vector. Assume that $W = (w_1, w_2, \ldots, w_n)$ is a coefficient vector to represent the energy cost of each method. We have the linear equation $E = XW^T$ that describes the relationship between the method invocation times and energy measurement values. In theory, solving this equation will get the assignments of coefficient vector W, which forms the core of the energy model.

However, from the preliminary experimental results, we observe that due to the improper initial values and the large number of the coefficients in the model, the straightforward solving of the equation is hard. So we try to fit a set of optimal solutions instead. It is feasible that w_i $(1 \leq i \leq n)$ is in range of $[0, \min \frac{e_i}{x_{ij}}]$, since the energy cost of each method is non-negative and less than the total energy consumption of an execution. To get the global optimum solution as much as possible, we employ the simulated annealing (SA) algorithm for the linear regression analysis. Given W a group of initial values, the SA algorithm randomly changes the value of an element in the matrix and iteratively moves from the current candidate solution S to a new candidate solution S'. It accepts bad solutions by the probability $e^{-(c(S\prime)-c(S))/KT}$, where $c(S\prime)$ and $c(S)$ are calculated by

$$c(S) = \frac{1}{2m}\sum_{i=1}^{m}(E_S - E)^2.$$

In the acceptance probability function, K is Boltzmann's constant, and T stands for the temperature, a global time-varying parameter. Formula $c(S)$ represents the cost of the candidate solution S.

Finally, we obtain an energy model, which maps the energy consumption to the methods and can be used to predict the energy cost of the specific execution sequences.

E. Feedback

We perform a feedback approach to optimize the effect of the energy model by using the reserved data set. If the prediction accuracy of the energy model is not satisfactory, we replace a case in the training data set with a reserved one in each iteration. Then we can get a new prediction model and compare it to prior one. If the model is better, we accept the change. Otherwise, we rollback the replacement and try another reserved case. This feedback process will terminate when the accuracy of prediction is acceptable or the iteration times reach the upper bound.

The reason for introducing the feedback mechanism is that the size of training data set designed artificially may be not feasible. In addition, from the results of our experiments IV-E, a set of 250 cases is enough to get an acceptable accuracy model for our instances. We can start from a small set of training data and adjust it gradually if needed.

IV. Evaluations

We implemented our approach in Python language with some ADB [1] scripts. The apk instrumentation part uses Androguard [3], a static analysis tool for Android apps, to construct the control flow graph of the Dalvik bytecodes to help find the location of instrumentation. We employ APKTool [2] to decompile and repackage the Android apk files and SignApk [4] to conduct the re-signature process if needed.

A. Experimental Setup

We ran the experiments on a Xiaomi 2S cellphone, with its CPU 1.7GHz, 2GB RAM, and the battery capacity 2000mAh. We choose Trepn running on the device to measure the energy consumption and employ Monkey to control the execution on a Windows 7 desktop platform containing an Intel (R) Core (TM) i7 CPU 2.93GHz with 8GB RAM.

Table I lists the six popular applications from a Chinese Android market, most of which have been downloaded more than one hundred thousand times. These apps cover five commonly addressed categories and can be valid representatives of the different profiles of energy consumption. We also give the category, size, number of classes (#C) and number of methods (#M). We choose these applications from various categories to enrich our experiments.

TABLE I
Experimental Applications

App	Category	Size (B)	#C	#M
Saolei	Game	475K	132	724
Piano	Game	721K	449	2838
CRadio	Media	1568K	668	4252
CNTV	Video	7961K	905	6700
SogouBrowser	Browser	8417K	2970	18542
SogouReader	Reader	7446K	3331	22519

To evaluate the accuracy of our energy model, we adopt two statistical metrics, the multiple correlation coefficient (R^2) and the average relative error (ARE).

In statistics, the R^2 is a well-known measure of how well a given variable can be predicted by other variables. It is the correlation between the variable's values and the best predictions, which can be computed from the predictive variables. In our cases, the given variable to be predicted is the energy consumption of the i'th execution ($\hat{e_i}$), and the set of other variables are each method's energy consumption. The formula of R^2 is defined as follows.

$$R^2 = \frac{\sum(\hat{e_i} - \overline{e})^2}{\sum(e_i - \overline{e})^2}$$

where the $\hat{e_i}$ denotes the energy consumption calculated for the i'th execution sequence based on our energy model. The e_i denotes the measured energy consumption of the i'th execution sequence at runtime and $\overline{e}$ is the average of all the measured ones. The R^2 takes values between 0 and 1 with a higher value close to 1 indicating a better predictability.

The ARE gives an indication of how good a prediction is relative to measurement value. The lower ratio of ARE, the more accurate the model is. The formula is shown as

$$ARE = \frac{1}{n}\sum\frac{|\hat{e_i} - e_i|}{e_i}$$

B. Instrumentation Overhead

Since it is hard to measure the energy consumption of the instrumentation code directly, in order to estimate the overhead of instrumentation, we compare the number of instructions of the apps' bytecodes before and after instrumentation, and we also did several groups of experiments to compare the average execution time of two versions of apps under the same event sequences. We set 20,000 as the number of random events for an execution sequence. Table II shows the overhead. The first column shows the app name, followed by the number of instructions before and after instrumentation (i.e., #Instruction bef. and aft.) and its overhead O_n, calculated by $O_n = (aft - bef)/bef$. The last main column shows the execution time, where T_b and T_a denote the average time consumption by each application before and after instrumentation respectively, and $O_t = (T_a - T_b)/T_b$ denotes the corresponding overhead of execution time.

TABLE II
OVERHEAD OF INSTRUMENTATION

App	#Instruction			Time		
	bef.	aft.	O_n	$T_b(s)$	$T_a(s)$	O_t
Saolei	25447	25985	2.1%	77.66	80.48	3.6%
Piano	102049	104191	2.0%	69.27	70.32	1.5%
CRadio	149476	152153	1.7%	114.73	118.95	3.6%
CNTV	233186	237820	1.9%	62.05	64.54	4.0%
SogouBrowser	739369	752799	1.8%	135.08	138.34	2.4%
SogouReader	950227	966410	1.7%	79.45	81.60	2.7%

From the two tables, we observe that only about 2% instructions are inserted into the original apps. On average, the instrumented versions take within an average about 3% more time than the original one. Therefore, these results are fairly satisfactory and indicate that the instructions added by our instrumentation technique have little influence to energy consumption of apps under test.

C. Analysis Time and Accuracy

To evaluate the efficiency and accuracy of our approach, we ran the subject apps on the phone and measured the time, then calculated the accuracy of the produced energy model.

The same as experiments in the former subsection, we also set 20,000 as the number of random events for an execution sequence. For the energy modeling, we take 250 executions as a training set with 20 cases as the reserved set to feedback. Table III summarizes the overall experimental results. The analysis time is shown in the second major column and the analysis accuracy in the last. We consider two aspects of the approach's analysis time, time to instrument each application T_i and time to perform the energy modeling T_m. For the analysis accuracy, we consider R^2 and ARE to measure how well the energy model can predict for the training set.

TABLE III
ANALYSIS TIME AND ACCURACY

App	Analysis Time		Analysis Accuracy	
	$T_i(s)$	$T_m(s)$	R^2	$ARE(\%)$
Saolei	1	35	0.85	3.9
Piano	3	48	0.83	4.2
CRadio	4	64	0.87	8.3
CNTV	10	69	0.91	8.4
SogouBrowser	31	143	0.88	7.7
SogouReader	44	162	0.94	7.3

From Table III, we observe that even when the application size is large (i.e., SogouBrowser, SogouReader), the analysis time is still within several minutes. And the values of R^2 range from 0.83 to 0.94 with an average of 0.88, which shows that the predicted energy consumption by our model fit well to the measured ones. The values of ARE range from 3.9% to 8.4% with an average of 6.63%, which shows that the value of predicted energy consumption is close to the measured ones. Overall, these data indicate that our model behaves well.

D. Cross Validation

In addition to evaluating the analysis accuracy via the statistical methods described above, we also adopt cross validation, a technique for assessing how the results of a statistical analysis will generalize to an independent data set, to estimate how accurately the predictive model will perform in practice.

We randomly partitioned all the preprocessed data into 2 subsets, one is retained as test data set for testing the model, and the remaining samples are used as training data that produce the energy model. Then by applying the trained energy model to the test data, we would obtain a set of energy costs. Fig. 9 compares the predicted energy consumption and the ones measured by Trepn. For each application, we list 50 groups of comparison experiments, with the solid line representing the measured energy consumption and the dotted line representing the predicted ones. Each group of experiments are listed along the X-axis and the two different energy cost values in mAh (since the battery supplies a constant power voltage) are shown on the Y-axis.

As can be seen from Fig. 9, the trend of the two lines in each application is very close. For a quantitative analysis, we show the average relative error (ARE) and the standard deviation (STDEV) of 50 experiments in each application in Table IV.

TABLE IV
CROSS VALIDATION ACCURACY

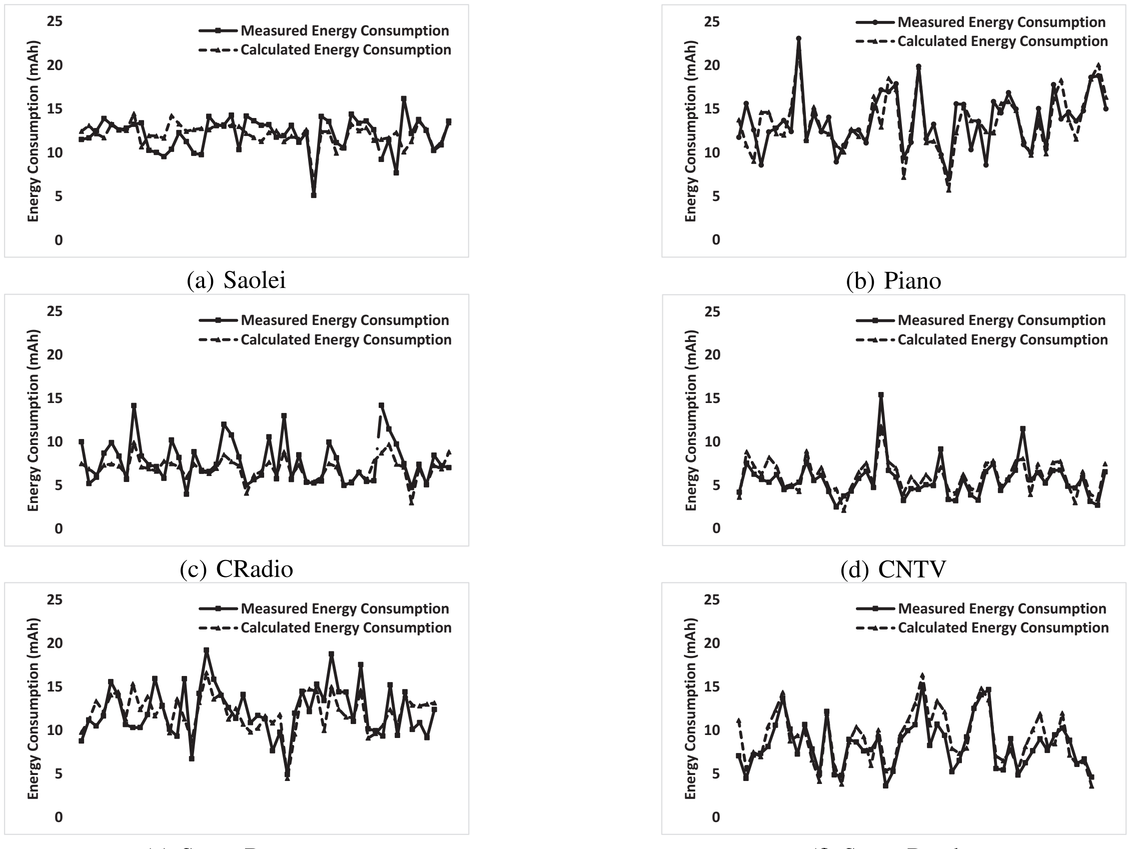
App	ARE (%)	STDEV (%)
Saolei	14.2	6.4
Pinao	12.5	8.7
CRadio	15.7	12.1
CNTV	15.6	14.1
SogouBrowser	16.7	15.2
SogouReader	17.8	12.0

As shown in the table, the values of ARE range from 12.5% to 17.8% with an average of 15.41%. The results indicate that our approach predict the execution sequence energy consumption well with the STDEV averaging 11.42%.

E. Size of Training Set

It is time-consuming to collect the large amount of training data set from runtime measurement since it needs to execute the apps under test. In order to statistic the best option of the size of training set, we conduct 5 groups of experiments for each app, that adopt 100, 150, 200, 250, 300 cases as the training set, respectively. We use the same testing set containing 50 cases to compare the effect of the trained model for each app. The experimental results are shown in Fig. 10.

The sizes of training sets are listed along the X-axis and the average relative errors calculated from the model under the test data are shown on the Y-axis. As can be seen from



(a) Saolei (b) Piano (c) CRadio (d) CNTV (e) SogouBrowser (f) SogouReader

Fig. 9. Energy Consumption Prediction of Execution Sequences

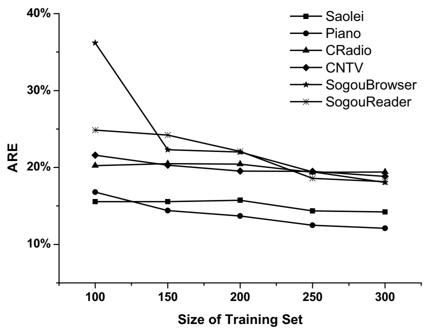


Fig. 10. Influence of Training Data Set Size

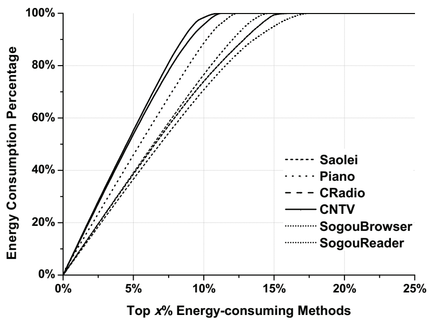


Fig. 11. Method Energy Consumption Percentage

the figure, the error is relatively large (20% above) when the training set is composed of 100 cases. With the increase of the training data size, the error is in gradual decline and it tends to be stable after the size is up to 250, that we consider to be a suggested choice to be able to obtain a better model.

F. Method Energy Consumption Distribution

From our produced energy model, we find out some high-cost methods, which take an average proportion of 9.65% in quantity but consume the 80% battery power as shown in Fig. 11. Its X-axis presents the top x% energy-consuming methods, and the Y-axis shows the corresponding energy consumption percentage.

In order to analyze the internal structure of these high-cost methods, we check the top 100 methods' bytecodes and conclude the classification results shown in Fig. 12. In the clock wise order, beginning with the part of 24%, each part respectively represents the complicated computation, the database operation, the file-related operation, the frequent instance initialization, the audio operation, the multiple loops, and some others. Note that the poorly written program code, such as the reported high-cost methods always contain a large amount of repeated computing and loops, or initialize instances frequently, that may result in the app die the battery life out.

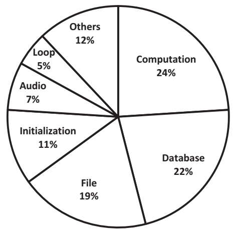


Fig. 12. Energy Consumption Percentage

V. Related work

In recent years, many researches aim to estimate the energy cost of the mobile apps from different points.

Pathak et al. track the activities of energy-consuming entities (WiFi, GPS etc.) when the app is executing on the mobile phone [18]. They compute the approximate energy consumption of applications and functions that invoke system calls. Jindal et al.[9] evaluated the energy consumption of the device subsystems (e.g., GPU or GPS). It is useful to understand the hardware and the impact of I/O on the battery lifetime. However, understanding the VM services design tradeoffs cannot be refined based on that approach because the VM is not linked to the I/O expenses. Hence, the experimental environment focuses on the energy consumed by the CPU which ranges between 20-40% of the total device consumption.

Ferrari et al. [8] presents POEM to help developers automatically test and measure the energy consumption of single application component down to the control flow level. Their approach focuses on the static analysis of the bytecode and code injection techniques to obtain measurements with several levels of granularity (e.g., class level, method level, etc.) while our approach focuses on dynamic execution techniques at the method level. Liqat et al. [16] study the energy consumption with static analysis at ISA and LLVM IR levels, and reflects it upwards to the higher source code level. Their results suggest that it is a good choice to estimate the energy consumption on these levels for conventional programs.

There are also techniques detecting the energy performance bugs [10], [17], [19], which arise from improper use of power control APIs and result in battery drainage. There are also some techniques for detecting display energy hotspots in Android applications [14], [22]. They believe that the user interfaces of a mobile app is related to the energy consumption. Li and Gallagher [15] propose an energy-aware programming approach, which is guided by an operation-based source-code-level energy model and can be placed at the end of software engineering life cycle. These techniques leverage power modeling and some display transformation approaches, such as changing the color scheme, to predict and reduce the energy consumption of an app.

VI. Conclusion

Energy consumption is a serious problem for pocket devices. We investigated this problem and proposed a lightweight and automatic method-level approach to estimate the energy consumption for Android apps. From the experimental results, we observe that the method-level estimation is a proper trade-off between accuracy and efficiency, that is useful to analyze the energy cost of the real-world and industry-sized applications.

There are some possible ways to improve our approach. We can make use of a test tool to generate more efficient and concise test inputs with high coverage. Besides, if we can generate the feasible execution sequences, then it is possible to predict the worst case energy consumption for an application, which will be a valuable metric for evaluating the quality of an app.

References

[1] http://developer.android.com/intl/zh-cn/tools/help/adb.html.
[2] http://ibotpeaches.github.io/apktool/.
[3] https://code.google.com/p/androguard/.
[4] https://code.google.com/p/signapk/.
[5] https://developer.qualcomm.com/software/trepn-power-profiler.
[6] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *FSE 2014*, pages 588–598, 2014.
[7] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *2010 USENIX Annual Technical Conference*, 2010.
[8] A. Ferrari, D. Gallucci, D. Puccinelli, and S. Giordano. Detecting energy leaks in Android app with POEM. In *PerCom Workshops 2015*, pages 421–426, 2015.
[9] A. Jindal, A. Pathak, Y. C. Hu, and S. P. Midkiff. Hypnos: understanding and treating sleep conflicts in smartphones. In *EuroSys 2013*, pages 253–266, 2013.
[10] K. Kim and H. Cha. Wakescope: Runtime wakelock anomaly management scheme for Android platform. In *EMSOFT 2013*, pages 27:1–27:10, 2013.
[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
[12] I. König, A. Q. Memon, and K. David. Energy consumption of the sensors of smartphones. In *ISWCS 2013*, pages 1–5, 2013.
[13] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for Android applications. In *ISSTA 2013*, pages 78–89, 2013.
[14] D. Li, A. H. Tran, and W. G. J. Halfond. Nyx: a display energy optimizer for mobile web apps. In *ESEC/FSE 2015*, pages 958–961, 2015.
[15] X. Li and J. P. Gallagher. An energy-aware programming approach for mobile application development guided by a fine-grained energy model. Technical report, Roskilde University, 2016.
[16] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, J. P. Gallagher, M. V. Hermenegildo, and K. Eder. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. 2015.
[17] Y. Liu, C. Xu, S. Cheung, and J. Lu. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *TSE*, 40(9):911–940, 2014.
[18] 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 *EuroSys 2012*, pages 29–42, 2012.
[19] 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 *MobiSys 2012*, pages 267–280, 2012.
[20] K. Paul and T. K. Kundu. Android on mobile devices: An energy perspective. In *CIT 2010*, pages 2421–2426, 2010.
[21] M. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-delay tradeoffs in smartphone applications. In *MobiSys 2010*, pages 255–270, 2010.
[22] M. L. Vásquez, G. Bavota, C. E. Bernal-Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk. Optimizing energy consumption of GUIs in Android apps: a multi-objective approach. In *ESEC/FSE 2015*, pages 143–154, 2015.