Science of Computer Programming 162 (2018) 132–147

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Lightweight energy consumption analysis and prediction for Android applications

Yan Hu[a], Jiwei Yan[b,d], Dong Yan[c,d], Qiong Lu[c], Jun Yan[b,c,d,*]

[a] School of Software, Dalian University of Technology, China
[b] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[c] Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, China
[d] University of Chinese Academy of Sciences, China

A R T I C L E I N F O

*Article history:*
Received 6 January 2017
Received in revised form 29 April 2017
Accepted 10 May 2017
Available online 22 May 2017

*Keywords:*
Android application
Energy consumption analysis
Energy prediction
Linear regression

A B S T R A C T

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 Android 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 analyze and predict the energy consumption for Android apps. We conduct the study from a method-level and API-level perspective. The method-level analysis gives developers facts about the energy consumption of the user methods in their apps, while the API-level analysis shows the energy consumption of Android APIs, which can help them make good decisions about how to choose appropriate APIs to improve the energy efficiency of an Android app. 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 method sequences or API sequences. The experiments on several real-world apps show that the proposed techniques have low overhead while persisting acceptable accuracy.

## 1. Introduction

Android application (app) market is expanding rapidly. This expansion is mainly driven by the development of mobile apps, which provide users with more interesting and diverse functionalities. However, one of the most frustrating things is that these applications often require a large amount of battery power, which drains the battery 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, researchers have explored some techniques on the platform layer [1]. However, their improvements are not sufficient for reducing the energy cost since a poorly written app will still die the battery out. Hence, it is very important to optimize the energy cost of Android apps. In order to perform energy

* Corresponding author at: State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China.
*E-mail address:* yanjun@ios.ac.cn (J. Yan).

http://dx.doi.org/10.1016/j.scico.2017.05.002
0167-6423/© 2017 Elsevier B.V. All rights reserved.

optimization, one thing we must do first is to analyze the energy consumption distribution in an app. It can help developers to know how the battery is utilized inside an app and give them hints about how to improve the app's 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. [2]. 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 [3] 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 [4] 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 and API (the basic unit in the program development) energy estimation appears to be more reasonable. Furthermore, most developers only care about the energy hole methods and APIs, 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 and API 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., [5,6]) 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 and APIs. After we execute the app under the monitoring of 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 Android APIs, 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 apply our approach to several 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 main contributions described in this paper are as follows.

- We propose a lightweight instrumentation technique in the register-based Dalvik bytecode instead of the source code.
- We propose an automatic approach to construct an accurate model to estimate the energy consumption of the methods, which can be used to predict the energy cost of the execution sequences.
- We extract API traces from method sequences, and build an API level energy model, which can be used to predict the energy cost of API traces.
- We conduct experiments on several real-world apps. The proposed approach can achieve acceptable prediction accuracy with low overhead.

The rest of this paper is organized as follows. In Section 2, we briefly introduce some related concepts and definitions. Details of our approach are discussed in Section 3. We explain the implementation of our approach in Section 4. Then we present the design of experiments and evaluate our approach in Section 5. The related works and conclusion are in Section 6 and 7, respectively.

## 2. Background

In this section, we present some background knowledge that will be used in the following parts. Firstly, we introduce the Dalvik virtual machine that is an essential part of Android app runtime, and the instrumentation technique to get runtime information. Then we describe the energy consumption of Android methods and APIs in the following two subsections. Next, we show how the Android GUI exploration works and introduce the Android power monitoring tool in the following subsection. Finally, we will introduce the linear regression used in our energy model construction and corresponding algorithms.

### *2.1. Dalvik bytecode*

The Android application is actually an Android Package (APK) file, which is a type of archive file with .apk as the filename extension. The source codes of an Android app are commonly compiled to Dalvik bytecode, which is a register-based instruction set. Our instrumentation is based on `smali`, a dex format of Dalvik bytecode. The `smali` file is composed of several statements, which comply with a series of rules, with the keyword `.class`, `.locals`, and so on. Take method as an example, it starts with `.method` and ends with `.end method`, between which the instructions form a method

```
.method protected onCreate(LAndroid/os/Bundle;)V
        . locals  2
        .param p1, “savedInstanceState”
        //other bytecode
.end method
```

**Fig. 1.** A `Smali` code snippet.

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

**Fig. 2.** Register distribution.

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

**Fig. 3.** A `Java` code snippet.

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

Note that the straightforward instrumentation may result in apps crashing 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.

## 2.2. Method energy consumption

There are many trivial application-specific and runtime environment factors that can influence the energy consumption of Android applications, such as user behavior, device temperature, different mobile devices with variant hardware platforms, etc. As such, even the energy consumption measurement values of the same method sequence could be different in two executions. In order to estimate app energy consumption effectively, diving too deep into those details is not a good option. Therefore, we decide to give a rough estimation at the method level, by ignoring the pre-mentioned minor factors. 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 $w(m)$ denotes average energy consumption by method $m$ which contains the instruction set including basic instructions but not the nested invocation methods. For example, as shown in Fig. 3, the energy consumption of method `a()` involves the energy consumption of the three instructions in `a()` excluding that of method `b()`. Furthermore, we define the method sequence energy consumption $e(s)$ denoting average energy consumption by method sequence $s$ which contains the set of methods $M_s$. Without consideration of measurement errors, we have

$$e(s) = \sum_{m \in M_s} w(m).$$

### 2.3. *API energy consumption*

The execution of the instrumented Android bytecode yields an execution sequence of program elements. In the previous subsection, we regard each element as a method call. Besides, we can make use of other program elements, such as API calls, to model the energy consumption of the execution sequence. Android APIs are major sources of energy consumption, and developers often pay much attention to the usage of them during the energy profiling and optimization phase in energy-aware development of Android apps.

M.L. Vásquez et al. [7] conducted an empirical study on the effect of API patterns to the energy consumption of Android applications. More than 100 energy-greedy Android APIs are collected from the benchmark applications in the experiments. The experimental results show that energy-greedy Android APIs are the main sources of power consumption for Android apps. In this paper, typical energy greedy Android APIs are divided into several categories according to their functionalities, including GUI and image manipulation, database, activity and context, services, Web, media and animation, datastructure manipulation, file manipulation, geo location, and networking. APIs related to GUI and image manipulation represent 60 percent of the energy greedy APIs.

Developers care about the energy consumption of Android APIs, as discussed in paper [7]. They have discussions about API energy consumption related topics on forums like StackOverflow [8]. The work shows that API energy consumption data is very useful in detecting energy related bugs, and also helpful to Android app energy optimization. GUI refreshing, database management, and Web related APIs like the constructors of WebView class, are found to be energy greedy.

### 2.4. *Android GUI exploration*

Most Android apps come with GUI interfaces. In order to reveal more behaviors of Android apps, we must use automatic tools to control the GUI, and make them generate variant traces.

Many researchers have explored the topic of Android GUI exploration. The methods include random testing, heuristic guided, and symbolic execution based ones. In our work, we will adopt the random testing technique, as other guided exploration techniques will add more runtime overhead, which is inappropriate for energy measurement.

Monkey [9] is the most basic random method for Android GUI testing. It is a generic built-in Android instrumentation. It randomly sends emulated events to the Android runtime, and does not need to worry about the state transition of GUI models. It also does not require any code knowledge and does not require any control over a specific app.

Robotium [10] is a popular Android UI testing framework. In order to perform Robotium based testing, a test app should be created and be associated to the app under test. The two apps are installed into the mobile device, and should be running in the same process. Thus, it is able to retrieve the UI structure of the app's current view, and send key strokes or tap events to control UI interactions by executing Robotium test scripts.

Monkey and Robotium can both be used to generate the event inputs to serve the purpose of generating a wide variety of method and API sequences. However, some drawbacks of Robotium make it not appropriate for our energy analysis task, which will be discussed in Subsection 5.2.

### 2.5. *Android power monitoring tool*

Power measurement is required for Android energy modeling. There are several ways to measure power consumption on mobile platforms:

1. read power or energy related data directly from mobile phone hardware. This is the most accurate method, but is restricted by specific phone hardware architecture;
2. read data from `/proc` and `/sys` files. It is very restricted in that data can only be read at the sampling points, not realtime;
3. read from the `BatteryStats` class. This method can only estimate power consumption at a very coarse level. It is also difficult to control the data sampling rate.

As pure software measurement approaches, like 2 and 3, is not very accurate, hardware based approaches are needed where accurate energy consumption data is required. One ideal option is that we simulate battery with real-life power source, and then measure the energy consumption in real-time. In this way, the result should be very accurate. However, setting up the power monitoring hardware environment would be very costly. Another way is to use specific hardware tool to connect to the mobile device, and collect detailed energy consumption data. Monsoon power monitor [11] is an example. But the hardware monitor is still very expensive.

Luckily, as the SnapDragon CPU provider, Qualcomm provides Trepn profiler [12]. It is a power and performance profiling application, which can be installed on target mobile devices to monitor the CPU usage and energy consumption. Trepn can monitor the energy consumption for a single Android app, which suites our approach very well. Therefore, it is chosen as our energy measurement tool.

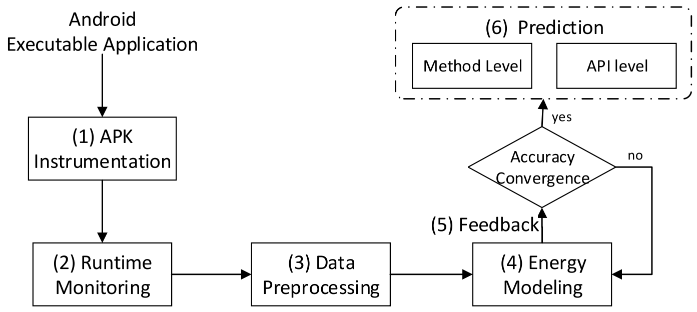

**Fig. 4.** The workflow of our approach.

### 2.6. *Linear regression analysis*

We use linear regression (LR) algorithms to generate the energy model. In this subsection, we will give a brief description of LR methods.

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

$$Y = XA^T.$$

The Simulated Annealing (SA) algorithm [13] can generate a solution to fit a proper assignment of $A$, which is a probabilistic technique for approximating the global optimum of a given function. Algorithm LARS Lasso [14] and Bayesian Regression [15] can also help to solve a linear model.

## 3. Approach

In this paper, we aim to perform energy consumption analysis and prediction of Android applications. Firstly, in order to given reasonable energy estimation, we propose a lightweight approach to generate energy consumption models at both method and API level. We then use the energy model to predict the energy cost of a specific execution sequence, which consists of a set of methods or Android APIs. In this section, we will explain the workflow of our approach in the first subsection, and give detailed descriptions of the six components in our approach from subsection 3.1 to subsection 3.5.

### 3.1. *Workflow of our approach*

Generating energy models is of first priority in our approach. Fig. 4 shows the workflow of our energy model generation process. The process consists of five modules: (1) APK Instrumentation, (2) Runtime Monitoring, (3) Data Preprocessing, (4) Energy Modeling, (5) Feedback and (6) Prediction.

In our approach, we do not require the source code of Android applications, but perform analysis on a set of Android APK files (the Android executable file). First of all, we statically instrument method monitoring probes into the APK file to track and log the sequences of method invocations. Then we install the instrumented application into the device, run the application and get the runtime path information and energy consumption data, which are stored in log files and the energy profiler's database, respectively. Next, we parse the collected raw data, and count the corresponding methods' invocation times. The energy consumption of each run is also calculated, based on the measured energy data stored in energy consumption database. The method invocation and energy consumption statistics serve as 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. That sequence can be method level or API level. To predict the energy cost of an API sequence, we transfer a given method trace to API trace by the function call relationship.

The 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 energy prediction values.

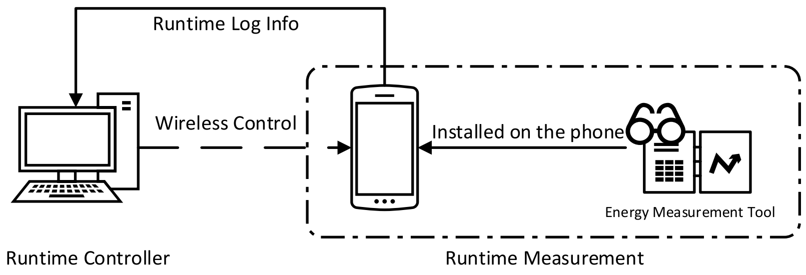

**Fig. 5.** Runtime monitoring.

### 3.2. *Runtime monitoring*

After the first step, we instrument an Android application with path profiling instructions. In this step, we will install the instrumented application into a physical device, execute it several times and measure the energy consumption of each run.

The runtime monitoring process is illustrated in Fig. 5. The monitoring task is fulfilled with the interaction between two major components: the runtime controller and the runtime measurement module.

The controller is built on the computer, while the measurement module 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.

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 inputs like touch, gesture, and so on. We can also use Robotium to control Android apps. However, as Robotium requires the installation of a control app on the device, it may cause extra power consumption, which can be rather unpredictable. Therefore, we finally choose Monkey as the controller.

For the energy measurement at runtime, we employ a tool that is specifically designed for power analysis of mobile applications: Trepn profiler 5.1 [12] 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. 5, after this step finishes, we will 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.

### 3.3. *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.

### 3.4. *Energy modeling*

In this step, we employ a linear regression algorithm on preprocessed energy monitoring data to derive an energy model.

After the data preprocessing step, we get a number of features. Let $x_{i,j}$ denote the invocation times of the $j$th $(1 \le j \le n)$ method in the $i$th test and $e_i$ denote the energy consumption for the same test. We can define the feature of the $i$th test as $\{x_{i,1}, x_{i,2}, \dots, x_{i,n}, e_i\}$. With $m$ executions, let $X = (x_{i,j})_{m \times n}$ be the method invocation times matrix, $E = (e_1, e_2, \dots, e_m)^T$ be the average energy measurement values vector. Assume that $W = (w_1, w_2, \dots, w_n)$ 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.

```
Lcn/domob/Android/j/a;−>a(Z)V −−−−>
Ljava/util/Timer;−>cancel()V
Lcn/domob/Android/j/a\$5;−><init>(Lcn/domob/Android/j/a;)V
LAndroid/app/Activity;−>runOnUiThread(Ljava/lang/Runnable;)V
Lcn/domob/Android/j/a;−>q()V
Lcn/domob/Android/j/c;−>b()V
Lcn/domob/Android/i/f;−>b(Ljava/lang/String;)V
```

(a) Original Invocations in a Method

```
Lcn/domob/Android/j/a;−>a(Z)V −−−−>
Ljava/util/Timer;−>cancel()V
LAndroid/app/Activity;−>runOnUiThread(Ljava/lang/Runnable;)V
```

(b) Trace of API Calls after removing user-defined methods

**Fig. 6.** Function calls in a method.

As we have discussed in Section 2.6, several algorithms can be used to solve the linear 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 $w_i$ $(1 \le i \le n)$ 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 $e^{-(c(S')-c(S))/KT}$, where $c(S')$ 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 $S$. The contrast experiment between SA and other algorithms please refer to section 5.5.

Finally, we obtain an energy model, from which we can get an estimation of the energy consumption of each method in a specific application.

### *3.5. API-level prediction*

In power-aware Android app development, developers tend to pay lots of attention to the task of choosing appropriate Android APIs. Android APIs are the hot spots in Android applications, in terms of energy consumption. Energy greedy Android APIs, like those related to file IO, network IO, GUI rendering, GPS, etc., will greatly affect the energy consumption of a user method in the target Android application. Therefore, it is useful to predict the energy consumption at API level. We can simply apply the energy model similar to that of subsection 3.4 on the sequence of API calls, and predicate energy consumption value of the new run of the application.

We take a further step to refine the energy model with API call information within each method run. In the runtime monitoring phase, we have already recorded the traces of method calls during each run of an application. The method trace can be directly transferred into API call traces. We make use of the decompiled smali files to obtain the API calls related to each method. In the smali files, `invoke` instruction is used for calling a method, which can help to construct a function call graph (FCG) of the target application. According to the package name, the called methods in FCG can be divided into two groups, user-defined methods and APIs. We filter out the calls of user-defined methods to get the trace of API calls. Then we put the new trace to linear regression analysis and use the training result to predict the energy consumption of new API traces. Take Fig. 6 as an example. The part (a) illustrates the invocations in the smali code of a method `a(Z)`. We remove the user-written method calls and get the trace of API calls shown in part (b), which only contains the calls to APIs.

## 4. Implementation of energy estimation and prediction

We have implemented our proposed approach in Python language with some ADB [16] scripts. The implementation is illustrated in Fig. 7.

Our approach does not require the source code of Android apps. We rely on APKTool [17] to decompile APK file and generate the disassembled code. Then we analyze the disassembled smali code and construct the FCG to obtain the relation

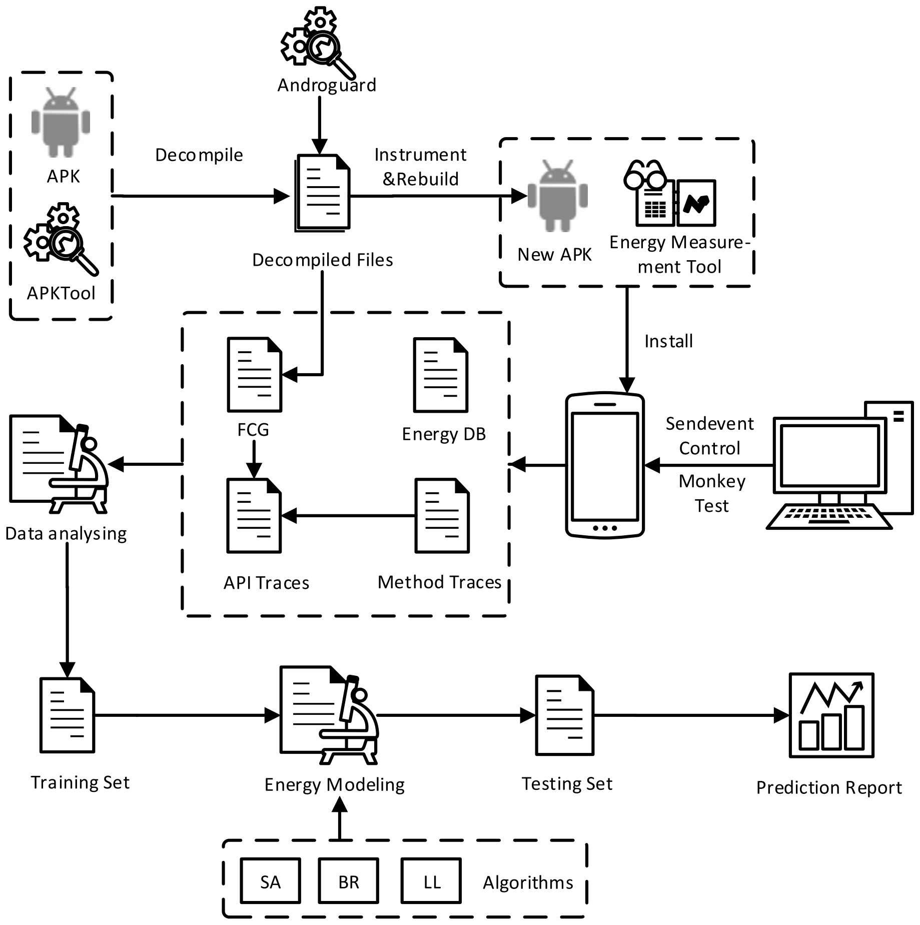

**Fig. 7.** Implementation of energy estimation and prediction.

of the user-defined methods and API calls. We also use Androguard [18], a static analysis tool for Android apps, to construct the control flow graph to help find the proper location of bytecode instrumentation. After instrumenting log statements into the disassembled code, we use APKTool to repack the Android APK file and use SignApk [19] to conduct the re-signature process if needed. After that, the instrumented APK file will be installed into an Android device. The Runtime Monitoring part monitors the execution of the instrumented APK file. In each measurement we should connect the Android device via wireless, open the power profiling tool Trepn, choose the target app in Trepn, start Trepn measurement and monkey testing, finish energy measurement and save database file. All these events are automatically executed by sending Android events using adb shell command `sendevent` and `monkey` to the Android device.

## 5. Evaluations

In this section, we present the experimental study of our proposed energy prediction approach. Firstly, we describe the setup of the experiments. Next, we compare the Monkey and Robotium based controller scheme to support our choice of Monkey to control the energy profiling experiments. Then, we present other details of evaluations, including: the overhead of instrumentation, the size of training set, the accuracy of regression analysis for method and API level prediction, and cross validation results. Finally, we analyze the method energy consumption distribution, and present the result in subsection 5.7.

### *5.1. Experimental setup*

We run the experiments on a Xiaomi 2S cellphone, with its CPU 1.7 GHz, 2 GB RAM, and the battery capacity 2000 mA h. 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 2.93 GHz with 8 GB RAM.

Table 1 lists the twelve 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

**Table 1**
Experimental applications.

| App | Category | Size | #C | #M | App | Category | Size | #C | #M |
|---|---|---|---|---|---|---|---|---|---|
| Saolei | Game | 475 | 132 | 724 | GoChess | Game | 1462 | 48 | 347 |
| Piano | Game | 721 | 449 | 2838 | Tomcat | Game | 53085 | 3953 | 25576 |
| CRadio | Media | 1568 | 668 | 4252 | QtRadio | Media | 9452 | 6216 | 44613 |
| CNTV | Video | 7961 | 905 | 6700 | WhtVideo | Video | 9844 | 2444 | 14248 |
| SgBrowser | Browser | 8417 | 2970 | 18542 | AoyouBrowser | Browser | 10847 | 4068 | 27511 |
| SgReader | Reader | 7446 | 3331 | 22519 | SuningReader | Reader | 12727 | 4174 | 25121 |

**Table 2**
Monkey and robotium.

| App | Time (s) | | Energy (mA h) | |
|---|---|---|---|---|
| | Monkey | Robotium | Monkey | Robotium |
| Saolei | 87 | – | 11.2 | – |
| Piano | 99 | 1442 | 12.6 | 147.3 |
| Cradio | 135 | – | 9.7 | – |
| Cntv | 198 | 7510 | 10.4 | 102.6 |
| SgBrowser | 154 | 5123 | 13.4 | 159.2 |
| SgReader | 205 | 6125 | 12.8 | 128.9 |

of the different profiles of energy consumption. We also give the category, size (KB), number of classes (#C) and number of methods (#M).

To evaluate the accuracy of our energy model, we adopt three statistical metrics: the multiple correlation coefficient ($R$), average value of relative error ($ARE$) and the standard deviation ($STDEV$) of relative error.

In statistics, $R$ is a well-known measure of how well a given variable can be predicted using a linear function of a set of other variables. It is defined as the Pearson correlation coefficient [20] 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 predictive variables are each method's energy consumption. The formula of $R$ is defined as follows,

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

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 $R$ takes values between 0 and 1 with a higher value close to 1 indicating a better predictability.

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

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

The statistical metric $STDEV$ can be used to quantify the amount of variation or dispersion of a set of data values. The lower the value of $STDEV$, the closer the data points are to the mean. In this case, we use it to measure the amount of variation or dispersion of the relative error in each energy prediction. The formula of $STDEV$ is defined as follows.

$$STDEV = \sqrt{\frac{\sum(\frac{|\hat{e_i} - e_i|}{e_i} - ARE)^2}{n-1}}$$

### 5.2. Monkey and robotium

We perform a comparative study of Monkey and Robotium controller, and Table 2 shows their time and energy measured after executing the same number of events. We set 20,000 as the number of events to be executed. As Robotium can not be used for random testing directly, we write a script to randomly generate and execute events based on the UI structure of the target app. In the Monkey testing, we set no throttle time to make it slow down, however, in Robotium testing, we set a 100 milli seconds of waiting time or else the app will easily crash.

In our experiments, two apps under test fail to run with Robotium instrumentation for energy measurement, while Monkey works well in all the cases. From the results, we can see that using Robotium will induce more energy overhead. That explains why we choose to use Monkey as the main controller in the energy modeling and prediction experiments.

**Table 3**
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% |
| SgBrowser | 739369 | 752799 | 1.8% | 135.08 | 138.34 | 2.4% |
| SgReader | 950227 | 966410 | 1.7% | 79.45 | 81.60 | 2.7% |

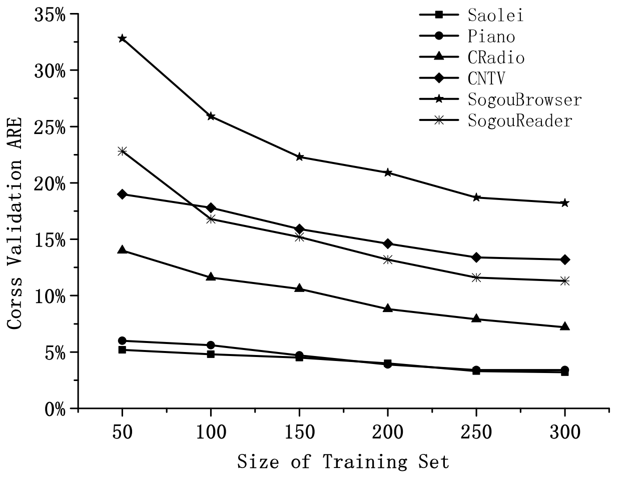

**Fig. 8.** Influence of training data set size.

## 5.3. Instrumentation overhead

As we have to insert instrumentation code to monitor method invocations, it is necessary to know in advance the overhead of the instrumentation on energy consumption. However, it is hard to measure the energy consumption of the instrumentation code directly. In order to estimate the instrumentation overhead, we compare the number of instructions of the apps' bytecode 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 3 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.

From the table, 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 on the energy consumption of the apps under test.

## 5.4. Size of training set

It is time-consuming to collect a 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 6 groups of experiments for each app, that adopt 50, 100, 150, 200, 250, 300 cases as the training set, respectively. We use a testing set containing 50 cases to compare the effect of the trained model for each app. The experimental results are shown in Fig. 8.

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 the figure, the error is relatively large (30% above) when the training set is composed of 50 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 being a suggested choice to be able to obtain a better model.

**Table 4**
Accuracy of training – user method.

| App | SA | | | BR | | | LL | | |
|---|---|---|---|---|---|---|---|---|---|
| | *R* | ARE | STDEV | *R* | ARE | STDEV | *R* | ARE | STDEV |
| Saolei | 0.79 | 6.2% | 11.4% | 0.40 | 6.4% | 13.3% | 0.39 | 6.3% | 13.3% |
| Piano | 0.89 | 5.7% | 5.0% | 0.89 | 5.6% | 7.5% | 0.85 | 5.3% | 7.1% |
| CRadio | 0.78 | 10.8% | 10.3% | 0.60 | 11.5% | 15.6% | 0.16 | 10.5% | 14.0% |
| CNTV | 0.66 | 17.4% | 17.7% | 0.58 | 29.5% | 83.2% | 0.31 | 3.2% | 70.4% |
| SgBrowser | 0.46 | 22.5% | 31.1% | 0.12 | 30.7% | 55.6% | 0.18 | 2.8% | 51.0% |
| SgReader | 0.82 | 14.6% | 29.3% | 0.64 | 16.7% | 32.3% | 0.69 | 2.0 % | 33.3% |
| GoChess | 0.46 | 22.3% | 17.2% | 0.11 | 24.5% | 32.3% | 0.34 | 19.9% | 25.6% |
| Tomcat | 0.65 | 14.3% | 16.0% | 0.50 | 25.5% | 77.7% | 0.21 | 17.1% | 51.9% |
| QtRadio | 0.90 | 14.8% | 12.9% | 0.85 | 20.1% | 26.0% | 0.61 | 1.1% | 1.3% |
| WhtVideo | 0.28 | 33.1% | 47.1% | 0.20 | 40.0% | 71.3% | 0.11 | 32.1% | 59.2% |
| AoyouBrowser | 0.85 | 9.5% | 8.5% | 0.20 | 18.4% | 47.9% | 0.57 | 14.0% | 28.2% |
| SuningReader | 0.50 | 20.0% | 19.6% | 0.69 | 23.7% | 38.9% | 0.23 | 19.4% | 31.7% |

**Table 5**
Accuracy of training – API.

| App | SA | | | BR | | | LL | | |
|---|---|---|---|---|---|---|---|---|---|
| | *R* | ARE | STDEV | *R* | ARE | STDEV | *R* | ARE | STDEV |
| Saolei | 0.79 | 5.8% | 6.4% | 0.42 | 6.3% | 13.3% | 0.39 | 6.3% | 13.2% |
| Piano | 0.88 | 5.9% | 5.2% | 0.87 | 5.8% | 7.7% | 0.83 | 5.8% | 7.7% |
| Cradio | 0.76 | 11.2% | 11.0% | 0.62 | 11.4% | 15.4% | 0.14 | 14.6% | 19.3% |
| CNTV | 0.65 | 19.3% | 26.7% | 0.60 | 29.6% | 83.7% | 0.69 | 29.4% | 84.0% |
| SgBrowser | 0.30 | 24.5% | 24.9% | 0.06 | 30.2% | 55.3% | 0.10 | 25.9% | 48.8% |
| SgReader | 0.62 | 22.5% | 22.1% | 0.53 | 22.7% | 35.7% | 0.55 | 26.6% | 37.8% |
| GoChess | 0.43 | 22.7% | 17.0% | 0.13 | 24.8% | 32.9% | 0.04 | 22.9% | 28.6% |
| Tomcat | 0.52 | 16.5% | 19.7% | 0.12 | 26.2% | 82.2% | 0.04 | 22.1% | 49.0% |
| QtRadio | 0.86 | 17.6% | 14.9% | 0.88 | 21.5% | 26.9% | 0.80 | 11.4% | 14.1% |
| WhtVideo | 0.26 | 38.5% | 58.9% | 0.16 | 40.0% | 71.3% | 0.03 | 25.6% | 45.9% |
| AoyouBrowser | 0.81 | 10.4% | 9.2% | 0.13 | 18.3% | 43.6% | 0.44 | 10.2% | 18.1% |
| SuningReader | 0.50 | 20.1% | 19.6% | 0.11 | 23.5% | 38.3% | 0.02 | 15.6% | 25.1% |

## 5.5. Accuracy of the regression analysis

To evaluate the efficiency and accuracy of our approach, we ran the target apps on the phone and analyzed the training accuracy of the produced energy model. We set 20,000 as the number of random events to send by Monkey to generate an execution sequence. We employ three regression algorithms (Simulated Annealing: SA, Bayesian Ridge: BR, LARS Lasso: LL) to train energy models. In the experiments of the first six apps, we adopt 250 cases as the training set with 20 cases as the reserved set to feedback. And as a comparison, we only use 100 cases in the following six apps. For the analysis accuracy, we consider $R$, $ARE$ and $STDEV$ to measure how well the energy model can predict for the training set. The accuracy result of training for method level model is depicted in Table 4 and the result for API level model is illustrated in Table 5.

In our approach, we define the method energy consumption as average of all the possible executions. However, in some cases, complex functions (like video decoding, web page parsing, etc.) may be implemented in a method, which could affect the accuracy of energy estimation results. As a consequence, we can see that several apps (e.g.,WhtVideo, GoChess and SgBrowser) have higher values of $ARE$ than other apps using all three regression approaches.

According to the comparison result of three regression approaches, Simulated Annealing approach can produce stable and more accurate results than the other two. And same as we have shown in Fig. 8, the results vary a lot with different size of training sets.

## 5.6. 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. For the first six apps, adopt 250 cases as the training set and 50 cases as the testing set, and adopt 80 cases and 20 cases respectably in the next six apps. Then by applying the trained energy model to the test data, we would obtain a set of energy costs. The cross validation results for method level prediction is illustrated in Table 6. As a comparison, the cross validation results of API level prediction is

**Table 6**
Cross validation accuracy – user method.

| App | SA | | BR | | LL | |
|---|---|---|---|---|---|---|
| | ARE | STDEV | ARE | STDEV | ARE | STDEV |
| Saolei | 4.2% | 3.3% | 16.1% | 30.2% | 15.8% | 31.0% |
| Piano | 4.0% | 3.4% | 13.1% | 11.3% | 15.0% | 13.9% |
| Cradio | 10.1% | 7.9% | 19.0% | 25.8% | 26.8% | 18.8% |
| CNTV | 16.9% | 13.4% | 29.0% | 43.3% | 41.7% | 81.4% |
| SgBrowser | 22.1% | 18.7% | 29.4% | 38.7% | 29.7% | 40.9% |
| SgReader | 14.6% | 11.6% | 27.4% | 51.4% | 25.6% | 41.9% |
| GoChess | 31.5% | 25.2% | 23.1% | 28.5% | 24.7% | 30.4% |
| Tomcat | 16.5% | 9.0% | 140.7% | 120.8% | 37.3% | 97.3% |
| QtRadio | 24.1% | 23.4% | 25.1% | 30.6% | 32.5% | 36.4% |
| WhtVideo | 32.7% | 19.8% | 30.5% | 42.5% | 28.9% | 41.2% |
| AoyouBrowser | 28.9% | 22.6% | 15.3% | 16.4% | 17.6% | 18.4% |
| SuningReader | 34.4% | 24.9% | 28.6% | 46.5% | 61.1% | 99.8% |

**Table 7**
Cross validation accuracy – API.

| App | SA | | BR | | LL | |
|---|---|---|---|---|---|---|
| | ARE | STDEV | ARE | STDEV | ARE | STDEV |
| Saolei | 3.8% | 3.2% | 16.0% | 29.5% | 16.0% | 31.22% |
| Piano | 4.0% | 3.4% | 13.3% | 11.6% | 14.3% | 13.22% |
| Cradio | 9.1% | 7.9% | 18.9% | 25.7% | 36.1% | 50.52% |
| CNTV | 17.8% | 14.2% | 28.5% | 42.1% | 28.9% | 41.22% |
| SgBrowser | 21.5% | 18.2% | 29.3% | 38.7% | 29.5% | 39.5% |
| SgReader | 24.8% | 16.5% | 33.0% | 50.0% | 31.8% | 40.9% |
| GoChess | 32.0% | 26.5% | 22.1% | 27.5% | 25.9% | 32.7% |
| Tomcat | 13.6% | 9.3% | 40.1% | 118.4% | 42.2% | 125.3% |
| QtRadio | 19.0% | 15.6% | 22.7% | 25.1% | 22.9% | 26.5% |
| WhtVideo | 25.2% | 20.2% | 30.5% | 42.5% | 33.1% | 49.6% |
| AoyouBrowser | 21.1% | 20.4% | 16.1% | 18.1% | 15.0% | 15.7% |
| SuningReader | 26.1% | 18.0% | 28.6% | 45.6% | 33.8% | 48.3% |

described in Table 7. From the results, we can see that with a proper size of the training set, our SA approach has much lower ARE and STDEV than the other two regression algorithms.

We also show the trend of the measured energy consumption by Trepn and the value predicted by energy model in Fig. 11. For each application, we list 50 groups of comparison experiments, each group of experiments are listed along the X-axis and the two different energy cost values in mA h (since the battery supplies a constant power voltage) are shown on the Y-axis. And the result shows that the trend of the measured energy curve and the two predicted energy curves (method and API level) in each application are very close.

### 5.7. *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. 9. 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' bytecode and conclude the classification results shown in Fig. 10. 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.

## 6. Related works

In recent years, power estimation and prediction becomes a big concern for the increasingly popular mobile platforms. Therefore, many researches aim to estimate the energy cost of mobile system from variant aspects.

Some researches try to perform power estimation with cycle-accurate simulators or full-system emulators. David et al. [21] proposed a statistical model based evaluation for the systematic analysis of energy saving techniques. R. Mittal et al. [22] proposed an emulator based energy consumption profiling method for the CPU, wireless communication (3G, Wi-Fi) and display. The idea is to exploit power-related parameter values obtained through the simulation to abstract the

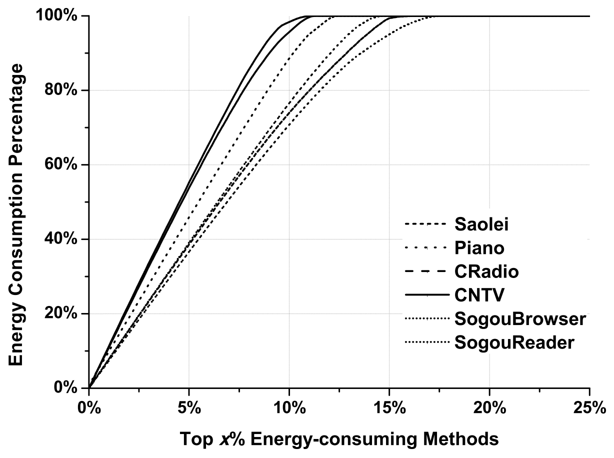

**Fig. 9.** Method energy consumption percentage.

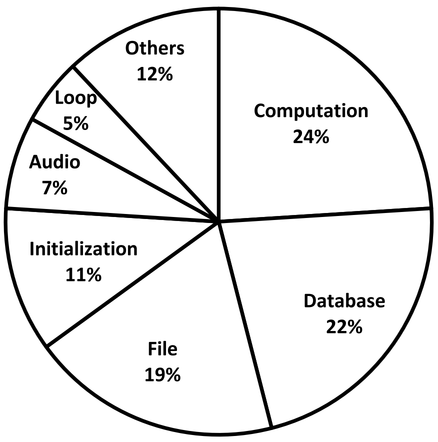

**Fig. 10.** Energy consumption percentage.

power consumption of a target system. The parameters include the number of CPU instructions executed, the number of cache hits/misses, the number of pipeline stalls and the number of branches taken. The accuracy of these methods can be improved by increasing the number of model-parameters accounting for the power consumption. The work in [22] targets the Windows Phone platform. It is not easy to port to Android devices, as it is difficult to maintaining the timing accuracy with different CPU simulation and very slow emulator speed.

Works based on operating system (OS) [23] or in virtual machines [24] require significant integration and modification to the runtime systems and is not portable. Furthermore, the workload of building models can be high.

Due to the drawbacks of the above mentioned emulator and OS level approaches, many recent researches focus on application level analysis instead. A. Pathak et al. track the activities of energy-consuming entities (Wifi, GPS etc.) when the app is executing on the mobile phone [25]. They compute the approximate energy consumption of applications and functions that invoke system calls.

A. Ferrari et al. [26] 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. The results of POEM show that with the fine-granularity instrumentation of POEM, the energy consumption is more than 1000 times of that on the clean application. The evaluation of POEM is conducted on self-made toy applications due to the substantial overhead.

U. Liqat et al. [27] 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 show that the accuracy of estimating at LLVM IR level (with average error 9%) is comparable to ISA level (with average error 3%). It suggests that it is viable to estimate energy consumption at bytecode level.

For the source level estimation, D. Li et al. proposed an approach, *vLens* [4], using a combination of program analysis and statistical modeling based on hardware power measurements. It aims at estimating source line level energy consumption

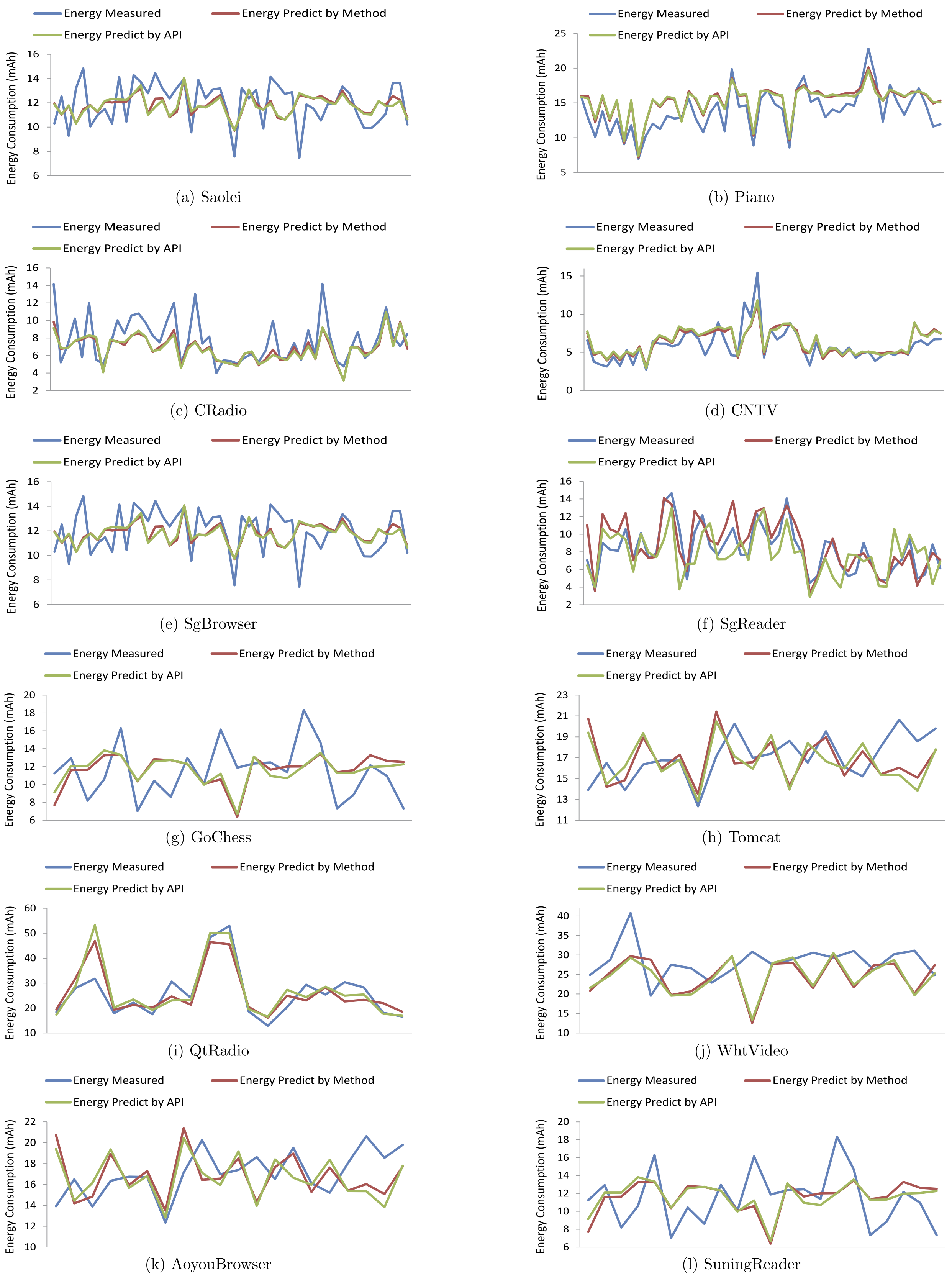

**Fig. 11.** Energy consumption prediction of execution sequences.

for mobile apps. For the selected 5 apps, the estimation error of *vLens* is around 3% to 9%. While in our approach, the estimation error is around 4% to 22% in the estimation (training) phase. However, those apps with estimation error more than 10% are huge apps with thousands of classes and more than 10,000 methods. Thus, the accuracy of our approach is comparable to *vLens*. Unlike our approach with the inputs generated automatically by Monkey, *vLens* needs to manually design input scenarios, which is labor intensive. As a poorly prepared input set may lead to low coverage of source lines that will decrease the model accuracy, the inputs for *vLens* have to be carefully adjusted. Besides, the number of coefficients of their model (same as that of the source lines) is usually very large for state-of-the-art real-world apps, which is a challenge for regression algorithms to fit and construct an accurate linear model.

There are also techniques detecting the energy performance bugs [28–30], 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 [31,32]. They believe that the user interfaces of a mobile app are related to the energy consumption. Li and Gallagher [33] 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.

Apart from the energy estimation and prediction techniques, the study of power measurement techniques is also closely related to our research. The general approach for these techniques is to use a power measurement device, such as the LEAP2 [34] or Monsoon [11] power meters, that can sample energy measurements at a certain frequency. These measurements are then combined with software based techniques to provide useful information to software developers. C. Sahin and colleagues [35] map energy consumption to different component level design patterns. Their work explored the correlations between energy consumption and the use of different design patterns. At a higher level, J. Flinn and colleagues [36] measured the energy consumption of applications and mapped the energy to individual OS processes. This was done, in part, by instrumenting the operating systems.

L. Zhang et al. [37] present the methods that use a power model to estimate the power consumption of the entire device, using operating time of each part of the device as parameter. However, it is difficult to identify the contribution of an application to the total power consumption because smart phones can run several processes simultaneously.

In some fine-grained research studies, hardware based measurement dominates, whose measured power is very accurate indeed. In comparison, software energy estimation is less accurate because software power meters are unable to measure and record energy fast enough compared to hardware ones. The hardware approach for power measurement demands specific hardware toolboxes, which can be very costly. A less costly, and more generic software based measurement approach, like Trepn, is more reasonable for power-aware mobile app development.

## 7. Conclusion

Energy consumption is a serious problem for pocket devices. We investigated this problem and proposed a lightweight and automatic energy consumption analysis for Android apps. Our approach is based on bytecode level instrumentation and software-based runtime energy monitoring. We proposed a Simulated Annealing based linear regression algorithm to generate the energy model. Energy analysis and prediction are conducted in two aspects. We provide the method-level energy model, to help developers get a concrete view of the energy consumption of the new methods during their app development. We also conduct an smali analysis, and generate an API-level energy model to give insights about the energy consumption of Android APIs. In the experiments, we adopt the more effective Monkey based UI exploration to generate a rich set of method and API sequences. We also compared the proposed Simulated Annealing based linear regression method with two standard regression algorithms implemented in the popular machine learning package scikit-learn. From the experimental results, we observe that our proposed approach is a proper trade-off between accuracy and efficiency, and thus, is practical and feasible to be used in energy-aware development of real-world and industry-sized mobile applications.

## Acknowledgements

This work is supported by National Natural Science Foundation of China (Grant No. 61672505, 61300017) and the National Key Basic Research (973) Program of China (Grant No. 2014CB340701).

## References

[1] K. Paul, T.K. Kundu, Android on mobile devices: an energy perspective, in: CIT 2010, 2010, pp. 2421–2426.
[2] A. Carroll, G. Heiser, An analysis of power consumption in a smartphone, in: 2010 USENIX Annual Technical Conference, 2010.
[3] A. Banerjee, L.K. Chong, S. Chattopadhyay, A. Roychoudhury, Detecting energy bugs and hotspots in mobile apps, in: FSE 2014, 2014, pp. 588–598.
[4] D. Li, S. Hao, W.G.J. Halfond, R. Govindan, Calculating source line level energy information for Android applications, in: ISSTA 2013, 2013, pp. 78–89.
[5] I. König, A.Q. Memon, K. David, Energy consumption of the sensors of smartphones, in: ISWCS 2013, 2013, pp. 1–5.
[6] M. Ra, J. Paek, A.B. Sharma, R. Govindan, M.H. Krieger, M.J. Neely, Energy-delay tradeoffs in smartphone applications, in: MobiSys 2010, 2010, pp. 255–270.
[7] M.L. Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M.D. Penta, D. Poshyvanyk, Mining energy-greedy API usage patterns in Android apps: an empirical study, in: 11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31–June 1, 2014, Hyderabad, India, 2014, pp. 2–11.

[8] Stackoverflow, http://stackoverflow.com.
[9] Monkey, http://developer.android.com/tools/help/monkey.html.
[10] Robotium, http://code.google.com/p/robotium/.
[11] Monsoon power meter, https://www.msoon.com/LabEquipment/PowerMonitor/.
[12] Trepn, https://developer.qualcomm.com/software/trepn-power-profiler.
[13] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680, http://dx.doi.org/10.1126/science.220.4598.671, http://science.sciencemag.org/content/220/4598/671.full.pdf, http://science.sciencemag.org/content/220/4598/671.
[14] Lars lasso, http://scikit-learn.org/stable/modules/linear_model.html#lars-lasso.
[15] Bayesian regression, http://scikit-learn.org/stable/modules/linear_model.html#bayesian-regression.
[16] Adb, http://developer.android.com/intl/zh-cn/tools/help/adb.html.
[17] Apktool, http://ibotpeaches.github.io/Apktool/.
[18] Androguard, https://code.google.com/p/androguard/.
[19] Signapk, https://code.google.com/p/signapk/.
[20] Pearson correlation coefficient, https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.
[21] D. Basile, F. Di Giandomenico, S. Gnesi, Model-Based Evaluation of Energy Saving Systems, Springer International Publishing, Cham, 2017, pp. 187–208, http://dx.doi.org/10.1007/978-3-319-44162-7_10.
[22] R. Mittal, A. Kansal, R. Chandra, Empowering developers to estimate app energy consumption, in: The 18th Annual International Conference on Mobile Computing and Networking, Mobicom'12, Istanbul, Turkey, August 22–26, 2012, 2012, pp. 317–328.
[23] T. Li, L.K. John, Run-time modeling and estimation of operating system power consumption, in: Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2003, June 9–14, 2003, San Diego, CA, USA, 2003, pp. 160–171.
[24] A. Kansal, F. Zhao, J. Liu, N. Kothari, A.A. Bhattacharya, Virtual machine power metering and provisioning, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10–11, 2010, 2010, pp. 39–50.
[25] A. Pathak, Y.C. Hu, M. Zhang, Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof, in: EuroSys 2012, 2012, pp. 29–42.
[26] A. Ferrari, D. Gallucci, D. Puccinelli, S. Giordano, Detecting energy leaks in Android app with POEM, in: PerCom Workshops 2015, 2015, pp. 421–426.
[27] U. Liqat, K. Georgiou, S. Kerrison, P. López-García, J.P. Gallagher, M.V. Hermenegildo, K. Eder, Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR, in: Foundational and Practical Aspects of Resource Analysis – 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015, 2015, pp. 81–100, Revised Selected Papers.
[28] K. Kim, H. Cha, Wakescope: runtime wakelock anomaly management scheme for Android platform, in: EMSOFT 2013, 2013, pp. 27:1–27:10.
[29] Y. Liu, C. Xu, S. Cheung, J. Lu Greendroid, Automated diagnosis of energy inefficiency for smartphone applications, IEEE Trans. Softw. Eng. 40 (9) (2014) 911–940.
[30] A. Pathak, A. Jindal, Y.C. Hu, S.P. Midkiff, What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps, in: MobiSys 2012, 2012, pp. 267–280.
[31] D. Li, A.H. Tran, W.G.J. Halfond, Nyx: a display energy optimizer for mobile web apps, in: ESEC/FSE 2015, 2015, pp. 958–961.
[32] M.L. Vásquez, G. Bavota, C.E. Bernal-Cárdenas, R. Oliveto, M.D. Penta, D. Poshyvanyk, Optimizing energy consumption of GUIs in Android apps: a multi-objective approach, in: ESEC/FSE 2015, 2015, pp. 143–154.
[33] X. Li, J.P. Gallagher, An energy-aware programming approach for mobile application development guided by a fine-grained energy model, CoRR, arXiv:1605.05234.
[34] D. McIntire, T. Stathopoulos, W.J. Kaiser, etop: sensor network application energy profiling on the LEAP2 platform, in: Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN 2007, Cambridge, Massachusetts, USA, April 25–27, 2007, 2007, pp. 576–577, http://doi.acm.org/10.1145/1236360.1236448.
[35] C. Sahin, F. Cayci, I.L.M. Gutiérrez, J. Clause, F.E. Kiamilev, L.L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: First International Workshop on Green and Sustainable Software, GREENS 2012, Zurich, Switzerland, June 3, 2012, 2012, pp. 55–61.
[36] J. Flinn, M. Satyanarayanan, Powerscope: a tool for profiling the energy usage of mobile applications, in: 2nd Workshop on Mobile Computing Systems and Applications, WMCSA '99, February 25–26, 1999, New Orleans, LA, USA, 1999, pp. 2–10, http://dx.doi.org/10.1109/MCSA.1999.749272.
[37] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R.P. Dick, Z.M. Mao, L. Yang, Accurate online power estimation and automatic battery behavior based power model generation for smartphones, in: Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2010, Part of ESWeek '10 Sixth Embedded Systems Week, Scottsdale, AZ, USA, October 24–28, 2010, 2010, pp. 105–114.