Detect and optimize the energy consumption of mobile app through static analysis: an initial research

Jingtian Wang[1, 2] Guoquan wu[1] Xiaoquan Wu [1, 2] Jun Wei[1]

[1] Technology Center of Software Engineering

Institute of Software, Chinese Academy of Sciences

[2]Graduate University of Chinese Academy of Sciences

{wangjingtian10, gqwu, wuxiaoquan07, wj}@otcaix.iscas.ac.cn

ABSTRACT

Although the market for smartphones is growing rapidly, their utility remains severely limited by the battery life. As such, much research effort has been made to understand the power consumption of the application running on mobile devices. However, dynamic profiling tools need to run on the customized android platform, making them not suitable for ordinary mobile app developers. To address this limitation, this paper proposed a light-weight approach to find possible I/O energy wasting code in Android apps through static program analysis technique. We also provide a case study to evaluate the effectiveness of our approach.

General Terms

Algorithms, Measurement, Performance, Experimentation

Keywords

Mobile app, Energy profiler, Static analysis

1. INTRODUCTION

Friendly, open-source, and abundant third-party applications, all these characteristics of Android platform have attracted many users and developers. Despite the immense popularity of smartphones and app market, their utility remains severely limited by battery life. However, battery life limits the utility. In terms of batter technology, the trends indicate that battery's ability will not have great improvement in recent years [9]. Thus, energy optimization for Android platform is of critical importance.

Hardware components are the sources of energy consumption on smartphones, such as LCD screen, CPU, WiFi, NIC, GPS, and Secure Digital Card (sdcard for short). All these components' energy costing behavior is triggered by the applications running on the smartphones besides background services. However, many apps developed so far are not energy-aware. Most developers focus on the app's functionality while ignoring the energy consumption. Thus, developers need a tool to help them to develop energy-aware mobile apps.

In [1][2], Pathak et. al proposed an energy profiler, called Eprof, to understand and optimize the energy consumption of mobile apps. The characteristic of Eprof is that it can get the energy consumption at the granularity of methods or threads in an application based on system call tracing. Using Eprof, the authors also observed that most energy in an app is spent on accessing I/O components, and tail energy typically accounts for the largest fraction of the I/O energy. To better understanding the energy consumption, a new energy accounting presentation called *I/O energy bundle*, is proposed to capture the energy dissipation at high level. Bundle is defined as a continuous period of an I/O component, and corresponds to the process from an I/O component energy base state to next base state in its power FSM. However, this dynamic approach needs to modify the android platform to log all system calls in order to trace method execution, and it's not easy to implement such fine-grained energy profiler tool. Since all system calls of the I/O components can be reflected through the API of the I/O components provided by Android SDK, we start with static program analysis of the app's source code instead of modifying the android platform to understand the energy behavior of the I/O components.

For energy optimization, as most I/O components have tail power state which keeps a high power state after finishing the I/O activities, what we can do is to reduce the number of tail energy occurrence. That is, we can put I/O operations that will cause energy tail as close as possible, in order to overwrite the previous operation of the tail effect. Thus, we define the appearance of periodic I/O operations as I/O energy wasting pattern.

The approach we offered can help developers to locate energy wasting pattern, and consequently optimize smartphone application. The main steps that finding energy wasting pattern in mobile app include: 1) locating routines that implement I/O operation according to the I/O operation list we collected; 2) getting the I/O routines' reverse control flow by static program analysis, in order to confirm whether there are adjacent and continuous I/O operations, 3) analyzing statements between adjacent I/O operations through data flow analysis, which can benefit developers to analyze whether there are operations between adjacent I/O can be pre-processed or centralized process to overwrite tail power state.

I/O component includes 3G, WiFi, sdcard, LCD screen etc. We have done some WIFI I/O experiments, the results suggest that WIFI I/O operation don't have obvious tail power state, the WIFI component quickly get back to the low power state after I/O execution. According to our strategy, it has little space to optimize. In addition, for LCD screen, its usage results from users' operation, if the app do not acquire screen wakelock[6], it have

Internetware'12, Octorber 29–31, 2012, Qingdao, China.

little direct influence on screen energy consumption. Thus, we mainly focus on 3G and sdcard in this paper.

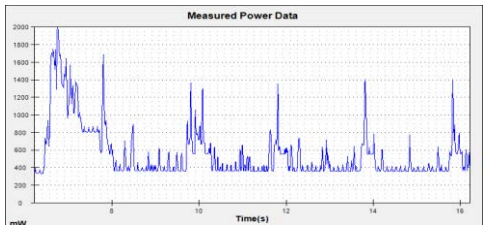
This paper is organized as follows. Section 2 presents the observations we have founded during abundant use and analysis of applications, then we introduce the method to find possible I/O energy wasting pattern in Section 3. In Section 4, a case study is presented to verify whether the method mentioned in section 2 can bring energy reduction. We present related works in section 5. In the end we conclude the paper.

2. OBSERVATIONS

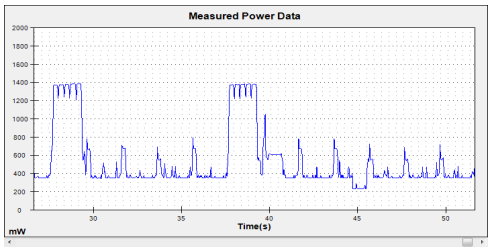
In [2], I/O energy bundles present intuitive view of how I/O routines executed in apps and find there are always some I/O operations which occurs repeatedly. In order to better understand I/O characteristics in smartphone apps, we used apps that contain the I/O operations on sdcard or network, and decompile the Android bytecode to analysis, and have the following observations.

First, numbers of operations exist during two adjacent I/O operations which delayed I/O component tail power state. Many apps such as News, Microblog, Social Network, E-commerce app, use I/O to load resources(pictures, texts etc.) In this paper, we consider the picture resources, since the size of text resource is small, and apps can load all text resource once which generally does not lead to multiple I/O operations. As Android adopts event-driven programming model, many apps choose to download pictures when users scroll on the current page. Also, the energy-unaware app development can lead to periodic I/O at runtime. Thus periodic I/O is common.

Periodic I/O operations can result in many tail power states, as presented in fig.1 (a) [9], the each summit of the curve indicates each real I/O activities, while the subsequent curve did not immediately return to the initial level, and periodic I/O brought multiple transition stage. Intuitively, batching multiple I/O activities is the way to reduce tail power state, which can cut down multiple tail power state to the last one tail state, as presented in fig.1(b), the summit of the curve followed by another summit, the intermediate transition stages are overwritten.



(a)



(b)

Fig.1 Energy consumption of I/O

Second, the periodic I/O operations always appeared in loops, like PhotoSlide, PhotoBucket, NYTimes. Apps need to execute multiple I/O to load plenty of resources, and putting the several uniform operations in loops is efficient.

These two observations can help us to find possible energy wasting pattern and to offer optimizing suggestions. Based on these observations, in section 3, we propose a light weight approach for app developers to find the potential inappropriate design of I/O operations in apps source code.

3. APPROACH

The process to find possible energy wasting pattern mainly include three steps (see figure 2):

1) **Locating I/O operations**. The idea is that searching source code to judge whether there are statements matched the operations in the list we have collected (see Section 3.1);

2) **Searching specific code pattern**. I/O energy wasting pattern is defined as the occurrence of periodic I/O operations inside loops. We use static analysis techniques to search this pattern. If the pattern matched, then obtain the flow graph between two adjacent I/O operations (Section 3.2);

3) **Analysis**. This step uses data flow analysis to analyze the flow graph generated in the second step, then offer advices of energy-aware optimizing, the details are described in Section 3.3.

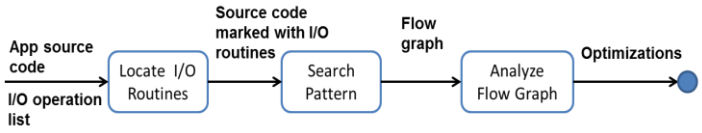


Figure 2. The process of method

3.1 I/O Operations Locating

In our experience, I/O activities are always accompanied by several fixed operations. It needs to track those statements to locate the I/O routines. Therefore, we collect I/O component (NIC, sdcard) operation list in which operations will appear when execute I/O.

3.1.1 Network

In android SDK, there are two types of network connections operating, listed in Table 1. After building connection, apps may download or upload resources with servers.

On android platform, each resource needs a connection since the unique URL is an attribute of the connection. Download a lot of resources will bring periodic I/O.

	Source Code
I	DefaultHttpClient http = new DefaultHttpClient(); HttpGet method = new HttpGet(url); HttpResponse response =http.execute(method);
II	URL url = new URL(uri); HttpURLConnection connection = (HttpURLConnection) url.openConnection(); connection.connect();

Table 1. Network I/O operation list

3.1.2 sdcard

Sdcard is mounted as the file system, the write/read operation was the same as natural file system. File path is the unique difference, the default path of sdcard root directory is "/sdcard/*". Apps need to built directory in sdcard to store data. The common methods to find specific file include: a. Find the file according to the absolute path.

```
Environment.getExternalStorageDirectory();
```

b. Obtain the metadata of files in specified type by MediaScanner, to enumerate all files information on external storage, such as name, size, author, and file path. The necessary operation is listed as following:

```
Cursor managedQuery(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder);
```

The above operations are only the preparation for sdcard I/O operations. We already pinpoints in previous paragraph that the operation of files on sdcard are the same as operating of natural file system. The actual I/O operations are listed in table 2.

App←sdcard	Inputstream.read() FileInputStream.read() BitmapFactory.decodeFile()
App →sdcard	Outputstream.write() FileOutputStream.wirte() Bitmap.compress()

Table 2. sdcard I/O operation list

3.2 Specific Code Pattern Searching

Firstly, locate routines that execute I/O operations according to operation lists shown above.

Secondly, it needs to judge whether the apps has a loop which contains multiple continuous and uniform I/O operations. We use reverse search of control flow from the I/O method located in the first step, namely, generating the control flow from the bottom up. 1) when another uniform I/O is found, add the "traversed" tag and continue to search up. If the marked statement exceeded two times, we can draw the conclusion that there is a specified energy wasting pattern in the application. 2) if there are no related I/O routines along the control flow till the root method, we can decide that this application doesn't contain this specified pattern. Otherwise, for the I/O operations that are not the same with the marked I/O beforehand, we start to search another path from these I/O operations.

3.3 Flow Graph Analyzing

To obtain flow graph between two adjacent I/O after locating continuous I/O operations, then we adopt reaching definitions (RD for short) data-flow analysis, which statically determines which definitions may reach a given point in the code, to decide whether other operations are related to I/O operations and how to optimize the implementation of I/O operations. There are three possible situations in this step:

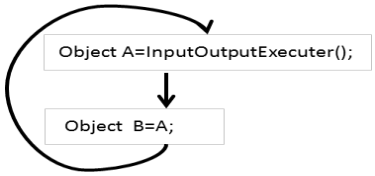


Figure 3. The first case

InputOutputExecuter in figures represents I/O operation. In this situation (see figure 3), the result returned by I/O operation is used by the subsequent operations. The suggested optimization is to batch the I/O operations, that is prefetching the resource that multiple I/O transferred. Taking loading pictures by I/O as an example, developers can prefetch a group of pictures instead of one picture a time. The number of the prefetched pictures is related to the size of pictures and the memory that the application could employ. Different scenarios have different settings and it needs to be determined by developers.

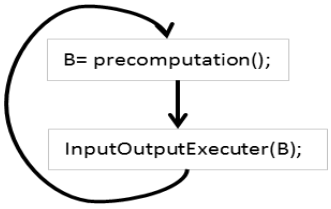


Figure 4. The second case

In this case (see figure 4), the parameters of I/O routines come from the previous operations. This situation limited the order of operations that the I/O operations cannot be pre-processed. The previous operations could be obtaining path of the file on sdcard, getting the URL of network resources, or assigning the location for the resources will be loaded. The optimization strategy for this case is first to process/calculate the parameters that I/O operations need, then to bundle multiple I/O operation in order to overwrite tail power states.

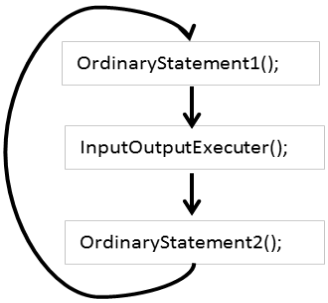


Figure 5. The third case

We find the situation, illustrated in figure 5, through data flow analysis. There are some operations which are not related to the I/O routines occurred in the loop. In typical usage, there are time-setting operations such as Thread.*sleep* function or time scheduler. For this case, I/O batching strategy needs to take into account the time factor. The quantity of I/O operations in each batching should not be too large in order to guarantee the accuracy of the trigger time. There is also another situation that the unnecessary operations are brought by developers' oblivious developing manner. The corresponding optimization strategy is to move the

uncorrelated operations out of the loop in order to reduce the count of operations between two continuous I/O operations.

However, the optimization advice is application-specific. The present approach needs developers' participation to keep the balance between user experience and energy-saving purpose. We leave automatic accurate optimization as future work.

4. CASE STUDY

In case study, we use the decompile tool dex2jar [11] and jd-gui [12] to get the app source code. While in the actual scenarios, developers can use the approach on source codes directly.

We use Photoslide app as an example to demonstrate the previous workflow. Photoslide is a slide show application which can transfer pictures on smartphones into a slide show.

First, locate the I/O routines according the I/O operation list. In Photoslide source code, there is a unique I/O module, *ViewerUtil.LoadPicFirst*

```
ViewerUtil.LoadPic()
{
    .......
    localObject = BitmapFactory.decodeFile(paramString, localOptions);
    .......
}
```

Listing 1. Photoslide I/O routine operation list

Then, get the bottom-up control flow at the beginning of *ViewerUtil.LoadPic*. In the traversal process, the marked I/O routine (*ViewerUtil.LoadPic*) periodically appeared many times, exceeding the threshold we set. We can decide that in Photoslide there is a possible I/O energy wasting implementation. The control flow is presented in figure 6.

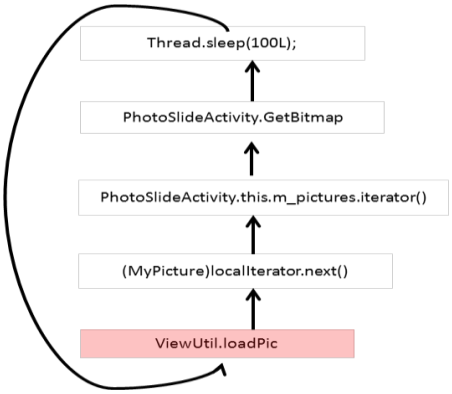


Figure 6. Control flow of corresponding entry

Finally, we analyze the local control flow between two adjacent I/O operatings(*BitmapFactory.decodeFile*), shown in figure 7. We do RD analysis for the flow graph to find operations related to I/O. *GetBitmap* function is to obtain the file path. The result returned from this function acts as the parameter of *decodeFile*, which matches the second pattern in section 3.2. The parameter of *BitmapDrawable.draw* is related to the I/O operation result, applications can batch several I/O operations before *BitmapDrawable.draw*, and this match the first pattern. Thus, the app could prefetch all file paths and then bundle multiple I/O operations to reduce energy consumption. Note that, the sleep function appears in operation sequence suggest that the count of the bundle I/O operation group should not be too much to guarantee the accuracy of the trigger time.

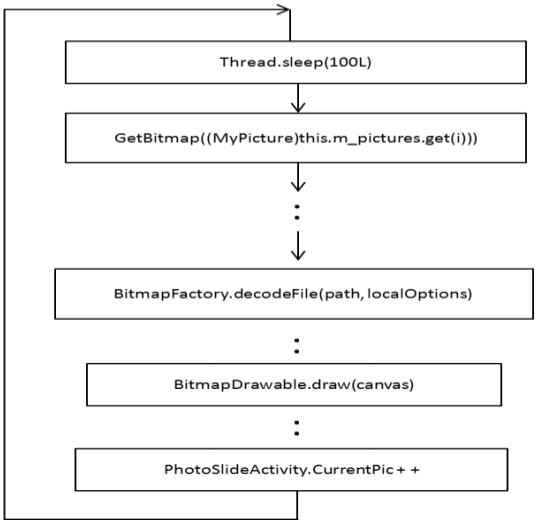


Figure 7.control flow between adjacent I/O

To verify whether the optimization category can bring energy reduction, we rewrote the slide function in Photoslide, and design another application according to the optimization advice. The two applications both load 20 pics from sdcard.

APP I load one picture once a time, stop 2 seconds, and load next pictures. The energy consumption shown as figure 8.

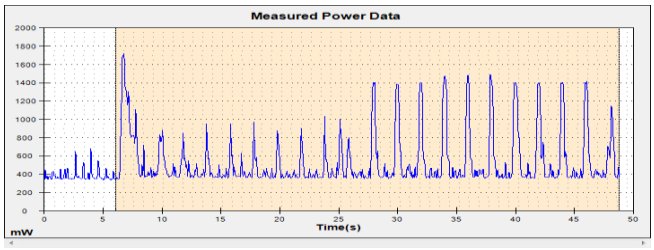


Fig 8. Result of App I (sdcard)

APP II, the optimized application load a group of pictures once a time(in this experiment we set the number of group is five), then load picture from memory, stop 2 seconds, load next picture from memory. When the group was complete shown, the app load another group. The energy consumption is shown as figure 9.

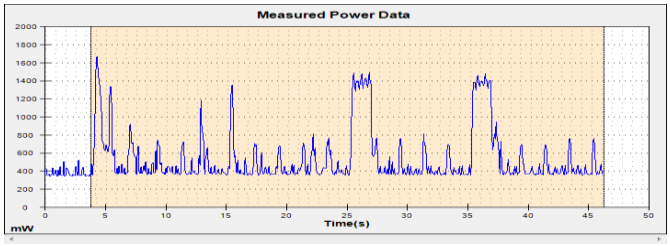


Fig 9. Result of App II (sdcard)

The result shows that in the table 3, app II saves almost 20% energy.

We also rewrote the modules that responsible for downloading resources from network, which is frequently used by kinds of apps, such as News, Microblog and E-commerce app, to verify whether the optimization category applies to network I/O. Both two

applications download 10 pictures by 3G.

	Description		Average energy consumption
App I	Load 20 pics from sdcard	Load one picture once	2023uAH
App II		Load a group(five) pictures once	1614uAH

Table 3. Results of sdcard I/O experiment

App I download one pictures once, does computation 300ms, and then downloads next picture. The energy consumption is shown as figure 10.

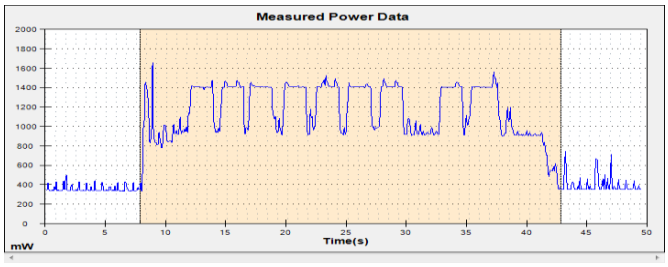


Fig 10. Result of App I(3G)

App II, the optimized one downloads all 10 pictures once a time, and do computation 3000ms, The energy consumption is shown as figure 11.

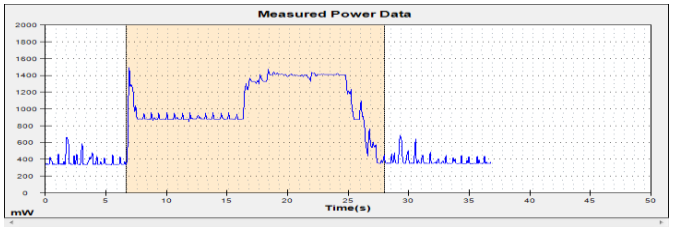


Fig 11. Result of App II(3G)

The result shows that in the table 4, app II saves almost 44% energy.

	Description		Average energy consumption
App I	Download 10 pics from network	Load one picture once	3107uAH
App II		Load all pictures once	1740uAH

Table 4.Results of 3G I/O experiment

We can find that batching 3G/sdcard I/O operation can bring much energy saving through above experiment result.

5. RELATED WORK

There are many researches related to energy profiling and optimizing. Our work is to find specific I/O energy inefficiency pattern by static analysis. Characterizing and detecting no-sleep bugs [6] also use static program analysis method to find no-sleep code pattern, the problem they solved is different from us. PowerTutor[7], and Sesame[8] can get every application's energy consumption by dynamic profiling, but they cannot get intra-application energy cost info, therefore they cannot offer developers energy-aware optimization.

6. CONLUSION

In this paper, we offered a light-weight approach to help developers to locate energy wasting pattern and provide suggested optimizations.

However, Java object reference and intent resolution, the indirect control transfer mechanism, pose challenge for static analysis to get complete control information [6]. In future work, we will add dynamic analysis to our work to analyze all routines reference.

7. REFERENCES

[1] Abhinav Pathak, Y. C. H., Ming Zhang (2011). Fine-Grained Power Modeling for Smartphones Using System Call Tracing. EuroSys.

[2] Abhinav Pathak, Y. C. H., Ming Zhang (2012). "Where is the energy spent inside my app? Fine grained Energy Accounting on Smartphones with Eprof." EuroSys.

[3] Feng Qian, Z. W., Alexandre Gerber (2011). "Profiling Resource Usage for Mobile Applications: A Cross-layer Approach." MobiSys.

[4] "Transferring Data Without Draining the Battery" URL: http://developer.android.com/training/efficient-downloads/index.html

[5] Arni Einarsson, Janus Dam Nielsen. A Survivor's Guide to Java Program Analysis with Soot

[6] Abhinav Pathak, A. J., Y.Charlie Hu (2012). "What is keeping my phone awake? Characterizing and Detecting No-Sleep Energy Bugs in Smartphone Apps." MobiSys.

[7] Lide Zhang, B. T., Zhiyun Qian, Zhaoguang Wange (2010). "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones." CODES+ISSS.

[8] Mian Dong, L. Z. (2011). "Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems." MobiSys.

[9] Eduardo Cuervo, A. B. (2010). "MAUI: Making Smartphones Last Longer with Code Offload." MobiSys.

[10] "Monsoon Power Monitor." URL: http://www.msoon.com/LabEquipment/PowerMonitor/

[11] dex2jar URL: http://code.google.com/p/dex2jar/downloads/list

[12] jd-gui URL: http://java.decompiler.free.fr/