Detecting and Diagnosing Energy Issues for Mobile Applications

Xueliang Li
College of Computer Science and Software Engineering
Shenzhen University
Shenzhen, China

Yuming Yang
College of Computer Science and Software Engineering
Shenzhen University
Shenzhen, China

Yepang Liu
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China

John P. Gallagher
Department of People and Technology
Roskilde University
Roskilde, Denmark
IMDEA Software Institute
Madrid, Spain

Kaishun Wu
College of Computer Science and Software Engineering
Shenzhen University
Shenzhen, China

ABSTRACT

Energy efficiency is an important criterion to judge the quality of mobile apps, but one third of our randomly sampled apps suffer from energy issues that can quickly drain battery power. To understand these issues, we conducted an empirical study on 27 well-maintained apps such as Chrome and Firefox, whose issue tracking systems are publicly accessible. Our study revealed that the main root causes of energy issues include unnecessary workload and excessively frequent operations. Surprisingly, these issues are beyond the application of present technology on energy issue detection. We also found that 25.0% of energy issues can only manifest themselves under specific contexts such as poor network performance, but such contexts are again neglected by present technology.

In this paper, we propose a novel testing framework for detecting energy issues in real-world mobile apps. Our framework examines apps with well-designed input sequences and runtime contexts. To identify the root causes mentioned above, we employed a machine learning algorithm to cluster the workloads and further evaluate their necessity. For the issues concealed by the specific contexts, we carefully set up several execution contexts to catch them. More importantly, we designed leading edge technology, e.g. pre-designing input sequences with potential energy overuse and tuning tests on-the-fly, to achieve high efficacy in detecting energy issues. A large-scale evaluation shows that 91.6% issues detected in our experiments were previously unknown to developers. On average, these issues double the energy costs of the apps. Our testing technique achieves a low number of false positives.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397350

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Mobile Applications, Energy Issues, Energy Bugs, Android

ACM Reference Format:
Xueliang Li, Yuming Yang, Yepang Liu, John P. Gallagher, and Kaishun Wu. 2020. Detecting and Diagnosing Energy Issues for Mobile Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397350

1 INTRODUCTION

Mobile devices have evolved into a wide ecosystem, providing millions of third-party apps to serve various user needs. Energy efficiency is a desirable quality attribute of mobile apps. However, many real-world mobile apps suffer from energy misuses. For example, Huang et al. [32] reported that most energy issues in mobile devices are caused by apps (47.9%) rather than systems (22.2%). We also randomly sampled 89 open-source Android apps and found that 27 (30.3%) of them suffer from serious software energy issues.

Despite many pieces of existing work (e.g., [2, 31, 32, 39]), the root causes and manifestations of energy issues in mobile apps are still not very well studied. Due to this reason, there exist few effective testing techniques to uncover serious energy issues. This motivates us to conduct an empirical study on 27 energy-inefficient Android apps to learn the real root causes of energy issues and their manifestation in practice. Specifically, we aim to answer two research questions:

- **RQ1 (Issue Causes):** *What are the common root causes of energy issues?*
- **RQ2 (Issue Manifestation):** *How do energy issues manifest themselves in practice?*

We analyzed 171 energy issues from the 27 open-source projects. For RQ1, we identified four main root causes of energy issues: 1) *unnecessary workload*, 2) *excessively frequent operations*, 3) *wasted*

Table 1: Comparison with the existing technology in terms of ability to deal with root causes and manifestations.

Root Cause		Existing work [2]	This work
Unnecessary Workload	(46.7%)	✗	✓
Excessively Frequent Operations	(17.8%)	✗	✓
Wasted Background Processing	(20.0%)	✓	✓
No Sleep	(32.2%)	✓	✓
Manifestation Type		**Existing work [2]**	**This work**
Simple Inputs	(9.2%)	✓	✓
Special Inputs	(68.4%)	✓	✓
Special Context	(25.0%)	✗	✓

background processing, and 3) *no-sleep* (**Finding 1**). For RQ2, we found that many energy issues require special inputs (68.4%) or special context (25.0%) to trigger and only 9.2% energy issues can manifest themselves with simple inputs (**Finding 2**).

We further studied the state-of-the-art testing technique for energy issues [2], which was proposed by Banerjee et al., and made two observations. First, as shown in Table 1, the existing technique is only capable of exposing issues resulting from wasted background processing and no sleep. This is because the technique diagnoses energy issues based on *E/U ratio*, i.e., the ratio of energy-consumption to hardware-utilization. If E/U ratio is high, it means that energy consumption is high, while hardware utilization is low, implying that the app under analysis is energy-inefficient. This point of view seems reasonable. For energy issues caused by wasted background processing and no-sleep, the E/U ratio could be remarkably high. However, according to Finding 1, many energy issues can also be caused by unnecessary workload and excessively frequent operations. These issues cannot be detected via analyzing E/U ratio: they may enlarge E and U simultaneously, hence E/U ratio is not a good indicator of the existence of such issues. Second, regarding Finding 2, the existing technique is capable of generating simple and special inputs to trigger energy issues, but does not simulate special contexts (e.g., poor network performance). Due to this limitation, it may miss many real energy issues (25.0% of our studied issues can only be triggered under special context).

Based on these observations, we propose a novel testing framework for effectively detecting energy issues. It works in two critical steps. First, our framework examines apps with a large variety of well-designed input sequences and runtime contexts, aiming to provoke energy issues to manifest themselves. Next, we need to recognise the appearance of energy issues. This is challenging due to the lack of effective test oracle. To address the challenge, our framework deals with different issue causes respectively. For wasted background processing, we compute statistical dissimilarity of the power traces before and after use of the app. If the dissimilarity exceeds a threshold, it reveals that the backgrounded (i.e. after-use) app is not as moderate as supposed to be, and is probably suffering from energy issues. We handle no-sleep in an analogue manner. To detect unnecessary workload and excessively frequent operations, we employ a machine learning algorithm to assess the *necessity* of the app's workloads according to several key criteria, such as lengths of continuous-high-power periods. If the workload is assessed *unnecessary* and *severely* energy consuming, then it will be identified as the occurrence of an energy issue.

We also design a package of practical techniques to enhance issue-detection efficacy of framework. For instance, before testing, our framework scans and analyses the source code of apps to extract the input sequences that are most likely to incur energy issues. During testing, the framework adjusts test targets on-the-fly to increase chances of exposing energy issues.

We performed experiments to evaluate our framework. The results show that our testing framework can uncover a large number of serious energy issues in high-quality apps, 91.6% of which have never been discovered before. On average, these issues double the energy cost of the apps. Manual verification also shows that our framework only reports a low number of false positives.

The key contributions of this paper are as followed:

- To the best of our knowledge, we conducted the largest-scale empirical study on developer-reported energy issues in mobile apps (*largest previous empirical study [30]: 8 app subjects, 10 energy issues; this paper: 27 app subjects, 171 energy issues*). Our findings can greatly benefit the research on energy issue detection and diagnosis.
- Inspired by the findings, we designed and implemented an automated testing framework for detecting energy issues. Our innovations include extracting battery-hungry input sequences from source code, steering the test direction on-the-fly for high detection efficacy, and employing machine learning to classify app workload.
- We empirically evaluated our framework and the results are promising: it detected 83 issues in 89 apps, creating many opportunities for optimizing the energy efficiency of these apps. As far as we know, this evaluation of a testing framework for energy issue detection is of the largest scale (*largest previous evaluation [2]: 30 app subjects and detected 12 issues; this paper: 89 app subjects and detected 83 issues*). Our subjects are also of higher quality than previous work, which are selected considering metrics such as high popularity and maintenance quality. In constrast, most subjects in previous work do not meet this standard.

In the remainder of this paper, we first introduce the data source for empirical study in Section 2, and discuss the findings in Section 3. We present our testing framework and technical details in Sections 4 and 5. Finally, we present an evaluation of our framework and discuss the results in Section 6.

2 DATA SOURCE

Open-source projects typically have publicly accessible issue tracking systems and code repositories. In the issue tracking systems, developers can post an issue report, which contains a title and a main body part, to report the symptoms of their observed bug/issue [1] and the steps to reproduce the issue (optional). Following that, developers can discuss the issue and comment on the report. Those developers who are assigned to fix the issue can propose potential code revisions. Typically, after code review by other project members and further changes, such revisions will be committed to the project's code repository.

[1]We may use the terms of bugs and issues interchangeably in this paper.

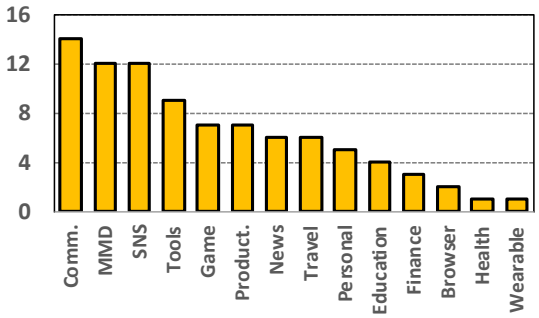

Figure 1: The 89 app subjects of different categories.

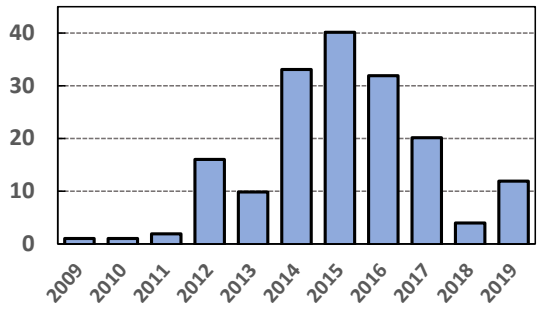

Figure 2: The yearly number of reported energy issues.

Our empirical study is conducted on well-maintained Android application projects from three popular open-source software hosting platforms: GitHub[2], Mozilla[3], and Chromium[4] repositories. The criteria for selecting app subjects for our study are these: 1) a subject should have achieved at least 1,000 downloads on the market (popularity), 2) it should have more than one hundred code revisions (maintainability). Following these criteria, we randomly selected **89** app subjects from those three software hosting platforms. These open-source applications are also indexed by the F-Droid database[5]. Figure 1 presents the numbers of app subjects of different categories. They cover most application categories in F-Droid and in total contain 108,193 issue reports, indicating that these subjects are quite large-scale.

To search for energy issues, we employed keyword searching in issue reports' title and body to locate potential energy issues. The keywords we used are *energy*, *power* and *battery*. While keyword searching generally helps to retrieve most energy-related issue reports, it can also produce false positive results when the issue reports accidentally contain any of our keywords. To filter out such irrelevant issue reports, we manually verified each returned issue report to make sure the issue concerned is indeed an energy issue. In total, we checked 976 retrieved issue reports and this helped us locate 171 real energy issue reports from **27** apps. Figure 2 shows the distribution of opening dates of these reports. There was a peak reporting period from 2014 to 2017. After, the number dropped in 2018 and 2019. Only 28.1% (48 out of 171) reported issues were fixed. The main reasons for no fixes are these: 1) many energy issues are irreproducible (60.8%), 2) the problematic code cannot be localized (9.3%), 3) energy-saving by fixing the issues cannot be evaluated (9.3%), and 4) the fix causes other issues (9.3%). So in this paper, we will present a cutting-edge technology for effectively pinpointing energy issues in the lab where testing condition is precisely controlled, so the issues can be strictly reproduced and localized. Also, energy-saving can be accurately evaluated.

[2]https://github.com
[3]https://dxr.mozilla.org
[4]https://www.chromium.org
[5]https://f-droid.org/

3 EMPIRICAL STUDY

To answer our research questions, we carefully studied the 171 energy issue reports. The results are as follows.

3.1 RQ1: What Are the Common Root Causes of Energy Issues?

Among the 171 reports, 90 explicitly show the information on root causes of the issues. We examined all of them and observed the following six root causes. Some issues were caused by multiple reasons, hence, the sum of the percentages below is over 100%.

Unnecessary workload (42/90=46.7%). Many applications perform certain computations that do not deliver perceptible benefits to users. These computations incur unnecessary workload on hardware components including CPU, GPU, GPS, network interface, and screen display. For example, in the reports of Chrome issue 662012 and 541612[6], the application produces frames constantly even when visually nothing is changed or repainted, which causes huge workload on CPU and GPU. And the report of OpenGPSTracker issue 406 shows that the app keeps recording users' location even after not moving for minutes, barely exhausting battery.

Excessively frequent operations (16/90=17.8%). Performing certain operations too frequently can also waste power. In comparison with unnecessary workload, when fixing energy inefficiencies caused by excessively frequent operations, the developers do not completely remove the operations (because their functionality is necessary), but reduce the frequency of operations. For example, in Firefox (issue 979121), whenever users type in the URL bar and the text changes, the app will query the database (e.g. for autocompletion). Considering users often visit the websites they visited before, developers suggested to store users' browsing history in memory to reduce database hits to save energy.

Wasted background processing (18/90=20.0%). As battery powered mobile devices are extremely sensitive to energy dissipation, it is good practice to make backgrounded applications as quiet as possible. Specifically, the "backgound" here means that after the use of an application or "activity" (a major type of application component that represents a single screen with a user interface[7]), users press the *Home* button or switch to another application or activity, so the previous application or activity goes to background. Typical examples are Chrome issue 781686 and Firefox issue 1022569: when users select a new tab (each tab is an activity), the invisible old tab would still keep being reloaded, which wastes battery. But these issues do not have an easy solution since users may want old tabs to be reloaded.

[6]"Chrome" is the app name, "662012" and"541612" are the issues' ID given by the corresponding issue tracking system.
[7]https://developer.android.com/guide/components/fundamentals.html

No-sleep (29/90=32.2%). The no-sleep issue means that when the screen is off and device is supposed to enter sleep mode, certain apps still keep the device awake, which usually results from misuse of asynchronous mechanisms [35] like services, broadcast receivers, alarms and wake-locks. For example, Kontalk issue 143 unnecessarily holds a wake-lock, preventing the device from falling asleep. For another example, in Firefox (issue 1026669), the Simple Service Discovery Protocol (SSDP) activates the searching service every two minutes when the screen is off, which not only incurs a large amount of workload but also prevents the device from entering the sleep mode. Program 1 gives the JavaScript patch for fixing this issue. It added the cases to deal with "application-background" and "application-foreground" for the SSDP service. Note that, the "application-background" defined by developers includes both screen-off time and the scenarios where users switch to another application. So this issue belongs to two categories: *no-sleep* and *wasted background processing*.

```
case ''ssdp-service-found'':
- {
-  this.serviceAdded(SimpleServiceDiscovery.findServiceForID(aData));
-  break;
- }
+ this.serviceAdded(SimpleServiceDiscovery.findServiceForID(aData));
+ break;
case ''ssdp-service-lost'':
- {
-   this.serviceLost(SimpleServiceDiscovery.findServiceForID(aData));
-   break;
- }
+ this.serviceLost(SimpleServiceDiscovery.findServiceForID(aData));
+ break;
+ case ''application-background'':
+ // Turn off polling while in the background
+ this._interval = SimpleServiceDiscovery.search(0);
+ SimpleServiceDiscovery.stopSearch();
+ break;
+ case ''application-foreground'':
+ // Turn polling on when app comes back to foreground
+ SimpleServiceDiscovery.search(this._interval);
+ break;
```

Program 1: JavaScript patch of Firefox issue 1026669.

Spike workload (2/90=2.2%). A spike workload can cause lagging UI [30], degrade user experience and heat up the device, inducing a huge energy waste. For instance, in RocketChat (issue 3321), when users send or receive GIF animation pictures, CPU utilization quickly rises to 100% and heavily affects the battery.

Runtime exception (3/90=3.3%). In some cases, runtime exceptions may provoke abnormal behaviors of a mobile application and cause energy waste. For instance, in AntennaPod (issue 1796), the "NullPointerException" makes the download process persist and consume power. In our study, such energy issues caused by runtime exceptions are not common and we only observed three cases.

Considering spike workload and runtime exception are of small proportions in practice, our issue-detection technology only focuses on the main root causes apart from them.

3.2 RQ2: How Do Energy Issues Manifest Themselves in Practice?

Out of the 171 reports, 76 contain explicit information that shows how the issues manifest themselves. We studied these 76 issues to answer RQ2. We observed three manifestation types: *simple inputs*, *special inputs* and *special context*. It is worthwhile to notice that, *simple inputs* and *special context* may combine to incur issues, on the other hand, *special inputs* and *special context* may also overlap.

Simple inputs (7/76=9.2%). Simple inputs mean one tap or swipe gesture in common interaction scenarios. We found seven issues are of this type of manifestation. For example, Andlytics issue 543 lets the app refresh itself whenever the user opens the app. And VoiceAudioBookPlayer issue 299 makes the app unnecessarily scan folders every time the user starts or leaves the app.

Special inputs (52/76=68.4%). The majority of the energy issues can only be triggered with certain specific inputs or a sequence of user interactions (e.g. text typing, taps, or swipes) under certain states of an application. For instance, the c:geo (a geocaching app) issue 4704 requires three steps to reproduce: 1) open the app and make sure GPS is inactive since the app starts, 2) change between cache details and other tabs of the same geocache, so GPS is activated, 3) put the device in standby and let timeout to screen-off. After a while, users would find GPS stays active even when the screen is turned off. To avoid energy waste, users decide to quit using the app.

Special Context (19/76=25.0%). Special context includes environmental conditions (rather than user interactions, e.g. taps) such as the accessibility of networks, location of the device, settings of the OS and applications. In our dataset, 19 issues require such special contexts to trigger. For instance, MPDroid issue 3 appears when the user is watching stream videos but the network is disconnected; the app then keeps trying to load the video and consumes battery. AnkiDroid issue 2768 occurs when users lock the phone screen when the application is in "review" mode and a notification comes in afterwards, so the screen will hold on until the battery is dead.

4 OVERVIEW OF TESTING FRAMEWORK

The following observations motivated us to design a novel testing framework for effectively detecting energy issues in real apps:

- From **Finding 1**, we address previously unaddressed energy issues caused by unnecessary workload and excessively frequent operations.
- From **Finding 2**, we found that 25.0% of energy issues can only be manifested under special context such as poor network performance. However, such factors were neglected.

According to the first observation, as we discussed in Section 1, *evaluating the necessity of app workload is crucial for identifying these issues*. We will use machine learning to cluster workloads, and further assess their necessity, as shown later in Section 5.3. According to the second observation, we will devise two types of most common special contexts for effectively revealing these issues, as shown in Section 5.1.2. Developers also can duplicate our experiment for detecting these hidden issues. Importantly, we also make practical designs and implementations to enhance the efficacy of our testing framework.

Figure 3 shows the framework overview. The framework first makes sophisticated preparation before testing. For example, it inspects the source code and collects the candidate input-sequences that are most suspected of energy overuse. A set of candidate runtime contexts (containing the above mentioned two types of special

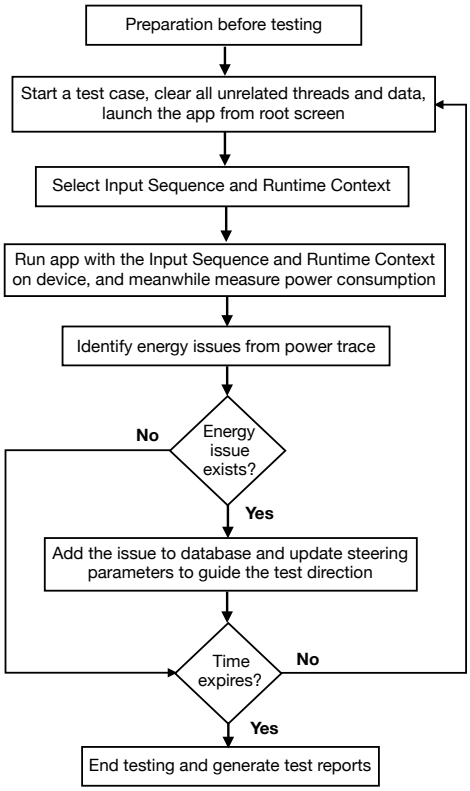

Figure 3: The flow chart for our testing framework.

contexts) are also carefully designed to increase the chance of provoking energy issues. Later, these candidate inputs and contexts will be explored under an effective and systematic scheme.

To start a test case, the framework at first clears unrelated threads and data to minimize the interference from other applications and previous test cases. Then, it will select one input sequence and one runtime context from the candidates, then run the app with them. During the entire test, the power consumption of device is traced with a power monitor. Our framework will look into the power trace and decide whether an energy issue exists. If one does exist, the issue information will be added into database. The entire test is limited with a time budget. If the time budget runs out, the test will quit and test reports will be generated for developers to help fix the issues. Otherwise, our framework will start a new test case.

As mentioned above, our framework explores the inputs and contexts under a systematic scheme, where the exploring direction is tuned on-the-fly. The rationale of our scheme is that if an energy issue occurs, it implies that the type of the input sequence and runtime context incurring this issue may be more likely to uncover energy issues than average case, since it did cause an energy issue to show up; we thus increase the chance of this type of inputs and context to be tested. Concretely, we utilize a set of parameters and iteratively update them to guide the test direction, as shown later in Section 5.2.

The large-scale evaluation (Section 6) shows that, exploiting these practical and targeted tests, our framework largely outperforms the state of the art on the efficacy in detecting all kinds of energy issues.

5 DETAILED TECHNOLOGY

This section introduces detailed implementations of our testing framework. It involves how to design candidate input sequences and runtime contexts, how to steer the test direction at runtime, and how to identify energy issues from the power traces, etc. The overall objective is to effectively and accurately pinpoint energy issues.

5.1 Preparation before Testing

As shown in Section 4, our test case is driven by the input sequence and runtime context. Our candidate input sequences are designed for high utilization of the main hardware components, such as CPU, screen display and network interface, since they are usually the culprits of energy waste, as shown in the literature [46]. Meanwhile, according to Finding 2, we will carefully devise a set of artificial runtime contexts for effectively detecting those energy issues.

5.1.1 Design of Candidate Input Sequences. We design two types of candidate input sequences. One is **weighted input sequences**, the other is **random input sequences**. The former helps our framework detect issues in a guided manner, the latter will cover some corner cases that are hard to predict.

Weighted input sequences are generated referring to the Event-Flow Graph (EFG) [36]. Each node in an EFG is a User Interface (UI) component, such as a button or a list item. If a user interaction on a UI component, say $node_1$, can immediately bring out another UI component, say $node_2$, then the EFG has a directed edge from $node_1$ to $node_2$. Technically, we utilize Layout Inspector[8] to construct the EFG of an app. *An arbitrary path in EFG could be a candidate input sequence for our testing framework.* In practice, our test cases always start from the root node (i.e. the initial UI component of the app). The lengths of paths are constrained with a limit. Note that, even though Layout Inspector can construct the EFG, it does not have a capacity to run apps with the paths. So in our test, we use Dynodroid[9] to feed the paths to apps.

Importantly, all input sequences generated from EFG are assigned a weight. The weight indicates potential of a sequence to cause energy waste; an input sequence with a larger weight has a higher priority to be tested. We use Equation (1) to calculate the weight for each input sequence. S is the number of a certain set of system APIs invoked by the input sequence. C is the number of function invocations and block transitions incurred by the input sequence. α and β are used for adjusting influences of S and C on the weight; $\alpha > 0, \beta > 0, \alpha + \beta = 1$. In practice, we set α at 0.6 and β at 0.4.

$$weight = \alpha * S + \beta * C \quad (1)$$

The reason why we resort to S and C to indicate the potential of excessive energy use is this: as shown in [46], the main energy-consuming components are CPU, screen display, network interface (cellular and WiFi), as well as GPS and various sensors. Except for CPU, all other components can only be controlled by a set of system APIs, as listed in [2]. The more this set of system APIs an input sequence accesses, the larger are the chances it causes energy waste. The CPU executes basic operations that constitute the source

[8] https://developer.android.com/studio/debug/layout-inspector
[9] https://dynodroid.github.io

code of apps, for example, arithmetic operations like additions and multiplications, and control-flow operations mainly including function invocations and block transitions. Recent research [22, 26] has shown that control-flow operations are the actual main energy-consumers for Android app source code. We therefore use the total number of the main control-flow operations (i.e. function invocations and block transitions) to indicate the potential CPU overload of an input sequence.

Note again that, we calculate S and C before testing. We first instrument the app source code, and run the app with the input sequences in the emulator on a powerful PC, and then record their S and C values individually.

Apart from **weighted input sequences**, we also designed **random input sequences** to cover exceptional cases we might not envisage. We use the Monkey tool[10] to generate random input sequences, such as taps and swipes. "Random" here means the position of the inputs on screen are randomly set. Monkey does not generate input sequences at runtime. Instead, Monkey pre-defined a large number of random input sequences. When being asked for an input sequence, Monkey will arbitrarily deliver one of them with its ID. The advantage of this design is that testers can conveniently use the same ID to repeat the input sequence and reproduce the test case. In our test, the pre-defined input sequences in Monkey are all adopted as our candidate input sequences.

In summary, our test combines two types of input sequences, namely, **weighted** and **random**. In Section 5.2, we will show the strategy for balancing the testing time on them.

5.1.2 Design of Candidate Runtime Contexts. The entire experiment is set in a signal shielding room, enabling us to manipulate the contextual factors, such as the strength of WiFi (in our test, we employ WiFi as the connection to Internet) and GPS signal. We designed three types of runtime contexts, namely, `Normal`, `Network Fail` and `Flight Mode`. In `Normal`, the WiFi and GPS work normally (package delivery delay is 36 ms and bandwidth is 3.2 Mb/s). In `Network Fail`, the signal is seriously weak (package or message delivery delay lengthens to 451 ms, bandwidth drops to 12.0 Kb/s). In `Flight Mode`, WiFi is closed at software level by the OS, and GPS works normally.

The **reason** for choosing `Network Fail` and `Flight Mode` as representatives for special contexts is that our empirical study shows they are the two major types of special contexts. The former accounts for 26.3% (5 out of 19), the latter for 15.8% (3 out of 19) of issues manifested under special context.

We also designed a special type of runtime context, `Non-background`. We designed this context because in our experiment we observed that it can provoke more no-sleep issues, as shown later in Section 6.3. In `Non-background`, the network and GPS work ordinarily, however, we do not input a press of *Home* button to the device after `EXECUTION` stage (the stage-division for test cases will soon be explained in Section 5.3). That is, the test case does not have `BACKGROUND` stage, and straight goes to `SCREEN-OFF`.

[10] https://developer.android.com/studio/test/monkey.html.

5.2 Steer the Test Direction On-the-fly

Our framework steers test direction dynamically based on test history. Algorithm 1 shows details of our steering scheme. The rationale behind it is this: when an energy issue is detected, it implies that this type of input sequence and runtime context may have a greater opportunity than usual to provoke energy issues since it did trigger an energy issue. Hence, our framework will generate slightly more of this type of test cases for larger chance of confronting energy issues.

Algorithm 1: Steer the test direction on-the-fly

Data:
$WeightedInputSequences = \{(sequence_i, weight_i)\}$;
$RandomInputSequences = \{sequence_j\}$;
$RuntimeContexts[N] = \{context_k\}$;
$0 < p_{wgt} < 1$, $P_{ctx}[N] = \{0 < p_k < 1\}$;
$\Delta_{wgt}, \Delta_{cxt}$;

1 Preparation before testing;
2 Start a test case, clear unrelated threads and data, launch app from root screen;
3 #---------Select Input Sequence and Runtime Context----------#
4 Determine the type of input sequence, and the type of "weighted" has a probability of p_{wgt} to be chosen;
5 **if** *the determined type is "weighted"* **then**
6 Select an unexplored sequence with highest $weight$ in $WeightedInputSequences$;
7 **else**
8 Randomly select one sequence from $RandomInputSequences$;
9 **end**
10 Select one context from $RuntimeContexts$ with its corresponding probability;
11 #----------#
12 Run app with the selected input sequence and runtime context on device, and meanwhile measure power consumption;
13 Identify energy issues from power trace;
14 **if** *there exists an energy issue* **then**
15 Add the energy issue to database;
16 #----------Update steering parameters----------#
17 **if** *the issue is incurred by a "weighted" sequence* **then**
18 **if** $p_{wgt} \leq wgt\_up\_threshold - \Delta_{wgt}$ **then** $p_{wgt} := p_{wgt} + \Delta_{wgt}$;
19 **else**
20 **if** $p_{wgt} \geq wgt\_down\_threshold + \Delta_{wgt}$ **then** $p_{wgt} := p_{wgt} - \Delta_{wgt}$;
21 **end**
22 **switch** *which context triggers the energy issue* **do**
23 **case** *e.g.* $context_k$ **do**
24 **if** $p_k \leq cxt\_up\_threshold - \Delta_{cxt}$ ***and*** *there are n elements (except p_k) in P_{ctx} that are greater than or equal to* $cxt\_down\_threshold + \Delta_{cxt}$ ***and*** $n > 0$ **then**
25 $p_k := p_k + \Delta_{cxt}$;
26 Decrease those n elements individually by Δ_{cxt}/n;
27 **end**
28 **end**
29 **end**
30 #----------#
31 **end**
32 **if** *time expires* **then End Testing** and generate test reports;
33 Go back to **line 2**

Data for the algorithm. The candidate input sequences and runtime contexts are designed based on the approach we demonstrated in Section 5.1.1 and 5.1.2. We present them in the data structures of $WeightedInputSequences$, $Random\ InputSequences$ and $Runtime\ Contexts$. N is the number of candidate runtime contexts. In our test, we devised 4 runtime contexts (including `Non-background`), so $N = 4$. p_{wgt} and P_{ctx} are the steering parameters for concretely

guiding the test. p_{wgt} is the probability of choosing a **weighted** input sequence for the upcoming test case. In practice, we initialize it as 50%, so the first test case has a chance of 50% to run with a weighted input sequence, and we will update and refine p_{wgt} dynamically during the entire testing. On the other hand, one element p_k in P_{ctx} represents the probability of choosing $context_k$ in $RuntimeContexts$. We will also update P_{ctx} at runtime.The summation of elements in P_{ctx} is bound to 1.

Δ_{wgt} is the increment used to increase or decrease p_{wgt} to renew p_{wgt}. Δ_{cxt} plays the same role for P_{ctx}. The larger Δ_{wgt} and Δ_{ctx} are, the more aggressively we tune the test direction.

Details of the algorithm. We first prepare data and initialize parameters (p_{wgt}, P_{ctx}, Δ_{wgt} and Δ_{cxt}). We then start a test case, clear unrelated threads and data, and launch the app from root screen. Next, we decide the type of input sequence; weighted input sequences have a probability of p_{wgt} to be chosen. If the chosen type is "weighted", we then select an unexplored sequence with the highest *weight* in $WeightedInputSequences$. Otherwise, we randomly select a sequence from $RandomInputSequences$. Likewise, for runtime context, we select one from RuntimeContexts with its corresponding probability.

We then feed the app with the selected input sequence and runtime context, and measure the device power. The power trace will be analysed to confirm whether an energy issue occurs. If there exists an energy issue, it implies that this type of input sequence and runtime context may be profitable for provoking more energy issues, our testing framework then steers lightly in this direction. Specifically, if it is triggered with a weighted input sequence, we increase p_{wgt} by Δ_{wgt}. And after being increased, p_{wgt} should not exceed $wgt\_up\ \_threshold$. If it is a random input sequence, we decrease p_{wgt} by Δ_{wgt}. Also, we keep $p_{wgt} \geq wgt\_down\ \_threshold$.

An analogous approach is applied to refining P_{ctx}. We check under which runtime context (e.g. $context_k$) the issue occurs, then increase its testing probability (e.g. p_k). However, the precondition is that there should be at least one element (except p_k itself) in P_{ctx} that are no less than $cxt\_down\ \_threshold + \Delta_{cxt}$, because on one hand, we intend to rebalance the probabilities, and on the other, we should let all contexts have at least a possibility of $cxt\_down\_threshold$ to be tested.

5.3 Identify Energy Issues from Power Trace

We divide the power trace into five stages, i.e. `PRE-OFF`, `IDLE`, `EXECUTION`, `BACKGROUND` and `SCREEN-OFF`. This division can help us identify three types of energy issues: execution issues (including issues caused by unnecessary workload and excessively frequent operations), background issues (i.e. wasted background processing) and no-sleep issues. Figure 4 shows an illustration of power traces with these three types of energy issues.

`PRE-OFF` stage is the beginning stage where the device is powered but the screen is off. Then, the test case will be transferred to `IDLE` stage by turning on the screen. To enter `EXECUTION` stage, the subject application will be opened and run with a certain input sequence and runtime context, which are selected as shown in Section 5.2. After `EXECUTION` stage, the application will be fed with a press of *Home* button to enter `BACKGROUND` stage. The final stage is `SCREEN-OFF` stage, which begins when screen is supposed to be

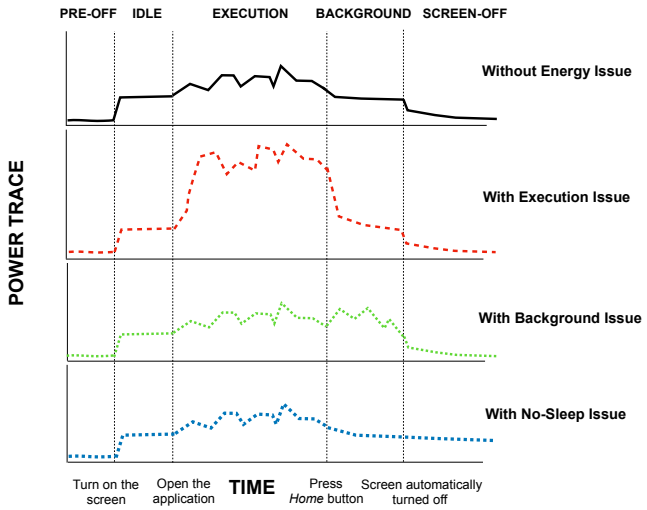

Figure 4: An illustration of power traces with energy issues.

turned off automatically, however, part of energy issues keep the screen on even at `SCREEN-OFF` stage, eating battery power badly.

5.3.1 Identifying Execution Issues. For execution issues, as we discussed in Section 1, evaluating the necessity of app workload is crucial for identifying them. Specifically, we employ the Dbscan clustering algorithm [8] (Density based spatial clustering of applications with noise) to fulfil this purpose. The objective of Dbscan is to classify multidimensional data points into three groups, namely, core points, border points and outlier points. After clustering, the data points should have the following properties: 1) For a core point, the number of its neighbours (the points within a range of ε from it) is no less than a certain value, $MinPts$. Generally speaking, the core points are "quite close and gathered". 2) For a border point, its neighbours are less than $MinPts$, but it is a neighbour of at least one core point or another border point. 3) For an outlier point, its neighbours are less than $MinPts$, and it does not have either a core or a border neighbour.

We treat each test case as a data point, and treat test cases in the same app category as a data set for clustering. The dimensions of each data point we employed for clustering are l_{chpp}, n_{chpp}, μ_{chpp}, μ_{exe}, which are all extracted from power trace of `EXECUTION` stage of each test case. l_{chpp} is the total length of continuous-high-power periods. Continuous-high-power period is when power continuously exceeds a certain threshold longer than a certain length. n_{chpp} is number of these periods. μ_{chpp} is average power of these periods. μ_{exe} is average power of the entire `EXECUTION` stage. Dbscan then classifies the test cases into those three groups. We label test cases in core and border groups as "normal", and the ones in outlier group as suspects for suffering from execution issues.

The motivation of this approach to probing for execution issues is this: usually "normal" energy use is sufficient to guarantee quality of user experience (QoE) of apps, outlier-level energy use is suspiciously problematic and unnecessary. And different app categories usually have distinct "normal" energy use (e.g. games *vs.* productivity apps). So we handle different app categories separately as shown above.

Table 2: Technique package. The bold items are our originally-designed techniques, the italic are inspired by [2].

Preparation before testing	Steer test direction
• *Candidate input sequences* ◦ *Weighted input sequences* - *System APIs (I/O components)* - **Control-flow operations (CPU)** ◦ **Random input sequences** • **Candidate runtime contexts** ◦ *Normal* ◦ **Non-background** ◦ **Flight Mode** ◦ **Network fail**	• **Balance weighted and random input sequences** • **Balance different runtime contexts** • **Improve efficacy of issue-detection**
Identify energy issues from power trace	
• **Identify execution issues** ◦ **Select dimensions** ◦ **Cluster** ◦ **Label outliers**	• *Identify background and no-sleep issues* ◦ *Dissimilarity analysis*

Note that, the ultimate clustering model is available only when all test traces are collected. So at the beginning of testing, such a model is inaccessible since we have no power traces to build it. So, we need to decide, from which point during testing, we should start building a model using the collected data. To make our design reasonable, we did a pilot study by running 100 randomly-chosen test cases. We extracted the key features from the power traces and visualized the features. We found two "obvious" outliers using Dbscan. With further manual verification, both outliers were confirmed to be real issues.

As a density-based clustering algorithm, Dbscan may generate biased-models when outliers are of significant proportion in the dataset. In our problem domain, the number of outliers (abnormally high energy use) is naturally low, as shown in our pilot study and in our final evaluation (Section 6). Hence, in our experiment, we at first collected 50 power traces to bootstrap the model building process and then iteratively added new power traces to the dataset. As the dataset grows, the model becomes more accurate and powerful for identifying outliers. At last, we used the final model to recheck all power traces for issue detection.

Later, our evaluation on 89 apps (involving 35600 test cases) shows that only 1.1% test cases are outliers. Our framework detected 47 candidate execution issues from those 1.1% test cases. Only three (out of the 47) are false positives, indicating the high reliability of this approach. In contrast, current technology [2] detected 3 candidate execution issues from 30 apps, and still one of them is a false positive.

5.3.2 Identifying Background and No-Sleep Issues. If the app is free from background issues, the power trace in `BACKGROUND` stage is supposed to be similar to that in `IDLE` stage. We thus compute the dissimilarity value of the two traces. If the value is above a certain threshold (40% in our experiment), we label this test case as a candidate for a background issue.

We identify the no-sleep issues in the same way. We compare `PRE-OFF` with `SCREEN-OFF`. If the dissimilarity outstrips a certain threshold (50% in our experiment), we speculate this test case is suffering from a candidate no-sleep issue.

5.4 Manual Verification

After the candidate issues are found, we manually verify whether they are actual energy issues. We adopt three criteria for distinguishing a real issue from a false positive. First, the energy cost is not from the OS. Second, the energy cost is larger than normal cases by at least 10%. Third, user experience can be improved after removing the workloads. For the third criterion, which may be subjective, we clarify it using an example: an app is downloading large files for functional purposes, such as maps in games, which causes abnormally high energy cost. In this case, there is no easy solutions to save battery power since users may accept the expensive downloading process (removing it will affect app functionality). We then identify it as a false positive. However, for cases such as the issue in Leisure (as shown later in Table 3), where GIF animations are played when they are invisible, we identify it as a real issue since animations in such cases do not generate user-observable benefits. One can optimize the apps by removing such expensive computation. However, we agree that evaluating necessity of workload is an obvious challenge. There is no very effective and general solution yet.

5.5 Comparison with the Most Relevant Work

Table 2 lists the differences between our technology and the most relevant work [2]: 1) For preparation before testing, since their work [2] only takes I/O components into account, their technology is impracticable for the important CPU-bound apps (e.g. games) and CPU-related energy issues. They also neglected the special runtime contexts which can trigger 25.0% of energy issues. 2) W.r.t steering test direction, their work does not have this feature because they only consider two dimensions of testing space, i.e. I/O-related input sequences and `Normal` context, whereas our framework additionally tests five more dimensions including CPU-related and random input sequences, and three more runtime contexts. So our searching space for energy issues is exponentially enlarged; we thus designed practical online steering strategy to balance different kinds of inputs and contexts, and improve the issue-detection efficacy. 3) For identifying energy issues from power trace, as we mentioned in Section 1, E/U theory [2] can hardly address execution issues (64.5% of all energy issues), by comparison, our framework employs advanced machine learning algorithm to analyse the energy use of apps and filter out these issues.

Benefiting from these targeting designs, our framework largely outperforms the state-of-art, as shown in Section 6.

6 EXPERIMENTAL EVALUATION

In this section, we first introduce our experimental setup. Then we evaluate our testing framework on various aspects, such as its efficacy in detecting energy issues, its comparison with the state-of-the-art, etc. The result shows that our testing framework largely outperforms current technology, showing the benefits both of our sound empirical study and our dynamic targeting techniques.

6.1 Experimental Setup

We employ the Odroid-XU4 development board[11], whose processor has four big cores with a frequency of 2 GHz and four small cores with a frequency of 1.3 GHz. The board possesses a powerful 3D accelerator, Mali-T628 MP6 GPU. The high capacity of Odroid-XU4

[11] https://wiki.odroid.com/odroid-xu4/odroid-xu4

Table 3: Examples of detected energy issues.

Application	Category	Issue Type	Activity or Symptom	Hit Rate[1]	Context or Non-background	Energy Waste	Reported before?
Leisure	News	Execution	3 GIFs playing on one page	1.0%	Normal	25.9%	No
GPS Status	Travel	Execution	Not obvious	47.0%	Normal	15.3%	No
BatteryDog	Tools	Execution	Editing lengthy text	1.0%	Normal	30.8%	No
Rocket Chat	Comm.	Execution	Connect to server	1.0%	Normal	12.8%	No
Chess Clock	Tools	Background	Device heated up	100.0%	WiFi Fail	59.2%	No
Vanilla	Multimedia	No Sleep	Enqueue many tracks	59.0%	WiFi Fail	242.8%	No
cgeo	Travel	No Sleep	App get stuck	2.0%	Flight Mode	179.4%	Yes
AntennaPod	Multimedia	No Sleep	Keep popping up messages	1.0%	Non-background	184.4%	Yes

1. Hit rate here is the percentage of test cases detected having the energy issue in that app.

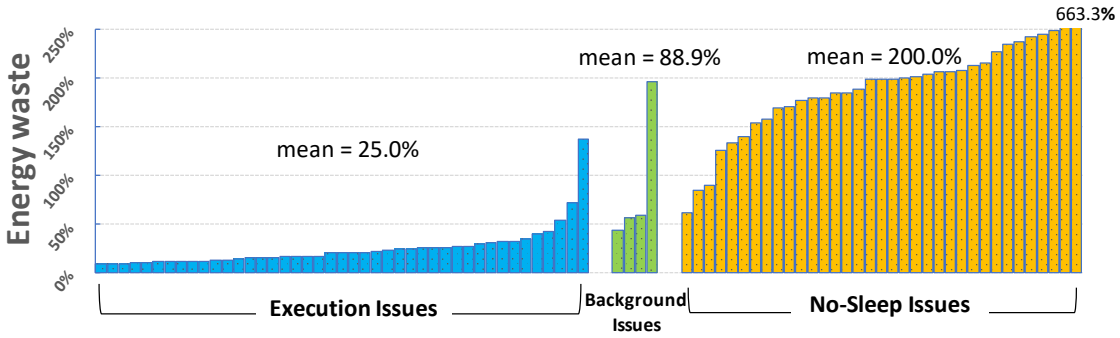

Figure 5: The energy waste of detected issues of different types

board guarantees its performance for most applications on the market. It is also equipped with a power monitor, Smartpower2[12], to measure the real-time power consumption. The sampling rate is 100 Hz. To assess its measurement variability, we randomly chose 20 test cases and ran each for 10 times. We thus collected 10 records of average power consumption for each test case. We employ coefficient of variation (C_v) to indicate the variability. Specifically, $C_v = \frac{\sigma}{\mu}$, where σ and μ are the standard deviation and mean of the 10 records for each test case. At last, the mean of the 20 C_vs is about 0.8%, meaning a low measurement variability. Due to these rich and solid features, the Odroid board is widely employed in the field of energy optimization for mobile devices [41, 52, 57].

We use Android 4.4 KitKat as our target OS. Android is open-sourced and it captures around 74.13%[13] of the worldwide mobile OS market by Dec 2019. We evaluate our framework on the 89 app subjects. 27 of them have issue reports, 62 do no, as we introduced in Section 2. The **considerations** of employing these 89 apps for the evaluation are these: 1) Our framework is inspired by issue reports from the 27 apps, so it may be inapplicable to new apps, we thus employ the 62 subjects without issue reports as **new** apps to evaluate the generality of our framework. 2) Referring to the issue reports, we also can check whether our framework is capable of detecting unreported issues. 3) These apps are popular, well-maintained and of much higher quality than the apps adopted by previous research.

Our total testing time for the 89 app subjects is 2373.3 hours, i.e. 98.9 days, which were evenly spent on each app. (i.e. 1.11 days for one app). Note that, the time of preparing weighted input sequences is also included, which takes 18.8% of the entire testing time.

6.2 The Efficacy of Our Testing Framework

The experimental result shows that our test detected 91 candidate energy issues, among which we manually confirmed **83 real energy issues**. 22.9% (19 out of 83) of these issues are from the 27 old apps, 77.1% (64 out of 83) are from the 62 new apps. After manual verification, we found **8 false positives**. Table 3 shows 8 examples of the detected energy issues. For instance, in `Leisure`, three animated GIFs are loaded and are played at the bottom of a certain page even though they are invisible to users most of the time. This execution issue wastes 25.9% energy use. It can be fixed by freezing the animation when the GIF pictures are not shown on the screen. For another example, when `Chess Clock` is not in use and backgrounded, the device will heat up from 41.2°C to 60.9°C due to the inefficient and long utilization of CPU. The average power of this issue is 59.2% higher than that of the `IDLE` stage.

Figure 5 shows the energy waste of detected energy issues. Energy waste is calculated using Equation (2).

$$w = (\frac{e_x}{e_n} - 1) \times 100\% \quad (2)$$

w is the energy waste of the issue, e_x is average power of the corresponding stage in the test case with the corresponding issue (`EXECUTION` stage for execution issues, `BACKGROUND` stage for background issues, `SCREEN-OFF` stage for no-sleep issues). e_n is "normal" energy cost. We define "normal" energy cost individually for different issues. For background and no-sleep issues, we adopt average power at `IDLE` and `PRE-OFF` stage as normal cost, respectively. For execution issues, we use mean value of average

[12] https://www.odroid.co.uk/odroid-smart-power-2
[13] http://gs.statcounter.com/os-market-share/mobile/worldwide

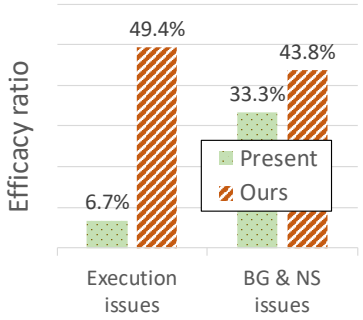

Figure 6: Comparison on issue-detection efficacy (r_s^d) with the present technology.

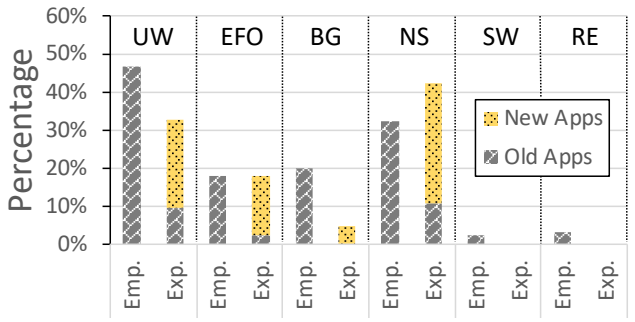

Figure 7: Issue causes in *empirical study* and *experiment*.

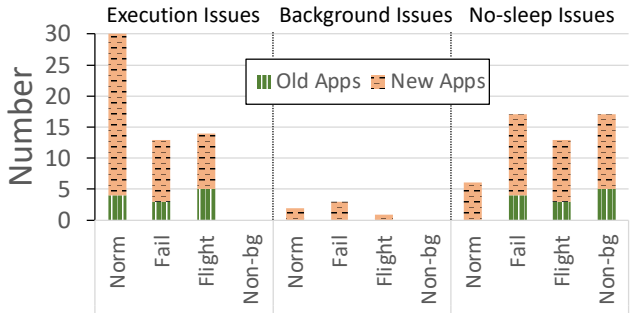

Figure 8: Energy issues manifested under different contexts.

powers of `EXECUTION` stage in test cases in the same app category, as normal cost.

The experimental result shows that the energy waste of execution issues is 25.0% on average and up to 137.6%; the energy waste of background issues is 88.9% on average and at maximum 196.2%; the values for no-sleep issues are 200.0% and 663.3%, respectively. Overall, the average energy waste of all the issues is 101.7%.

91.6% (76 out of 83) detected energy issues in our test are **newly-reported**, which on average double energy consumption of the apps. Without our tests, these serious energy issues might have never been detected even though they cause serious battery drain.

On the other hand, 95.9% (164 out of 171) energy issues in our empirical study are not listed in the issues detected by our test. The major reasons are as follows: Firstly, our standard of determining energy issues is much higher than that of developers. The issues detected in our test have outlier-level impacts on energy use; the energy wastage is usually above 10.0%. However, many issues detected by developers only cause small transient workloads, energy overuse can hardly reach 10.0%. For instance, Firefox issue 1057247 lets app re-fetch the failed favicon every 20 min, which is believed costly. So developers reduce the re-fetching frequency to conserve energy. For another example, VoiceAudioBookPlayer issue 299 makes the app scan material files everytime the user starts and leaves app. Developers then designed a smarter scanner to lessen the number of scans. However, these issues can not be noticed by our framework because they have very marginal influence on the metrics (i.e. l_{chpp}, n_{chpp}, μ_{chpp} and μ_{exe}) we chose to identify execution issues. Secondly, a number (35.4%, 58 out of 164) of the issues are not reproducible, so our test cannot trigger them either. Thirdly, the variety of input sequences and runtime contexts in our test is not large enough to cover all of them due to the time limit.

The first and third reasons also shed light on **false negatives** of our test. The first reason implies that the issues that draw developers' attention but consume non-significant power (compared with our detected issues) can be missed by our test. It is because only outlier-level energy cost is deemed as an energy issue in our test. Loosening this strict criterion can help significantly reduce false negatives but may lead to false positives. The dilemma is caused by the difficulty of objectively and automatically judging the necessity of workloads. There is no general solution for this problem yet. Nevertheless, our framework can be flexibly configured to detect more energy issues if users are willing to tolerate some false positives. W.r.t. the third reason, since we were testing 89 apps, one day's test for one app results in three months' test for all apps. Developers can test their own app more comprehensively to realize larger coverage.

We now compare the efficacy of our testing framework with current technology [2]. We use the efficacy ratio (r_s^d), i.e. the ratio of the number of detected issues to the number of subject apps, to indicate the efficacy of a framework. As shown in Figure 6, for execution issues, their r_s^d is 6.7% (2 issues out of 30 apps), ours is r_s^d is 49.4% (44 out of 89); for background (BG) and no-sleep (NS) issues together, their r_s^d is 33.3% (10 out of 30), while ours is 43.8% (39 out of 89). We combine background and no-sleep issues since their work did not distinguish them. The result confirms that our framework can detect a much larger number of more serious energy issues in higher-quality apps in comparison with the state-of-the-art. This is due to the fact that our work is based on the insightful empirical findings, rather than ungrounded assumptions.

6.3 Issue Cause and Manifestation

Figure 7 demonstrates breakdown of energy issues of different causes in empirical study and experiment. "Emp." is experiment, "Exp." is empirical study. Our experiment is conducted on both new and old apps, we thus plot them with different colours and patterns. "UW" is unnecessary workload, "EFO" is excessively frequent operations, "BG" is wasted background processing, "NS" is no-sleep, "SW" is spike workload, "RE" is runtime exception. As shown in Section 6.2, the issues detected in experiment are much more costly than those in empirical study. So this result indicates that, no matter for "big" or "small" issues, UW and EFO are always the very significant root causes, which justifies our Finding 1. And these issues are exactly the issues that current technology can hardly address.

Figure 8 demonstrates the number of energy issues triggered under different contexts in our experiment, which captures more detailed manifestation characteristics of energy issues. "Norm" is `Normal`, "Fail" is `Network Fail`, "Flight" is `Flight Mode`, "Non-bg" is `Non-background`. We can see that most execution issues are manifested in the `Normal` context because many scenarios where issues occur require normal network and GPS context. For example, as we discussed above, the issue in `Leisure` showed up only when those three GIF pictures were downloaded and showing on the page. We only have six ($6/83 = 7.2\%$) background issues, which indicates that OS is competent in clearing potential bad influence of backgrounded apps at `BACKGROUND` stage. However, backgrounded apps may still suffer from no-sleep issues: `Normal`, `Network Fail` and `Flight Mode` provoke 6, 17, 13 no-sleep issues respectively. Furthermore, even though `Non-background` and `Normal` run in the same context, the former tends to provoke more no-sleep issues. In our test, it induces 17 issues. This result implies that special contexts (and `Non-background`) tend to incur anomalous behaviours of apps, such as bad use of wake-lock, and thus cause more no-sleep issues.

Figure 8 also shows that 37.3% (31 out of 83) energy issues can only be triggered under `Network Fail` and `Flight Mode`. This confirms Finding 2: special contexts, such as network fail, hide a significant number of serious energy issues.

7 RELATED WORK

Our work is related to multiple lines of research work. We will discuss three aspects: *understanding energy issues*; *detecting and diagnosing energy issues*; *fixing energy issues and optimizing software*.

Understanding Energy Issues. The style of empirical study that mines the data in repositories has been widely applied. For example, to characterize performance issues, a large body of research has been done for PC and server side software [17, 37, 55]. The first empirical study on characteristics of energy issues in the system of mobile device was done by Pathak et al. [39]. They mined over 39,000 posts from four online mobile user forums and mobile OS bug repositories, and studied the categorization and manifestation of energy issues. Xiao et al. [32] conducted a similar survey on three Android forums. The above studies involve issues in multiple layers across the system stack of mobile devices, from hardware, OS to applications. And the issue reports adopted in above studies were mostly proposed by end-users and random developers, who can hardly contribute very insightful understanding of the issues (e.g. root causes). The most related study is conducted by Liu et al. [30], investigating performance issues (including energy issues, as how they treated) reported in issue-tracking systems maintained by developers who developed the apps. However, the number of the studied energy issues is very small compared with our study. In summary, our study is targeted at app-level energy issues, and our employed issue reports are documented by professional developers of the corresponding high-quality projects. Our findings are more insightful and comprehensive.

Detecting and Diagnosing Energy Issues. Non-energy issues (e.g. security [43], compatibility [24] and performance [38] issues) can be detected plainly by analysing program artefacts. In contrast, to detect energy issues, researchers have to first learn the energy characteristics of mobile devices and apps. Hence, researchers use OS, hardware and even battery features as predictors to infer energy information at device, component, virtual machine or application level [5, 18, 19, 21, 40, 44, 51, 54]. Shuai et al. [11] and Ding et al. [23] proposed approaches to obtaining energy information at source line level. The former requires the specific energy profile of the target systems. The latter utilizes advanced measurement techniques to obtain source line energy cost.

Two pieces of work [15, 16] from Jabbarvand et al. are close to our work. Our work differs from theirs in three main aspects. First, their works mainly address the challenges of issue manifestation (how to trigger the issues), while our work addresses the challenges of both issue manifestation and detection (how to identify the real existence of the triggered issues). Second, they assume that the over-use of certain APIs is the main source of energy issues, which is also assumed by [2]. However, APIs cannot cover all usage of CPU, which is a main energy consumer [10] on smartphones. In contrast, our work analyzes the CPU usage by tracing control-flow operations at code level. Third, their evaluations show the efficacy of their techniques for triggering previously-reported energy issues, but the efficacy for triggering previously-unknown issues is not justified due to the lack of issue-detection mechanisms.

The work of Banerjee et al. [2] is the most relevant to ours. Two key differences prevent it from uncovering most serious energy issues detected in our test. The first is that we have deeper understanding of energy issues, which helped us address the execution issues and special contexts that they can not handle. The second is technical difference, as we elaborated in Section 5.5: we took CPU overuse into account when pre-designing input sequences, we developed online test-steering strategy to explore the vast searching space, we employed advanced machine learning algorithm to identify execution issues from power trace, etc. In summary, owing to our careful empirical study and well-designed testing technology, our framework surpasses the state-of-the-art in detecting all kinds of energy issues.

Fixing Energy Issues and Optimizing Software. A large amount of research effort on energy-saving for mobile devices has been focused on the main hardware components, such as the CPU, display and network interface. The CPU-related techniques involve dynamic voltage and frequency scaling [7], heterogeneous architecture [9, 25, 27] and computation offloading [20, 48]. Techniques targeting the display include dynamic frame rate tuning [13], dynamic resolution tuning [12] and tone-mapping based back-light scaling [1, 14]. Network-related techniques try to exploit idle and deep sleep opportunities [28, 47], shape the traffic patterns [6, 42], trade-off energy against other design-criteria [4, 45, 53], etc. Such work attempts to reduce energy dissipation by optimizing the hardware usage; there are also several pieces of work aiming at designing new hardware and devices [49, 50, 56]. Besides, another line of research work is dedicated to solving background and no-sleep issues [5, 29, 35].

Two pieces of work [3, 34] provide systematic approaches to optimizing software source code for energy-efficiency. In the former, Boddy et al. attempted to decrease the energy consumption of software by handling code as if it were genetic material so as to evolve to be more energy-efficient. In the latter, Irene et al. proposed

a framework to optimize Java applications by iteratively searching for more energy-saving implementations in the design space.

8 CONCLUSION

In this paper, we conducted an empirical study on software energy issues in 27 well-maintained open-source mobile apps. Our study revealed root causes and manifestation of energy issues. Inspired by this study, we fully implemented a novel testing framework for detecting energy issues. It first statically analyses the source code of app subjects and then extracts the candidate input-sequences with large probability of causing energy issues. We also devised several artificial runtime contexts that can expose deeply-hidden energy issues. Our framework effectively examines apps with the inputs and contexts under a systematic scheme, and then automatically identifies energy issues from power traces. A large-scale experimental evaluation showed that our framework is capable of detecting a large number of energy issues, most of which existing techniques cannot handle. These issues on average double the energy cost of the apps. Finally, we showed how developers can utilize our test reports to fix the issues.

ACKNOWLEDGEMENT

This research was supported in part by the China NSFC Grant (61902249, 61802164, 61872248 and 61702343), Guangdong NSF 2017A030312008, Shenzhen Science and Technology Foundation (No. ZDSYS20190902092853047), Guangdong Science and Technology Foundation (2019B111103001, 2019B020209001), GDUPS (2015). Kaishun Wu is the corresponding author.

REFERENCES

[1] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 57–70, New York, NY, USA, 2011. ACM.

[2] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 588–598, New York, NY, USA, 2014. ACM.

[3] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, New York, NY, USA, 2015. ACM.

[4] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao. Rethinking energy-performance trade-off in mobile web page loading. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 14–26, New York, NY, USA, 2015. ACM.

[5] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 40–52, New York, NY, USA, 2015. ACM.

[6] C. Chiasserini and R. Rao. Improving battery performance by using traffic shaping techniques. *Selected Areas in Communications, IEEE Journal on*, 19(7):1385–1394, Jul 2001.

[7] V. Devadas and H. Aydin. On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications. *IEEE Transactions on Computers*, 61(1):31–44, Jan 2012.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[9] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 31(2):86–95, March 2011.

[10] M. Halpern, Y. Zhu, and V. J. Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76, 2016.

[11] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.

[12] S. He, Y. Liu, and H. Zhou. Optimizing smartphone power consumption through dynamic resolution scaling. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 27–39, New York, NY, USA, 2015. ACM.

[13] C. Hwang, S. Pushp, C. Koh, J. Yoon, Y. Liu, S. Choi, and J. Song. Raven: Perception-aware optimization of power consumption for mobile games. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, pages 422–434, New York, NY, USA, 2017. ACM.

[14] A. Iranli and M. Pedram. DTM: Dynamic tone mapping for backlight scaling. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, pages 612–617, New York, NY, USA, 2005. ACM.

[15] R. Jabbarvand, J.-W. Lin, and S. Malek. Search-based energy testing of android. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1119–1130. IEEE Press, 2019.

[16] R. Jabbarvand and S. Malek. Mdroid: An energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 208–219, New York, NY, USA, 2017. Association for Computing Machinery.

[17] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[18] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 39–50, New York, NY, USA, 2010. ACM.

[19] J. Koo, K. Lee, W. Lee, Y. Park, and S. Choi. Batttracker: Enabling energy awareness for smartphone using li-ion battery characteristics. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[20] K. Kumar and Y. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, April 2010.

[21] S. Lee, C. Yoon, and H. Cha. User interaction-based profiling system for android application tuning. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 289–299, New York, NY, USA, 2014. ACM.

[22] D. Li and W. G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 46–53, New York, NY, USA, 2014. ACM.

[23] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for Android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, New York, NY, USA, 2013. ACM.

[24] L. Li, T. F. Bissyandé, H. Wang, and J. Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 153–163, New York, NY, USA, 2018. ACM.

[25] X. Li, G. Chen, and W. Wen. Energy-efficient execution for repetitive app usages on big.little architectures. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 44:1–44:6, New York, NY, USA, 2017. ACM.

[26] X. Li and J. P. Gallagher. A source-level energy optimization framework for mobile applications. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 31–40, Oct 2016.

[27] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. *SIGPLAN Not.*, 47(4):13–24, Mar. 2012.

[28] J. Liu and L. Zhong. Micro power management of active 802.11 interfaces. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 146–159, New York, NY, USA, 2008. ACM.

[29] Y. Liu, C. Xu, S. Cheung, and V. Terragni. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2016*, 2016.

[30] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

[31] Y. Liu, C. Xu, S. C. Cheung, and J. LÃij. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, 40(9):911–940, Sep. 2014.

[32] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Presented as part of the 10th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 13), pages 57–70, Lombard, IL, 2013. USENIX.

[33] A. Mallik, J. Cosgrove, R. P. Dick, G. Memik, and P. Dinda. Picsel: Measuring user-perceived performance to control dynamic frequency scaling. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 70–79, New York, NY, USA, 2008. ACM.

[34] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, New York, NY, USA, 2014. ACM.

[35] M. Martins, J. Cappos, and R. Fonseca. Selectively taming background android apps to improve battery lifetime. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 563–575, Santa Clara, CA, 2015. USENIX Association.

[36] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 260–269, Nov 2003.

[37] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 237–246, Piscataway, NJ, USA, 2013. IEEE Press.

[38] A. Nistor and L. Ravindranath. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 282–292, New York, NY, USA, 2014. ACM.

[39] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[40] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[41] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference on*, pages 1–6, 2014.

[42] C. Poellabauer and K. Schwan. Energy-aware traffic shaping for wireless real-time applications. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 48–55, May 2004.

[43] L. Qiu, Y. Wang, and J. Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 176–186, New York, NY, USA, 2018. ACM.

[44] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 107–120, Berkeley, CA, USA, 2012. USENIX Association.

[45] A. Sehati and M. Ghaderi. Energy-delay tradeoff for request bundling on smartphones. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[46] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[47] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 261–274, New York, NY, USA, 2005. ACM.

[48] K. Sucipto, D. Chatzopoulos, S. KostaÂś, and P. Hui. Keep your nice friends close, but your rich friends closer âĂŤ computation offloading using nfc. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[49] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger. A 90nm low-power FPGA for battery-powered applications. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 3–11, New York, NY, USA, 2006. ACM.

[50] L. Wang, M. French, A. Davoodi, and D. Agarwal. FPGA dynamic power minimization through placement and routing constraints. *EURASIP J. Embedded Syst.*, 2006(1):7–7, Jan. 2006.

[51] F. Xu, Y. Liu, Q. Li, and Y. Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 43–55, Lombard, IL, 2013. USENIX.

[52] H. Yamamoto, T. Hirano, K. Muto, H. Mikami, T. Goto, D. Hillenbrand, M. Takamura, K. Kimura, and H. Kasahara. Oscar compiler controlled multicore power reduction on android platform. In *Languages and Compilers for Parallel Computing*, pages 155–168, Cham, 2014. Springer International Publishing.

[53] Z. Yan and C. W. Chen. Rnb: Rate and brightness adaptation for rate-distortion-energy tradeoff in http adaptive streaming over mobile devices. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 308–319, New York, NY, USA, 2016. ACM.

[54] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 36–36, Berkeley, CA, USA, 2012. USENIX Association.

[55] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 199–208, Piscataway, NJ, USA, 2012. IEEE Press.

[56] L. Zhong and N. K. Jha. Energy efficiency of handheld computer interfaces: Limits, characterization and practice. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 247–260, New York, NY, USA, 2005. ACM.

[57] Y. Zhu, M. Halpern, and V. J. Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149, Feb 2015.