Did I Make A Mistake? Finding the Impact of Code Change on Energy Regression

Shaiful Alam Chowdhury[1], **Stephanie Gil**[1], **Stephen Romansky**[1], **and Abram Hindle**[1]

[1]**Department of Computing Science, University of Alberta, Canada**

ABSTRACT

Software energy consumption is a performance related non-functional requirement that complicates building software on mobile devices today. Energy hogging applications are a liability to both the end-user and software developer. Measuring software energy consumption is non-trivial, requiring both equipment and expertise, yet many researchers have found that software energy consumption can be modelled. Prior works have hinted that with more energy measurement data one can make more accurate energy models but this data was expensive to extract because it required energy measurement of running test cases (rare) or time consuming manually written tests. We address these concerns by automatically generating test cases to drive applications undergoing energy measurement. Automatic test generation allows a model to be continuously improved in a model building process whereby applications are extracted, tests are generated, energy is measured and combined with instrumentation to train a grander big-data model of software energy consumption. This continuous process has allowed the authors to generate and extract measurements from hundreds of applications in order to build accurate energy models capable of predicting the energy consumption of applications without end-user energy measurement. We clearly show that models built from more applications reduce energy modelling error.

Keywords: Software Energy Models, Energy Efficiency, Test Generation

1 INTRODUCTION

How can we help software developers address the limited battery life of smart-phones and mobile devices? Perhaps energy modelling tools can help. Are these energy models good enough? Can we improve existing software energy measurement models by measuring the energy consumption of more applications? These questions are motivated by the short battery life of mobile devices, such as smart-phones and tablets.

Does software energy consumption matter? The answer is *yes*. Mobile device users prioritize longer battery life when investing in their next purchase [42, 24, 55]. Similarly, developers are trying to write more energy efficient code to meet the need of consumers [36, 48, 12]. Very recently, extremely short battery life was reported by the users of Microsoft Surface Pro 3. Microsoft acknowledged that as a software bug, "Based on our investigations we can confirm that it is not an issue with the battery cells, and we believe this is something that can be addressed via software" [29]. Research has shown that energy efficiency can be improved significantly with minor code optimization [44, 22, 30, 44, 13, 47].

To develop energy efficient software, developers need feedback about the energy consumption of their software. Developers need energy measurement tools to understand the energy consumption impacts of their source code changes, including new features and refactoring. Yet, developers are not sure how to measure their code changes in terms of energy [42]. In fact, measurement related StackOverflow questions are more complicated than others and receive the least accepted answers [48]. In recent surveys, software developers acknowledged the importance of having energy related models that would help to learn and develop energy efficient systems [42, 37]. Unfortunately, the insignificant number of easy to use energy models is noticeable [42, 4, 37].

We seek to help Android developers *accurately estimate energy consumption of their software without the need for hardware instrumentation* and without physically measuring their own software's energy consumption. Instead measurements from 3rd party applications are used to build a robust model that can accurately estimate their own software's energy consumption. Measurements of 3rd party applications are

hard to find as one needs a repeatable test case and corresponding energy measurements. *These test cases are costly to build manually and are the main limitation of empirically derived models* [13]. We address this limitation by demonstrating the effectiveness of automatic test generation to collect measurements for energy models.

We propose *GreenScaler*, an easy to interpret energy model for Android applications. *GreenScaler* leverages a continuous process of test generation to build an ever more robust corpus of energy measurements. Android developers can use *GreenScaler* to diagnose if a code change impacts an application's software energy consumption. As of writing, *GreenScaler* learns from a wide variety of 472 real world Android applications, which was made possible through automatic test generation. *GreenScaler* is count based and relies on counts of system calls, CPU time, and other OS-level statistics. Modelling energy from such a large set of applications improves model robustness and exposes causes of energy consumption. The advantage of learning from such a wide variety of applications is evident with the reduced error bound of *GreenScaler*, as we show later, compared to its predecessor *GreenOracle* [13]—a model trained only on 24 Android applications with manually created tests.

The main contributions of this paper are:

1) We propose a process of continuously building and training an ever more accurate software energy consumption prediction model using automatic test generation.

2) We propose the *GreenScaler* model that can accurately estimate software energy consumption of applications without hardware instrumentation.

3) We demonstrate the lack of a definitive relationship between code coverage of tests and software power use which enables test generation without source code access.

2 BACKGROUND

We dedicate this section to explain terms that are frequently used in this paper. Our data collection tool *GreenMiner* [26], along with the importance of generating automatic test cases, is also explained. Finally, a previous energy model *GreenOracle* [13] that set the pathway of this work is described.

2.1 Power is not Energy

Power (P) is defined as the rate of work completion and measured in *watts* whereas energy (E) is the total amount of work done for a given time and expressed in *joules* [5, 10, 11]. The difference between power and energy, $E = P \cdot T$, is one of the first requirement to understand in order to develop energy efficient system. A misconception exists among developers: improving execution time automatically improves energy efficiency [37]. For example, in developers understanding on energy optimization: "no specific goals for energy usage, just 'don't be bad". Improving execution time reduces T in the equation. However, there is a chance that with the reduced execution time the CPU workload is also increased, so much so that instead of operating in the lowest power consuming state, the CPU switches to the highest power consuming state. This might have an adverse impact on the overall energy consumption. In addition, previous research observed significant improvement in energy efficiency with small source code modification that minimizes the number of tail energy components, instead of focusing on execution time [44].

2.2 Energy Bugs

An energy bug—in contrast to the traditional programming bugs producing incorrect/unwanted results—specifically leads to reduced battery life of battery-driven devices [56]. The sources of energy bugs can be of different types: *no sleep bug*—result of waking up the CPU, but not putting back to sleep mode [35]; *loop bug*—waiting for an event to happen, and thus periodically inspecting changes in a variable [43]; and *tail energy leak*—observed with some hardware components that do not stop drawing power immediately after job completion [11, 44].

2.3 System Call, CPU Time

System calls act as the bridge between an application and the operating system. For example, `socket` is a system call responsible for creating communication endpoints, whereas *read* takes the responsibility to read from a file. Counting the number of system calls of different types can thus provide us an estimation of the amount of different resource usage by an application [45, 4, 13].

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

In order to represent the CPU time expended by a process, we used the number of CPU jiffies provided by the Linux kernel. A CPU jiffy is a period of time assigned for a process to run without any intervention [28]. Counting the number of CPU jiffies is thus an indication of CPU usage by a specific process. One of the energy modeling complications is that a CPU can operate in different power consuming states, resulting in differing energy consumption for the same period of time [37]. As a CPU jiffy can be of different time intervals based on the CPU states, we expect that counting the number of CPU jiffies as the CPU usage measurements would mitigate the intricacy involved in software energy modeling.

2.4 *Physically Measuring Energy Consumption with GreenMiner*

For our data collection—measuring energy consumption and resource usage—we used *GreenMiner* [26]. *GreenMiner* is able to provide accurate energy measurements for Android applications and is widely accepted in the software energy research community [22, 13, 49, 25, 5, 4]. The main two components of this testbed—in addition to some other supporting hardware components like arduino uno, INA 219 breakout board—are four Android Galaxy Nexus phones and four raspberry PIs. Each PI is connected to a particular Galaxy Nexus, which is responsible for pushing and running test as well as collecting measurements afterwards. The responsible PI then uploads the measurements to a central server.

Some variation in energy consumption is observed in different measurements for the same test [22, 13, 5]. This is why all the *GreenMiner* based previous work measured energy consumption multiple times for a particular test and took the average. In this work, all of our tests for measuring energy and resource usage for a particular application were run 10 times and the mean value was used.

2.5 Manual Test Cases for Energy Measurements

In order to run an application on *GreenMiner*, a test script is required that can play the sequence of operations each time the application is uploaded to a phone. We used Android shell and busybox to make all our test cases. In general, a test script should emulate a normal user behavior with an application. For example, our manual Firefox script [13] opens Firefox, loads Wikipedia page for American Idol and scrolls over the page for five minutes—as if a user is reading the page. To develop such a script, we selected the pointer locations of interest and then converted to the appropriate `adb` command. One such example is `adb shell input tap 200 300`, which taps on an open application at pointer location 200 300.

Unfortunately, it is very time consuming to manually create test cases for individual applications. Furthermore, in our previous energy model, *GreenOracle* [13], we used only 24 Android applications and concluded that adding new applications in training data improves the model significantly. This provides motivation for our automated Android test generation tool, *AutoTestGen*, discussed in Section 3.2.

2.6 GreenOracle

GreenOracle followed the philosophy of modeling software energy consumption based on the counts of different resource usages, similar to *GreenScaler*. The former, however, has some serious limitations. First of all, this model was built only using 24 Android applications. Although the size of the training data was enlarged by adding different versions from those applications, this did not improve the model significantly. This is not surprising as we developed only one test case for each application—one use case scenario. As a result, different versions of the same application might execute the exact same source code for the same test case. In fact, for most of our applications we observed extremely small differences in energy consumption over its different versions. In this paper we show that a model based on such small dataset suffers from massive overfitting, and thus produces inaccurate feature set.

GreenOracle, however, advised the importance of adding more unique applications for a robust energy model, and thus showed the direction for *GreenScaler*.

3 METHODOLOGY

This section describes how to build and evaluate the *GreenScaler* model. The process of producing a big-data energy model, such as *GreenScaler*, is to: 1) collect Android applications (Section 3.1); 2) generate tests for the collected applications (Section 3.2); 3) collect energy consumption measurements, system calls measurements, and other process counters while running an application's test (Section 3.3);

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

4) add measurements to the training corpus; 5) and finally train our model. Figure 1 summarizes the process of developing *GreenScaler*. Of course we have to prove the effectiveness of this process, so model selection (Section 3.4), feature engineering (Section 3.5), and validation (Section 3.6) are also described in this section.

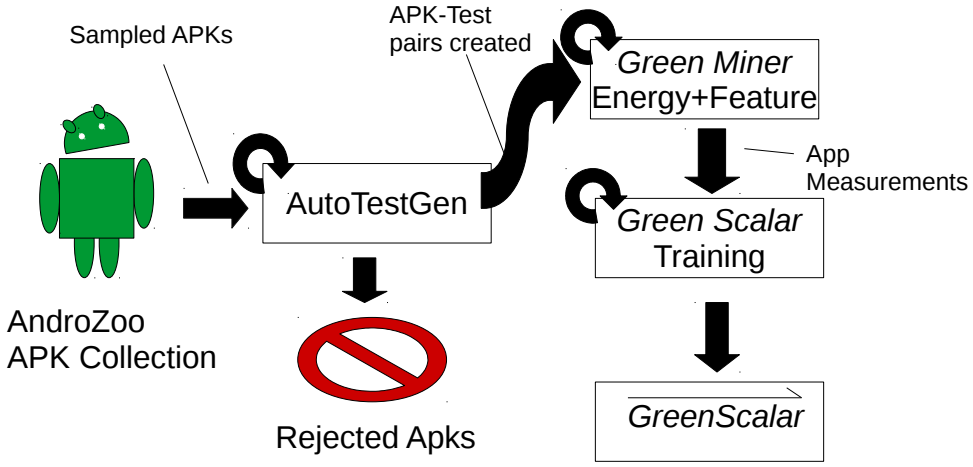


Figure 1. The process of developing *GreenScaler*. The model learns continuously with new applications using tests through *AutoTestGen*.

3.1 Collecting Android Applications

Manually collecting hundreds of Android applications from source like F-Droid [1] is a daunting task, and the types of selected applications may have bias due to the researcher's choice. Fortunately, Androzoo [7] offers millions of free to download Android applications with an objective to aid Android research. Androzoo collected applications mainly from 3 different app stores: Google Play, Anzhi, and AppChina. We sample applications randomly using the application database provided by Androzoo. This huge collection of applications, however, has its own drawback: there are a significant number of applications that do not install or run properly. We excluded those applications from our dataset. Finally, we collected 472 Androzoo applications to develop the proposed *GreenScaler* energy model.

3.2 Automatic Test Generation

Unlike manual test creation, automatic test generation does not require developer's effort. In contrast to other test generation techniques, like SimCoTest [39] which aims to generate tests with high fault revealing ability, *AutoTestGen* is our own GUI test generator that focuses on tests with high CPU usage. The reason to use CPU usage as the test fitness function instead of more intuitive and traditional code coverage approach is discussed in detail in Section 6. *AutoTestGen* is based on the approach presented in Algorithm 1.

A pool of `adb` events, such as `input tap x y`, `input swipe x1 y1 x2 y2`, `input text string/number`, `input keyevent ENTER`, was created where the values of pointer locations and strings/numbers are selected randomly in line 1 of the Algorithm. When an application is selected, *AutoTestGen* first checks if the application installs and runs properly (line 2). In case of a success, the application is then run to find the best test case—test with the highest CPU usage. When *AutoTestGen* starts generating test scripts for an application, it selects the number of events randomly—from 10 to 40 events so that a test is neither too short nor too long (line 5). A test script that is too short might not do anything useful, whereas having too many long test scripts would prolong our data collection period. If all the test scripts are of similar duration, any machine learning model would ignore duration as an important feature, which would be devastating for predicting unknown application's energy with a very different test duration. Each application is then run for a fixed 30 minutes before selecting the best test case (line 8).

Different events have different impacts on generating useful test cases. Many of our applications do not accept textual input. We did manual observations on 30 randomly chosen applications. Not

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

surprisingly, `input tap x y` was found as the most contributing event toward generating useful test cases. Events like `input swipe`, `input text`, and `input keyevents` were assigned similar priority, followed by `tapmenu`. So instead of selecting events totally randomly, *AutoTestGen* is biased so that a test script contains more number of `input tap` than any other events, whereas the event `tapmenu` occurs the least (line 9). A test script is thus a set of different events—separated by a 2 second sleep time—with possible repetitions, and the number of events in a test case might differ among the applications. After running the test script and measuring the CPU time (line 11), *AutoTestGen* checks if the assigned 30 minutes is over. If not, it again creates another test case with the previously fixed number of events and run it to compare if the CPU usage increased, in which case it updates the best test case as the most recent one. The best test case is selected as the one with the most CPU time in jiffies (line 14 – 17).

After spending 30 minutes to generate the best performing test case, *AutoTestGen* replays the best test case in order to capture the screenshots using `screencap` program and save the images (line 20). This allows us to manually investigate each application's behavior before running them on the *GreenMiner*. This is important as we found some malicious applications that can damage *GreenMiner* functionality, or modify *GreenMiner* settings which leads to inconsistent energy measurements.

input : An Androzoo application
output : Test case for the application

```
1  EVENTS_POOL={adb shell events};
2  if CrashCheck (App)==True then
3  |  Exit();
4  end
5  no_of_events ← Random(10, 40);
6  play_time ← 0;
7  cpu ← 0;
8  while play_time≤30 mins do
9  |  test_script ← SequenceOfEvents(no_of_events, EVENTS_POOL);
10 |  start_time ← GetCurrentTime();
11 |  cpu_time ← Execute(App, test_script);
12 |  end_time ← GetCurrentTime();
13 |  execution_time ← end_time-start_time;
14 |  if cpu_time>cpu then
15 |  |  cpu ← cpu_time;
16 |  |  best_test ← test_script;
17 |  end
18 |  play_time ← play_time + execution_time;
19 end
20 ScreenCap(App, best_test);
```

Algorithm 1: AutoTestGen

3.3 Collecting Energy Consumption and Resource Usage

All of our measurements, for energy and resource usage, were separate from each other so that the actual energy measurements are not effected by the programs capturing resource usage.

We used the *strace* program for tracing the summary counts of all the different system calls invoked by an application. As we observed little variation among different runs, mean counts of 10 runs were considered. In order to capture CPU usage, we used the GNU/Linux proc file system: `/proc/stat` for capturing global information and `/proc/pid/stat` for capturing information local to a particular process [13]. Global information is necessary because of the possibility of generating background processes by an application that can contribute highly to the total energy consumption. These two files provide the CPU time in jiffies both locally and globally, in addition to other pertinent information such as number of context switches, and number of page faults. For capturing global information, we take the difference of counts between after and before running a test. Similar to energy measurement and system

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

calls tracing, mean of CPU time, number of context switches, and number of page faults were taken from 10 different runs.

One more important feature that was totally ignored in *GreenOracle* is an application's interface color. In case of OLED screen, up to 40% reduction in screen-based energy consumption is achievable by switching interface with light background to dark background [33]. With such dependency on color, an energy model would be inaccurate if it does not consider screen color information. All the four phones connected to *GreenMiner* have OLED screens. The number of captured images from `screencap` unfortunately varies among different runs for the same test. We resolved this issue by running the `screencap` program five times and considering the one with the highest number of screenshots. Motivated by Dong *et al.* [14], we calculated the red, green, and blue values (RGB) for all the pixels in a screen and took the average. The average of these average values—one average from each screen—was then reported as the final RGB values. Each of these three averages (R, G, and B) is then multiplied with the test duration—as we aim to model total energy consumption instead of power.

3.4 Algorithms for Energy Models

We used three machine learning algorithms to select the best one as our *GreenScaler*: Ridge regression, Lasso, and *Support vector regression* (SVR). Ridge regression and Lasso are the simplest of the available regression algorithms and are very similar except their penalty terms. To avoid overfitting training data, Ridge uses l_2 penalty in contrast to l_1 penalty by Lasso [23]. The biggest advantage with these algorithms are that they are very simple to interpret. This helps developers to know which features and thus which segment of code are responsible for most of the energy consumption.

SVR, on the other hand, is much more complicated and can exhibit better performance than simple linear regressions in many cases [52]. The interpretation of such a model, however, is difficult and can be complicated for many software developers to find which features are contributing more toward energy consumption. For that matter, we only used the linear kernel instead of the more complicated sigmoid, radial basis function (RBF), and polynomial kernels. Ridge and Lasso were implemented in Octave. SVM^{light} implementation was used for SVR that is based on ε-SV regression [52] with the objective to find a function $f(x)$ that does not deviate more than ε from the ground truth.

3.5 Feature Engineering

3.5.1 Grouping System Calls

We observed some of the system calls, in spite of their very similar functionality, have different names. For example, according to the *man* page [2], both `fsync` and `fdatasync` do the same work—"synchronize a file's in-core state with storage device". If we treat these system calls the same, then applications that use either can benefit from training. We found many system calls with this property. Instead of treating them separately, similar system calls are grouped together. A detailed description of all the grouped system calls were reported in [13]. In general, if an application invokes 10 `fsync` and 10 `fdatasync`, a new feature Fsync (group name) was used with 20 counts in our model.

3.5.2 Feature Selection and Scaling

Compared to the number of training examples, the number of features in our dataset is quite large—22 from CPU and pertinent information, 4 for R, G, B and duration, and 99 groups from system calls. Moreover, some of the system calls are found only in a few applications' data. This large number of features with missing data leads to massive overfitting. Most importantly, a model with so many features is not helpful for the developers to understand the impact of code change on energy consumption. Among the three algorithms we used in this paper, SVR is hard to interpret and does not help us to find the best set of features. Between Ridge and Lasso, Lasso with $l1$ penalty term yields a more sparse coefficient vector (i.e., many features with coefficient 0) than Ridge and thus makes it more suitable for feature selection. However, because of high correlation among the features, Lasso selects many features with negative coefficient—a phenomenon that made the previous *GreenOracle* [13] less interpretable with reduced accuracy.

We addressed this issue by using a recursive feature elimination method with Lasso. After the first iteration, we manually removed the set of features with very low coefficients. We followed this procedure until we got a set of features with reasonably high coefficients. This procedure subsequently deleted highly correlated features. We also found that using lots of versions of a single application is harmful

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

Table 1. Selected features to model energy consumption for Android applications. The weight of each feature is explained in Section 4.1.

Features	Description	Weights
User	Number of CPU jiffies for normal processes executing in user mode	1.034e-2
Nice	Number of CPU jiffies for niced processes executing in user mode	8.660e-3
CTXT	Total number of context switches	8.000e-5
Major Faults	Number of major page faults for a process	1.117e-2
Duration	Length of the test case in seconds	6.300e-1
Red	Average level of red from screens · duration	5.000e-4
Green	Average level of green from screens · duration	4.000e-4
Blue	Average level of blue from screens · duration	5.200e-4
Fsync	System calls (fsync & fdatasync) to synchronize a file's state to disk	1.310e-3
bind	System call to bind a name to a socket	6.033e-2
recvfrom	System call to receive a message from a socket	5.000e-5
sendto	System call to send a message to a socket	1.761e-2
Dup	System calls (dup, dup2, dup3) to duplicate a file descriptor	5.406e-2
Poll	System calls (poll and ppoll) to wait for some event on a file descriptor	2.920e-3

for feature selection. For this reason, we did not use any of the *GreenOracle* dataset. Instead, 80% of the measurements from Androzoo (i.e., 377 randomly selected applications out of 472) were used for feature selection. Table 1 shows the final set of features that were selected from the feature selection process. As we show later, this very small number of features is indeed capable of estimating virtually any Android application's energy consumption. This small set of features also helps to understand what segment of code change can actually impact energy consumption. For example, if a code change invokes more `dup` system call than before, a developer can directly examine the modified file related code for possible optimization.

Machine learning algorithms perform slowly and suffer from low accuracy if the variance among the feature values is very high [54]. In our case, we indeed observed such high variance. This was solved by using 0-1 normalization, which normalizes any measured feature between 0 and 1.

3.6 Testing and Cross Validation

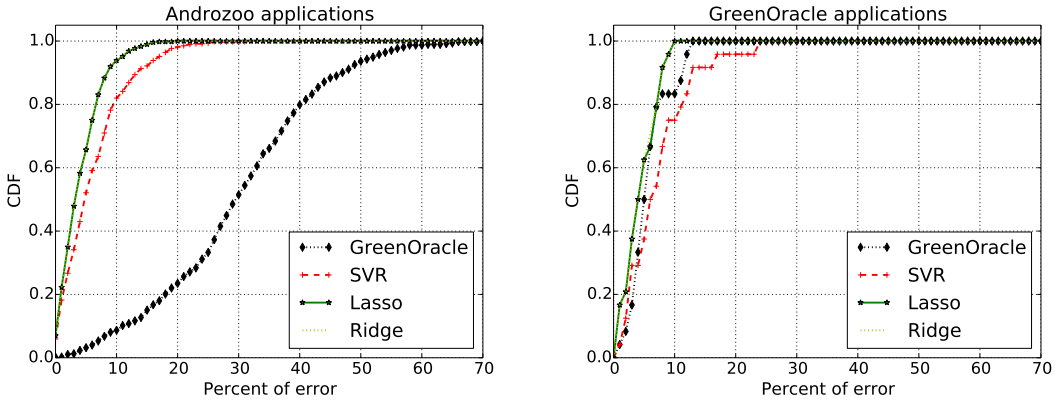
In order to avoid overfitting training data, we applied 10-fold cross validation for all the three algorithms to tune the penalty parameters. And again, only 80% of the Androzoo applications were used with none from the previous *GreenOracle* dataset. This provides an opportunity to evaluate our model on a dataset that was never used for feature selection, training, or for cross validation. The coefficients of the penalty terms for all the three algorithms were finalized during this cross validation phase. It is important to note that when an application is under test—either from *GreenOracle* dataset or Androzoo—that specific application, and all of its versions for *GreenOracle* dataset, were excluded from training data.

4 EVALUATING *GREENSCALER*

As of writing, *GreenScaler* is trained on 472 Androzoo applications. In case of accuracy testing, however, the application under test is excluded from training. We also evaluate *GreenScaler* on the 24 *GreenOracle* [13] applications with all of their versions.

Figure 2 (a) shows the Cumulative Distribution Function (CDF) of estimation percentage error in *joules* of our three selected algorithms compared with the previous *GreenOracle* for all the collected Androzoo applications. The significantly worse performance of *GreenOracle* is not surprising, as it was trained only on 24 applications, which led to the selection of inappropriate features with inaccurate coefficients. For example, according to the *GreenOracle* model, more number of `recvfrom` system call leads to less energy consumption [13], which is not realistic. With the new set of 472 applications and accurate feature set, all the three algorithms outperform *GreenOracle* with very large margin. Lasso and Ridge perform very similarly and show better accuracy than SVR. In case of Lasso, for example, almost 94% of the applications energy was estimated with an upper bound of 10% error. In extreme cases, the upper bound was only 15% compared to the 70% worst case error with *GreenOracle*. The

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017



(a) Random 472 Androzoo applications

(b) *GreenOracle*'s 24 applications

Figure 2. Models' accuracy (percent of error in joules) for both dataset. *GreenScaler* (Lasso) outperforms the old one in both cases. Mean error was considered for applications with multiple versions in Figure 2 (b).

indistinguishably similar performance of Lasso and Ridge is because of the very small (close to zero) penalty coefficient obtained from the cross validation phase. When the coefficient is zero, there is no difference between $l1$ and $l2$ penalties. With the very small number of features—features that indeed contribute to excess energy consumption—none of the models overfit the training data set, which led to a negligible penalty coefficient. However, during the feature selection phase, Lasso was very different than Ridge and helped us to find a good performing set of features. As a reward, we title Lasso with the coefficients in Table 1 as the final *GreenScaler*.

The previous *GreenOracle* was based on 24 applications. As the feature selection phase also involved those 24 applications, the model performed well for that set of applications. *GreenScaler*, on the other hand, has no information on those 24 applications. Encouragingly, *GreenOracle*, even for its own dataset, is outperformed by *GreenScaler* (Figure 2 (b)). The upper error bound for the new model is only 10% (i.e., with Lasso), in contrast to the 13% error with the old one. This is another demonstration of the robustness of *GreenScaler*.

4.1 The Final Model

Table 1 shows the final *GreenScaler* energy model. Although we evaluated our model using normalized features to train the model faster—for each single application under test, we had to train the whole model again with all other applications—the table shows coefficients without normalization so that one can interpret the model without complication. The difference in accuracy between model with and without normalization, however, is very negligible (below one percent) because of the selected very small and accurate set of features. The good accuracy of this model is self explanatory.

The table shows that sources of energy consumption in Android systems are related to CPU usage, test duration, screen color for OLED screen, file operations (Fsync, Dup and Poll), and data communication (sendto, recvfrom, and bind). The model suggests that transmitting (`sendto`) is more expensive than receiving (`recvfrom`), which is complemented by previous research [40]. Moreover, in terms of pixel color intensity, blue is the most expensive and green is the least expensive, exactly what was observed by Dong *et al.* [14]. Interestingly, our model suggests that memory related operations, except for handling major page faults, do not have significant impact on energy, which was also observed by Hasan *et al.* [22].

4.2 Finding Energy Regression in Source Code Changes

The main strength of *GreenScaler* is that it is precise, as it maintains similar shape between estimations and ground truths for all the versions of any particular application. This is exactly what a developer would want as they would be more interested to know if more energy consumption was introduced by the

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

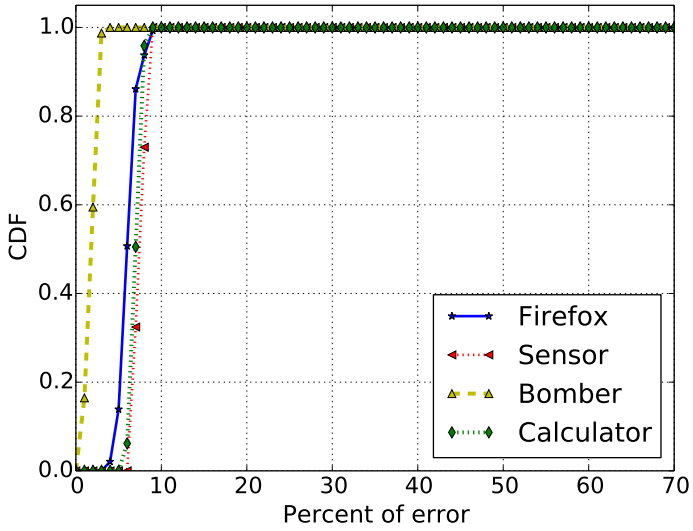


Figure 3. *GreenScaler* (Lasso) maintains similar error distribution among different versions of the same application. This is helpful to identify the unoptimized versions of an application.

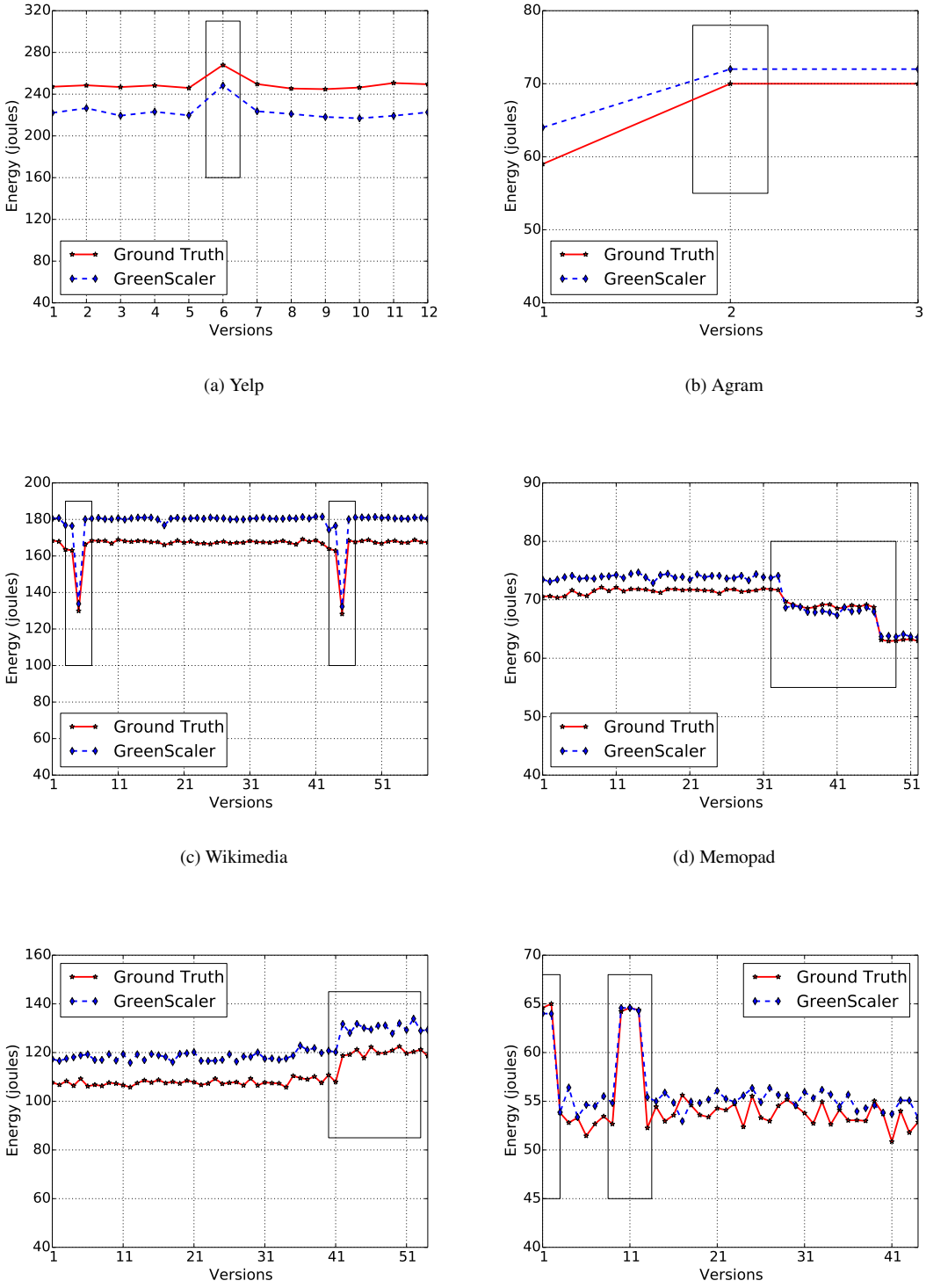
recent source code modification. We selected four applications from *GreenOracle* dataset that have lots of versions. Figure 3 shows that although *GreenScaler* accuracy varies among the applications, the error distribution is very similar among all the versions for the same application.

In order to further demonstrate the adeptness of *GreenScaler* in segregating energy buggy versions from the optimized ones, we select six of the applications from *GreenOracle* dataset. Unlike other applications, these six applications contain versions with very different energy profiles. In order to understand how energy profiles evolved over time, we sorted the versions based on the actual committed time.

Figure 4 shows that for all the six applications, *GreenScaler* successfully separates out the energy buggy versions. In the case of Yelp, a travel & local application, only one version has very different energy profile than all the others. *GreenScaler* distinguished that version accordingly. We observe similar performance for all other applications. Memopad, a drawing application, exhibits three interesting energy profiles throughout its life time—it got more and more energy efficient over time in contrast to Agram and Pinball. Our proposed model very accurately distinguished those three phases.

Now that *GreenScaler* is adept to know if a code change is scary (or advantageous) in terms of energy consumption, we ask: can it help to find out exactly which part of code modification caused the change in energy consumption? The answer is *yes*, and the success lies within the very simplistic philosophy that *GreenScaler* is built on—simple counts of different features. Here we provide two examples to support our claim. In case of Agram, an application to generate anagrams, our dataset contains only three versions. Figure (b) clearly suggests that version 2 and 3 consumes more energy than version 1. *GreenScaler* suggests that the number of context switches has increased very significantly from version 1 and stays similar to 2 and 3. Our first impression was that code related to thread interaction could have been modified. We used `git diff` to verify our intuition. We found that all the Java methods responsible for generating anagrams were indeed modified to `synchronized` methods. It is well-known that `synchronized` methods, when not optimized properly, fight unnecessarily for shared locks, which subsequently leads to more CPU usage [3, 48]. Similarly, we investigated the continuous improvements of Memopad in terms of energy consumption. The only difference in our model among all the versions of Memopad was their RGB counts, clearly suggesting the background color was changed over time. Indeed we found three distinct background colors. A complete white background (the most expensive for OLED screen) was used for versions up to 33, which was then modified to more efficient yellow followed by even more efficient red. This articulates how significant a simple choice of background color can be for devices with OLED screens, as also observed by previous research [33].

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017



(a) Yelp

(b) Agram

(c) Wikimedia

(d) Memopad

(e) Pinball

(f) Game_2048

Figure 4. *GreenScaler*'s efficiency in differentiating versions with different energy consumption. The model maintains similar shape to the actual energy consumption across the committed versions over time. This helps developers to know if a code change was more energy consuming then before. The rectangles show the significantly different versions.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

5 THE IMPORTANCE OF BIG-DATA

Does big-data help? Once again, the answer is *yes*. Figure 5 shows the reduction in error as more applications are used in training set. From the 472 random Androzoo applications, 50 applications were sampled randomly as test instances. Using the rest of the applications, accuracy of *GreenScaler* is tested using different training sets with different number of training applications (50, 100, 150 and so on). The accuracy of the test set can also vary for the same training size based on the selected applications—some applications capture more system calls than others. This is why for each training size x, we repeated each test 100 times with 100 different combinations of x number of applications. Figure 5 shows the combined error distribution for each training size.

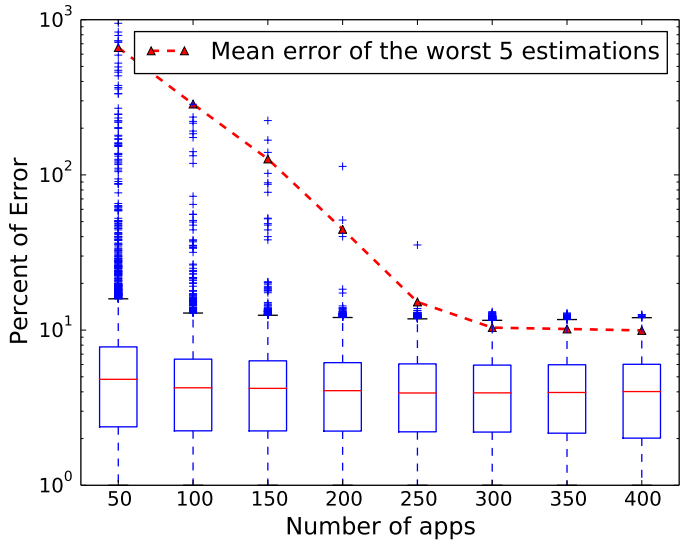


Figure 5. Error distribution Vs. training size. The accuracy improves significantly as we add more applications in training.

The box-plot graph in Figure 5 also shows that applications with high estimation errors (outliers) exist for all the training sizes. This high error, however, dwindles continuously as we add more applications in our training. In fact, with 400 applications in training the upper error-bound becomes very close to 10%. The dotted line shows the average of the 5 worst estimations with each training size. Although the decay of error rate becomes slow after adding 300 applications in training set, the least number of outliers with 400 applications suggest the possible improvement of *GreenScaler* with adding even more number of applications. In fact, the difference between the accuracy with different training sizes would be more obvious if we had not used the best set of features, which was revealed only after adding more than 400 applications.

It is evident that if we can add more applications in training, we can further improve the upper-error bound of *GreenScaler*. And this is where our test generation tool, *AutoTestGen*, becomes so useful. In fact, our data collection was still on at the time of writing this paper. *AutoTestGen* enables a continuous process that allows *GreenScaler* to improve continuously.

6 CODE COVERAGE VS. POWER USAGE

This section explains the rationale behind using CPU usage as the heuristic to select the best test case for our *AutoTestGen*.

In order to generate best test cases for our randomly selected applications, we need to ask what search-based software engineering approach we should follow. In general, we want tests with the most power consuming ability although we are modeling software energy consumption. If we set energy consumption instead of power usage as the criteria for best test, test with the highest execution time will be selected most of the time even if the test quality is subjectively poor—a test that runs for a long time

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

Table 2. Description of selected applications' master test suite coverage.

Feature	Klaxon	Password Hash	Storyhoard
SLOC	799	247	1959
Master Suite Size[1]	16	17	75
Class Coverage	24/33 (73%)	11/12 (92%)	56/81 (69%)
Method Coverage	85/150 (57%)	54/60 (90%)	306/497 (62%)
Line Coverage	383/799 (48%)	217/247 (88%)	1252/1959 (64%)
Block Coverage	1937/4378 (44%)	2225/2345 (95%)	5345/8504 (63%)

and does nothing. Given this observation, we explore the relationship between code coverage and power usage. Our methodology is a near replication of the work by Inozemtseva *et al.* [27]. The authors found that code coverage is not strongly correlated with test suite effectiveness (i.e., ability to find bugs). In our case, however, we use test suites of existing applications to determine the effectiveness of test suite's code coverage at exposing power usage—is code coverage strongly connected to power usage?

6.1 Selected Applications

We selected three open source Android applications: Klaxon—an SMS based pager that repeatedly notifies the user by means of a ringtone and/or vibration; Password Hash— an application for generating web specific passwords; and Storyhoard—create your own adventure application. The difficulty of finding open source Android applications with good Junit test suites limited our number of applications. To better match the work of Inozemtseva *et al.* [27], we also required the applications to have a coverage of at least about 50% for either class, method, line, or block. This requirement further narrowed down the available choices. Table 2 shows the characteristics of the selected applications. The varied number of lines in source code, number of methods, classes and blocks help us better to understand the relation between code coverage and power usage.

6.2 Generating Test Suites

Following the similar approach of Inozemtseva *et al.* [27], we generated test suites with different sizes. For example, a test suit of size 3 means there are 3 test cases in the test suit. We started with test suit size 2 and then repeated the procedure with test suites of sizes 4, 7, 10, 13, 16, 27, 40, 64, and 73. The selected sizes of test suites are similar to Inozemtseva *et al.*, but not identical because we used different projects in our paper. Once the sizes were decided, a Python program was written to randomly choose test cases from an application's master suite and create different sized test suites from them. For each test suite size, 100 test suites were generated, unless such 100 different combinations for a specific size was not possible. We expect this would allow us to have a diverse enough collection of test suites of various sizes and coverage levels.

For all the generated test suites, we calculated the code coverage and the associated energy consumption. Coverage was obtained by a third-party tool `emma` [16] on the source code of each application. Four different types of code coverage are supported by `emma`: line coverage, method coverage, class coverage, and block coverage. We used all of them in our analysis. We made sure to disable "*coverage true flag*" while measuring energy consumption to avoid any overhead incurred from coverage calculation. For measuring energy, *GreenMiner* was used. As discussed earlier, each suite was run 10 times to take the average energy consumption.

6.3 Result and Discussion

Table 3 shows the Pearson's correlation coefficient between code coverage (with test suite size not fixed) and power usage for all the three applications; power usage against test suite size is also presented. The strong correlation between energy and code coverage is not surprising; larger code coverage means larger execution time which has direct impact on the amount of energy consumption (i.e., $E = P \cdot T$). The correlation between energy and duration is indeed very high (minimum 0.98). In case of power, however, we observed strong correlation (Klaxon) to extremely weak correlation (Password Hash). We also evaluated the relation with fixed suite size and varied code coverage. The observation, however, is

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

Table 3. Correlation between code coverage and power usage. In addition, correlation between suit size and power is also presented.

Applications		Coverage				Suite size	Duration
		Method	Class	Line	Block		
Klaxon	**Power**	0.72	0.74	0.73	0.72	0.67	N/A
	Energy	0.84	0.85	0.86	0.85	0.80	0.98
Password Hash	**Power**	0.07	0.09	0.04	0.04	-0.17	N/A
	Energy	0.62	0.55	0.67	0.59	0.83	0.99
Storyhoard	**Power**	0.29	0.31	0.28	0.27	0.15	N/A
	Energy	0.86	0.84	0.85	0.85	0.95	0.98

very similar.

This indicates that covering more code during testing does not necessarily produce test case with high resource usage. This is very intuitive as not all code is equally power hungry; for example, a single line of code that makes a `HTTP` request might consume much more energy than other segments of code that do not ask for high CPU usage or network operations [32]. In our case—modeling software energy against the indirect measurement of resource usage—we need test cases that are more likely to exploit different resources. This led us to select test cases with the most number of CPU usage; we believe, CPU usage is also correlated to other I/O operations, as we observed more CPU jiffies leads to more number of different system call invocations [13].

7 THREAT TO VALIDITY

Modeling software energy consumption is a hard problem [37]. For example, based on the architecture, a CPU can operate in different power consuming states. Our expectation is that a CPU in high frequency state would have more CPU jiffies, which was indeed captured in our model. Our mapping mechanism—average from system call traces, CPU jiffies, and energy consumption—might not be 100% accurate as little variation between different measurements is observed. Modern mobile devices and their software are not as deterministic as one would hope. We mitigated this variation by running each scenario 10 times. External validity can be criticized for using a single version of Android phone.

8 RELATED WORK

With the increased usage of battery-driven devices, research has been done in several areas of software energy consumption: modeling software energy, techniques to find energy bugs and basic guideline for developing energy efficient system.

8.1 Modeling Energy Consumption

Most developers do not have access to real hardware-based energy measurement systems like *Green-Miner* [26]. Models have been developed to address this issue. We categorize previous models in three broad categories: instruction-based modeling, utilization-based modeling, and modeling energy using system call traces.

Instruction-based modeling refers to the technique of estimating energy consumption using program instruction cost. Energy consumption for Java-based distributed systems was modeled by Seo *et al.* [50]. The model, in addition to the communication cost, used component level energy consumption. Shuai *et al.* developed *eLens* [21]——a model based tool that can estimate energy at different levels: program level, method level, and instruction level. The basic problem of these approaches is their rigidness to one particular programming language: a model developed for a Java-based system can not be applied systems with different programming languages.

The most commonly used approach for modeling software energy consumption is the utilization-based approach [9, 51, 20, 17, 57, 15]. The philosophy of such approaches is simple and intuitive: if we know the usage time of a hardware component with its energy consumption, we can model its energy profile.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

The biggest limitation, however, is that such approaches cannot model tail energy leaks [45, 13, 5]—as tail energy is the energy consumed by a component after completing the workload assigned to it.

Modeling energy using system call traces is the most relevant approach to our work. Pathak *et al.* [45] proposed a Finite State Machine (FSM) based model using system call traces. Each hardware component is represented as a FSM which is accessed when a particular system call relevant to the component is invoked. In our earlier work, Aggarwal *et al.* [5, 4] applied system call counts to predict if energy consumption of different versions differ from each other based on the number of changed system call counts. This model, however, does not offer the actual energy consumption, and thus the developers would not be sure how bad the energy regression incurred from a change in source code is. None of these models take screen color information of applications and thus can profile other components inaccurately. The number of applications used for learning and validation is also very small compared to our dataset.

8.2 Energy Optimization and Guidelines

Wake locks are frequently used by Android developers to continue operations even when a device goes to sleep status [35]. Programmers write code to acquire wake lock before starting a task and release the lock after the task completion. Unfortunately, the code related to wake lock release might not execute ever with some unexpected run time exception. In many cases, programmers write code to acquire wake lock at Android's `onCreate` method, and write the release code at `onDestroy` method without knowing that `onDestroy` method might never execute [6]. This phenomenon severely impacts the energy consumption, rapidly emptying batteries. Pathak *et al.* [43] observed that 70% of energy bugs are the result of wake lock related problems. This is no wonder that much research [44, 6, 35, 8, 53, 46] has been conducted to characterize, detect, and minimize no sleep bugs so that the developers can optimize their code.

Another source of energy consumption is tail energy leaks. Bundling I/O operations together was found to be very effective to reduce the number of tail energy leaks [45, 11, 32]. Interestingly enough, energy consumption of smart-phones can be effected by the type of servers they communicate with. Chowdhury *et al.* observed that employing `HTTP/2` server can help significantly in reducing clients energy consumption [11]. As screen color is very sensitive for OLED screen's energy consumption, tools for automatic color transformation have been developed [33, 34]. In case of video streaming, pre-fetching has been found helpful to save energy [18]. Job off-loading to a server to save energy was also studied [41]. The overhead associated to data off-loading can be so expensive that it might even worsen energy consumption [40]. Advertisements are another source of overheads for the end users in terms of energy [19], so much so that using ad-blockers, in spite of its own cost, can actually help in saving energy [49].

Some studies concentrated on guiding developers to write energy efficient code during the development phase. Energy profiles of the frequently used Java collection framework for Android systems were formulated by Hasan *et al.* [22]. The authors found that selecting the best data structure can save 300% energy over selecting the worst one. A very similar study was conducted by Pereira *et al.* [47] for Linux server. Manotas *et al.* [38] developed a framework for automated selection of energy efficient Java collections. Ding *et al.* [31] proposed some actionable guidelines to write energy efficient code. For example, Application developers should be more concerned on the optimization of using APIs rather than their own code, and enormous amount of energy consumption can be saved by reducing idle/waiting time.

9 CONCLUSION AND FUTURE WORK

In this paper, we show the value of continuous software energy consumption model building through automatic test generation. We avoid expensive manual efforts in producing test cases to drive programs while undergoing energy measurement. This process builds *GreenScaler*, an ever improving software energy consumption model.

To support such automatic test generation, we demonstrate how irrelevant code coverage is to power measurements. In fact, code coverage correlates more with test runtime than with power usage. This prompted us to suggest test generation that aimed to produce tests with high CPU usage in order to access the primary functionality of applications.

There is a clear relationship between number of applications measured and the upper error-bound of count-based software energy consumption models. By automating formerly manual-labor intensive

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

testing work, we can continuously produce ever more accurate models that can be used by developers with no hardware-based instrumentation. We also demonstrate that these models work well in the relative case whereby version to version the model successfully predicts changes in energy consumption of an application undergoing modification.

Future work includes scaling up this big-data approach even further. There are numerous avenues of research in application choice, context of the application under test, and deriving domain-specific software energy models for software domains such as video-games, travel, or photography.

ACKNOWLEDGMENT

Shaiful Chowdhury is grateful to the Alberta Innovates - Technology Futures (AITF) to support his PhD research. Abram Hindle is supported by an NSERC Discovery Grant. The authors are also grateful to all the members of Software Engineering Research Lab, University of Alberta, for their insightful feedback.

REFERENCES

[1] F-droid: Free and open source android app repository. `https://f-droid.org/`. (last accessed: 2016-May-22).

[2] Intro Linux Man Page. `http://linux.die.net/man/2/intro`. (last accessed: 2014-May-22).

[3] Why are synchronize expensive in Java? `http://stackoverflow.com/questions/1671089/why-are-synchronize-expensive-in-java`. (last accessed: 2016-Jul-22).

[4] K. Aggarwal, A. Hindle, and E. Stroulia. Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 311–320, Bremen, Germany, Sept 2015.

[5] K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia. The Power of System Call Traces: Predicting the Software Energy Consumption Impact of Changes. In *CASCON '14*, 2014.

[6] F. Alam, P. R. Panda, N. Tripathi, N. Sharma, and S. Narayan. Energy optimization in android applications through wakelock placement. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014.

[7] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[8] Banerjee, Abhijeet and Chong, Lee Kee and Chattopadhyay, Sudipta and Roychoudhury, Abhik. Detecting Energy Bugs and Hotspots in Mobile Apps. In *FSE 2014*, pages 588–598, Hong Kong, China, Novemeber 2014.

[9] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the USENIXATC'10*, 2010.

[10] S. Chowdhury, K. Luke, J. Toukir, Imam Mohomed, S. Varun, K. Aggarwal, A. Hindle, and G. Russell. A System-call based Model of Software Energy Consumption without Hardware Instrumentation. In *IGSC '15*, Las Vegas, US, December 2015.

[11] S. Chowdhury, S. Varun, and A. Hindle. Client-side Energy Efficiency of HTTP/2 for Web and Mobile App Developers. In *SANER '16*, Osaka, Japan, March 2016.

[12] S. A. Chowdhury and A. Hindle. Characterizing energy-aware software projects: Are they different? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 508–511, New York, NY, USA, 2016. ACM.

[13] S. A. Chowdhury and A. Hindle. Greenoracle: Estimating software energy consumption with energy measurement corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 49–60, New York, NY, USA, 2016. ACM.

[14] M. Dong, Y.-S. K. Choi, and L. Zhong. Power modeling of graphical user interfaces on oled displays. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 652–657, New York, NY, USA, 2009. ACM.

[15] M. Dong and L. Zhong. Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems. In *Proceedings of the MobiSys '11*, pages 335–348, June 2011.

[16] Emma. EMMA: a free Java code coverage tool. `http://emma.sourceforge.net/`. (last accessed: 2016-JUL-22).

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

[17] J. Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *WMCSA '99*, pages 2–10, New Orleans, Louisiana, USA, February 1999.

[18] N. Gautam, H. Petander, and J. Noel. A Comparison of the Cost and Energy Efficiency of Prefetching and Streaming of Mobile Video. In *Proceedings of the 5th Workshop on Mobile Video*, MoVid '13, pages 7–12, Oslo, Norway, February 2013.

[19] J. Gui, D. Li, M. Wan, and W. G. J. Halfond. Lightweight measurement and estimation of mobile ad energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, pages 1–7, 2016.

[20] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John. Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 141–150, 2002.

[21] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating Mobile Application Energy Consumption Using Program Analysis. In *ICSE '13*, pages 92–101, 2013.

[22] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 225–236, New York, NY, USA, 2016. ACM.

[23] T. Hastie, R. Tibshirani, and J. Friedman. Linear methods for regression. In *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics.

[24] Hern and a. Alex. Smartphone now most popular way to browse internet – ofcom report. `https://www.theguardian.com/technology/2015/aug/06/smartphones-most-popular-way-to-browse-internet-ofcom/`. (last accessed: 2016-Jul-29).

[25] A. Hindle. Green Mining: Investigating Power Consumption Across Versions. In *ICSE '12*, pages 1301–1304, June 2012.

[26] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *MSR 2014*, pages 12–21, Hyderabad, India, May 2014.

[27] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[28] Jiffy. Linux Man Page. `http://man7.org/linux/man-pages/man7/time.7.html/`. (last accessed: 2016-Jan-10).

[29] B. Jones. Microsoft has found the source of recent surface pro 3 battery woes. `http://www.digitaltrends.com/computing/microsoft-surface-pro-3-battery-getting-patch/`. (last accessed: 2016-Jul-30).

[30] D. Li and W. G. J. Halfond. Optimizing energy of http requests in android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2015, pages 25–28, New York, NY, USA, 2015. ACM.

[31] D. Li, S. Hao, J. Gui, and W. G. J. Halfond. An Empirical Study of the Energy Consumption of Android Applications. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 121–130, Victoria, BC, Canada, September 2014.

[32] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond. Automated energy optimization of http requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 249–260, New York, NY, USA, 2016. ACM.

[33] D. Li, A. H. Tran, and W. G. J. Halfond. Making Web Applications More Energy Efficient for OLED Smartphones. In *ICSE 2014*, pages 527–538, Hyderabad, India, June 2014.

[34] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Optimizing energy consumption of guis in android apps: A multi-objective approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 143–154, 2015.

[35] Y. Liu, C. Xu, S. Cheung, and V. Terragni. Understanding and detecting wake lock misuses for android applications. In *FSE 2014*, Seattle, WA, USA, Nov 2016.

[36] H. Malik, P. Zhao, and M. Godfrey. Going green: An exploratory analysis of energy-related questions.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 418–421, Piscataway, NJ, USA, 2015. IEEE Press.

[37] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 237–248, New York, NY, USA, 2016. ACM.

[38] 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, 2014.

[39] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann. Simcotest: A test suite generation tool for simulink/stateflow controllers. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 585–588, New York, NY, USA, 2016. ACM.

[40] A. P. Miettinen and J. K. Nurminen. Energy Efficiency of Mobile Clients in Cloud Computing. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 4–4, Boston, MA, USA, June 2010.

[41] M. Othman and S. Hailes. Power Conservation Strategy for Mobile Computers Using Load Sharing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(1):44–51, January 1998.

[42] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about the energy consumption of software? *IEEE Software*, pages 83–89, 2015.

[43] 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, 2011.

[44] A. Pathak, Y. C. Hu, and M. Zhang. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *EuroSys '12*, pages 29–42, Bern, Switzerland, April 2012.

[45] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained Power Modeling for Smartphones Using System Call Tracing. In *EuroSys '11*, pages 153–168, Salzburg, Austria, April 2011.

[46] P. S. Patil, J. Doshi, and D. Ambawade. Reducing power consumption of smart device by proper management of wakelocks. In *Advance Computing Conference (IACC), 2015 IEEE International*, pages 883–887, June 2015.

[47] R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, and J. a. P. Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, pages 15–21, 2016.

[48] G. Pinto, F. Castor, and Y. D. Liu. Mining Questions About Software Energy Consumption. In *MSR 2014*, pages 22–31, 2014.

[49] K. Rasmussen, A. Wilson, and A. Hindle. Green Mining: Energy Consumption of Advertisement Blocking Methods. In *GREENS 2014*, pages 38–45, Hyderabad, India, June 2014.

[50] C. Seo, S. Malek, and N. Medvidovic. Component-level energy consumption estimation for distributed java-based software systems. volume 5282 of *Lecture Notes in Computer Science*, pages 97–113. Springer Berlin Heidelberg, 2008.

[51] A. Shye, B. Scholbrock, and G. Memik. Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *IEEE/ACM MICRO 42*, pages 168–178, New York, NY, USA, December 2009.

[52] V. Vapnik. *The nature of statistical learning theory*. Springer, 2000.

[53] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 387–399, Seattle, WA, USA, April 2014.

[54] C. wei Hsu, C. chung Chang, and C. jen Lin. A practical guide to support vector classification, 2010.

[55] V. Woollaston. customers really want better battery life. `http://www.dailymail.co.uk/sciencetech/article-2715860/Mobile-phone-customers-really-want-better-battery-life-waterproof-screens-p` `html`. (last accessed: 2015-APR-22).

[56] J. Zhang, A. Musa, and W. Le. A Comparison of Energy Bugs for Smartphone Platforms. In *MOBS'13*, pages 25–30, San Francisco, CA, USA, May 2013.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017

[57] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2419v3 | CC BY 4.0 Open Access | rec: 24 Feb 2017, publ: 24 Feb 2017