GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora

Shaiful Alam Chowdhury
Department of Computing Science
University of Alberta
Edmonton, Canada
shaiful@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
abram.hindle@ualberta.ca

ABSTRACT

Software energy consumption is a relatively new concern for mobile application developers. Poor energy performance can harm adoption and sales of applications. Unfortunately for the developers, the measurement of software energy consumption is expensive in terms of hardware and difficult in terms of expertise. Many prior models of software energy consumption assume that developers can use hardware instrumentation and thus cannot evaluate software running within emulators or virtual machines. Some prior models require actual energy measurements from the previous versions of applications in order to model the energy consumption of later versions of the same application.

In this paper, we take a big-data approach to software energy consumption and present a model that can estimate software energy consumption mostly within 10% error (in *joules*) and does not require the developer to train on energy measurements of their own applications. This model leverages a big-data approach whereby a collection of prior applications' energy measurements allows us to train, transmit, and apply the model to estimate any foreign application's energy consumption for a test run. Our model is based on the dynamic traces of system calls and CPU utilization.

1. INTRODUCTION

In recent years, the popularity of battery-driven devices, such as smart-phones and tablets, has become overwhelming. People now roam around with small computers—i.e., smart-phones and tablets—in their pockets [41]. According to eMarketer, the number of smart-phone users will exceed two billion by 2016 [21]. This enormous adoption led to a significant increase in mobile data traffic, and is predicted to increase 10-fold between 2014 and 2019 [19]. Recent developments have equipped these hand-held devices with different types of peripherals and sensors including digital cameras, Wi-Fi, GPS, etc. These advancements have elevated the expectation of the users. Consequently, application developers are compelled to develop more sophisticated applications, leading them to continually update and maintain their products. Such updates, however, can be harmful in terms of applications' energy efficiency [28]; most software developers are not aware of how their source code changes might have drastic repercussions on their application's energy consumption [37].

More energy consumption leads to shorter battery life, but mobile users are reluctant to frequently charge their device battery. A recent survey reveals that a large fraction of the smart-phone users desire longer battery life more than other non-functional requirements [35]. Yet the improvement in battery technology does not keep up with the advancements in computing capabilities. This indirectly emphasizes the importance of energy efficient software development.

The first impediment towards developing energy friendly applications is to know the actual energy consumption. This requires tools that are not only expensive and time consuming to develop, but demand expertise as well—a far cry from what most of the software developers have even today [37]. Among different energy related topics such as energy optimization, software developers mostly suffer from the measurement related issues [40]. Yet the number of models and tools to estimate software energy consumption and to locate energy bugs are noticeably low [37, 11]. Some of those models [25] are also too complicated for the developers to use or reproduce.

In this paper, we propose an accurate and simple resource count-based energy prediction model (*GreenOracle*) that is trained on a large corpus of Android applications' energy measurements that we evaluate on unseen Android applications. Our contribution can be summarized as: inspired by the power of system call traces [12, 39, 17] for estimating resource usage, we incorporate techniques from *Mining Software Repositories (Green Mining)* to build an energy model for Android applications. In addition to the counts of different system calls, CPU utilization, and pertinent information were added to our model. We collected 984 versions from 24 different Android applications, mostly from their open source repositories. We then profiled each version of the applications with their associated energy consumption and resource usage statistics (i.e., traces of system calls and CPU utilization) using *Green Miner* [28].

This significantly large and varied collection of data enabled us to train our energy model with high predictive ability. The proposed model does not need any energy measurement of the application under test, thus can save developers time by mitigating the complexity of measuring actual energy consumption using perplexing hardware instrumen-

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901763

tation. Software developers can download the *GreenOracle* model which is trained on many other applications. After capturing the resource usage of their applications, *GreenOracle* can be used to estimate the energy consumption in *joules*. When a developer modifies the source code, they can re-run the energy model for the new version to discover if the energy consumption exceeds their predefined threshold. Appalled at such a regression, the developer proceeds to optimize their source code so that the energy regression is within an acceptable limit. A complete workflow to apply *GreenOracle* is presented in section 6.

2. BACKGROUND AND RELATED WORK

In this section, we start with defining terms that are used in this paper. The related work is discussed in two broad categories: 1) modeling software energy consumption and 2) improving energy efficiency.

2.1 Power and Energy

The rate of doing work is defined as power. Power is measured in *watts*. Energy, on the other hand, is expressed as the total sum of power integrated over time and is expressed in *joules* [12]. An operation that uses 4 *watts* of power for a period of 30 seconds can be stated to consume 120 *joules* of energy. Although energy is proportional to power, using less power does not necessarily indicate consuming less energy. In energy optimization, a module can have higher power usage than another module with the same functionality, but can still consume much less energy if its runtime is sufficiently shorter [18].

2.2 System Calls

A system call is the gateway to access process and hardware related services and acts as the interface between a user application and the kernel [12, 39]. Different groups of system calls are responsible for different types of services: process control related system calls are used to initiate and abort processes; memory related system calls include remap memory addresses, synchronize a file to a memory map etc.; some of the most frequently used system calls related to file operations are file read, file write, file open.[1] As system calls are the only way to access such services, we hypothesize that counting the numbers of different system calls invoked by an application should roughly indicate the types and amount of resources required for an appointed task to complete.

2.3 Modeling Energy Consumption

2.3.1 Energy Modeling based on Component's Utilization

Carrol *et al.* [16] studied the energy consumption of the Openmoko Neo Freerunner, an Android smartphone. After observing and capturing energy usage in different scenarios, a simple energy model was developed. For example, $E_{audio}(t) = t \times 0.32W$ is the model to calculate the energy consumption for audio playback. Shye *et al.* [44] employed a logger application to log system performance metrics and user activities. The authors found that screen and CPU power consumption contributes highly toward the total energy drain. In a similar study by Gurumurthi *et al.* [24], disk was found to be the largest consumer of energy. Utilization based energy models were also studied by Flinn *et al.* [22], Zhang *et al.* [47], and Dong *et al.* [20].

The basic philosophy of utilization based approaches is to capture a component's utilization time to model its energy consumption. Such approaches, however, suffer from the tail energy leaks [39]—some components (e.g., NICs, GPS) stay in a high power state for a period of time even after completing the appointed task. The utilization of a component does not include this time period, thus tail energy can not be modeled with such energy modeling approaches.

2.3.2 Instruction Based Modeling

In order to estimate software energy consumption using program instruction cost, Shuai *et al.* proposed *eLens* [25]. *eLens* takes three types of inputs: a software artifact; system profiles which uses per instruction energy models; and the workload. *eLens* itself consists of three separate components: a workload generator which is responsible for creating a new instrumented version of the software artifact and can generate sets of paths in the application from the workload; an analyzer which estimates energy consumption using system profiles and sets of paths; and the source code annotator to produce the annotated version of the source code so that the developers know which line of code or part of code is energy expensive. Seo et al. [43] modeled energy consumption for Java based distributed systems. The proposed method considered component level energy consumption along with communication cost.

Instruction based models are mostly language dependent. A model developed for Java-based systems are hard to reproduce for other systems—systems developed with multiple programming languages, for example.

2.3.3 Energy Modeling from System Call Traces

Pathak *et al.* [39] proposed a model with several finite state machines (FSM). Each state in a FSM represents the power usage patterns for a specific component. Unlike the traditional utilization-based models, this FSM-based model was found to be more accurate when the tail energy leaks are severe. Yet such a model requires re-calibration for a totally new platform, which becomes more challenging when different FSMs have to be rebuilt for different energy sensitive device components. Aggarwal *et al.* [12] proposed a system call count based model which does not require complex FSMs. The author proposed a simple rule of thumb: "a significant change in the number of system calls indicates a significant change in the total energy consumption". The authors validated their model by evaluating the energy changes in different versions of two Android applications. This model, however, was not evaluated for a new application—an application that was never used in the training set. Moreover, it does not offer the actual energy consumed by an application except predicting whether energy consumption has changed or not.

We developed an energy model, inspired by the promises shown by system call traces, that can predict the total energy consumption for any Android application. We also included CPU usage as system calls are unable to capture this information [17]. Our proposed model enables Android developers to know the actual energy consumption, in addition to giving them the ability to compare different versions of the same application's energy consumption.

[1] http://man7.org/linux/man-pages/man2/syscalls.2.html last accessed: 05-Oct-2015

2.4 Energy optimization

Research has been done to understand different methods of improving software energy efficiency. I/O components are some of the dominant sources of smart-phones' energy drains [15]. This is partly because of the tail energy leaks that are common to exist when energy bugs are not considered carefully [38, 39]. In order to reduce the tail energy leak, bundling I/O operations together has been suggested [38].

For suitable jobs—when offloading data itself is not very expensive—transferring task to fixed servers can be compelling towards saving energy [36]. Unfortunately, a separate study revealed that for most of the mobile applications data offloading is too expensive to offer any gain in energy saving [34].

Automatic color transformation was offered by Li *et al.* [32] as another avenue to improve software energy efficiency. The objective is to have less energy expensive interface colors (e.g., black background) while maintaining the readability at the same time. Likewise, pre-fetching in video streaming has been suggested by Gautam *et al.* [23]. Rasmussen *et al.* [42] observed that ad-blockers, in spite of their own energy consumption, can help in reducing energy drains.

3. METHODOLOGY

Our proposed energy model takes the counts of different system calls and CPU utilization statistics of an Android application's test case as the input and produces the estimated *joules* of energy consumption of the test case as the output. In order to develop such a big-data based energy model, we followed the following steps. 1) We collected a large number of Android applications with their committed versions (section 3.1). 2) Energy measurements along with the resource usage patterns were captured for all of the versions (section 3.2, 3.3, and 3.4). 3) A grouping mechanism was used in order to deal with some system calls that have different names but similar characteristics (section 3.5). 4) Feature selection was used for identifying only the important features from the set of system calls and CPU related information that cause energy drains (section 3.6). Finally, models are developed using machine learning algorithms for regression (section 3.7) and validated using a separate cross validation set (section 3.8). Although our objective is to produce a simple linear regression based model (i.e., ridge regression), performance of some other algorithms are also presented. This is to evaluate if the prediction accuracy can be improved with added complexities of learners, such as Support Vector (SV) regression, and to understand the implication of adding more applications' data in our training corpus (e.g., bagging). This section describes each of these phases.

3.1 Collecting Versions of Android Applications

In order to build a robust generalized energy model, we need a significant number of measurements for training (i.e., applications' resource usage against actual energy consumption). The problem with having many different Android applications is that separate test cases are required for each application that are time consuming to develop and test. On the contrary, a single test case is sufficient for many of the different versions of the same application. For example, only one test script was sufficient to run and collect measurements for all the 156 Firefox versions (Table 1).

F-Droid [3], a free and open source android repository, was used to select applications from different domains—browser, game, utility, etc. *F-Droid*, however, usually contains at most three different versions of an application. This hinders the objective of having a sufficiently large training corpus with the least possible effort. As a result, we collected the source code for a significant number of commits of applications that are available on *GitHub*. We focused on a group of applications that are not only different in nature, but also have a large number of commits on *GitHub*—more commits offer more versions of the same application. The APKs for the committed versions were then generated using the *Apache Ant* tool. For a few applications, such as Firefox and ChromeShell, we collected the APKs directly from their APK repositories. For a few others, we collected the apks directly from *F-Droid* or other similar sources—when the source code was not on *GitHub* or the *Apache Ant* was not successful. Some of the applications with only one version were selected to test *GreenOracle*'s accuracy in predicting energy consumption for a wide range of applications.

Table 1 describes all the collected applications and their types; the total number of Android applications is 24 whereas the total number of versions is 984. A total of 106 unique system calls was observed in our dataset implying the richness and diversity of our training data.

3.2 Green Miner

Green Miner [29], a hardware-based energy profiler, was used for energy and resource profiling. *Green Miner* includes a YiHua YH-305D power supply, Raspberry Pi model B computer for controlling the experiments, Adafruit INA219 breakout board and Arduino Uno for collecting energy drain, and a Galaxy Nexus phone as the client. Four different setups with four Galaxy Nexus phones were used to speedup the data collection process. The Raspberry Pi pushes and executes tests on the phone and aggregates the measured data to store into a centralized server. Wi-Fi was re-enabled after enabling the airplane mode; this ensures the actual energy measurement is not contaminated by cellular radio and bluetooth. More details about Green Miner are available in prior work [29, 42].

3.3 Developing the Test Scripts

A separate test script was developed for each of the application which emulates a simple use case for a specific application. For example, in order to create a to do list for the Temaki application, a test script is required that can create a new list, enter some entries to the list and then delete the completed entries. These test scripts were automatized by injecting various touch inputs into the input systems using rudimentary Unix shell available on Android [29].

3.4 Collecting Energy and Resource Usage of the Applications

The final phase is to collect the resource usage, indirectly of course, of an application test run and the corresponding energy consumption, which enables the development of our proposed energy model. *Green Miner*, using the Raspberry Pi, pushes the test script with the input APK to run, collect, and store the respective energy consumption for a specific test case of an application. Each run was repeated 10 times in order to produce a stable average energy consumption; this was to address the observed variation among

Applications	Type	No. of versions	No. of unique system calls	Time period of commits of versions	Source
Firefox	Browser	156	84	Jul, 2011 - Nov, 2011	APK repos [4]
Calculator	Android Calculator	97	48	Jan, 2013 - Feb, 2013	GitHub
Bomber	Bombing game	79	47	May, 2012 - Nov, 2012	GitHub
Blockinger	Tetris game	74	56	Mar, 2013 - Aug, 2013	GitHub
Wikimedia	Wikipedia mobile	58	67	Sep, 2015 - Aug, 2015	GitHub
Sensor Readout	Read sensor data	37	51	Apr, 2012 - Apr, 2014	GitHub
Memopad	Free-hand Drawing	52	47	Oct, 2011 - Feb, 2012	GitHub
Temaki	To do list	66	50	Sep, 2013 - July, 2014	GitHub
2048	Puzzle game	44	60	Mar, 2014 - Aug, 2015	GitHub
ChromeShell	Browser	50	76	Mar, 2015 - Mar, 2015	APK repos [1]
Vector Pinball	Pinball game	54	48	Jun, 2011 - Mar, 2015	GitHub
Budget	Manage income & expense	59	56	Aug, 2013 - Aug, 2014	GitHub
Acrylic Paint	Finger painting	40	49	Apr, 2012 - Sep, 2015	GitHub
VLC	Video/Audio player	46	61	Apr, 2014 - Jun, 2014	APK repos [9]
Eye in Sky	Weather app	1	77	Sep, 2015	Google Play
AndQuote	Reading quotes	21	51	Jul, 2012 - Jun, 2013	GitHub
Face Slim	Connect to Facebook	1	65	Nov, 2015	Fdroid
24game	Arithmetic game	1	50	Jan, 2015	Fdroid
GnuCash	Money Management	16	56	May 2014 - Aug, 2015	GitHub
Exodus	Browse 8chan image board	3	60	Jan, 2010 - Apr, 2015	GitHub
Agram	single/multi word anagrams	3	46	Apr, 2015 - Oct, 2015	Fdroid
Paint Electric Sheep	Drawing app	1	66	Sep, 2015	Google Play
Yelp	Travel & Local app	12	78	Unknown	APK4Fun
DalvikExplorer	System information	13	54	Jun,2012 - Jan, 2014	code.google [2]

Table 1: Description of the applications

different runs of the same test case [17, 12]. System calls were traced using the simple Linux *strace* command; the *-c* option was used to produce the summarized counts of different system calls invoked by an application. In order to model the CPU usage with the energy, we collected the total CPU jiffies (A Linux CPU utilization measurement) used by our applications along with other relevant information such as the number of context switches, total interrupts, and major page faults. The Linux */proc* pseudo-file system was used for capturing CPU jiffies and other information about processes. Information local to a process was collected by accessing */proc/pid/stat* [8], and information global to the system behavior was captured from */proc/stat* [7]. The global information is not associated with any specific process; so we measured the difference of resource usage before and after the test case. In case of process specific information, however, capturing information at the end of our test run was sufficient.

The problem is that instrumented code such as running *strace* in parallel to the actual application can contaminate the application's energy measurement; instrumentation is work, and work consumes energy. This led us to enforce isolation in our measurements. For a single representational data point, we collected the energy consumption 10 times separately from *strace* and *stat* programs. Similarly, for the same test case *strace* was run 10 times followed by another 10 runs which access the */proc* file system. After taking the average from all the measurements, we mapped the values to a single example in our training dataset. In a word, a single data point in the training set required 30 different runs in *Green Miner*, which is an indication of our effort and time given to collect data for the 984 Android versions (a total of approximately 30,000 test runs). Our publicly shared dataset can be accessed and used for future research [5].

3.5 Grouping System Calls

We observed that in spite of their very similar characteristics, some system calls come with different names [17]. For example, *lseek* and *_llseek* are two system calls with the same purpose. This is problematic for our energy model when one of the applications call *lseek* whereas another one call *_llseek*. A generalized energy model would be hard to develop with such inconsistency in the training data. A model that has never seen *_llseek* in the training phase, does not know the contribution of *_llseek* towards the energy drains although the model knows the role of *lseek* which can be directly used for *_llseek*. We resolved this issue by manually grouping similar system calls together based on their semantics as described in Linux *man* page [6]. System calls that are unique in their functionality were not grouped with others. All the grouped system calls are presented in Table 2. For example, an application with 10 *write* and 10 *pwrite* is represented with a new feature called *Write* with 20 as its value in the training dataset.

3.6 Feature Scaling & Feature Selection

Machine learning algorithms often suffer when the ranges of values are very different among different features. As we observed such wide variety among the features, we normalized our data in the range 0 to 1. Such normalization not only speedups the learning time, but also improves the accuracy in prediction very significantly; features with vary large values, regardless of their importance, influence the model more than small valued features. Equation 1 was used for our feature normalization [13], where $\mathbf{x}$ is a feature vector.

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})} \quad (1)$$

After applying grouping, the dataset has 98 features from system call traces along with the 21 features that we got from *stat* program (for CPU and related information). Including duration of the test case, the total number of features is 120. This large number of features overfit the training data—although we have approximately 1000 Android APK versions, the number of applications is 24. Moreover, we are also interested in identifying the influential features that contributes to the actual energy consumption. Some feature selection algorithms like forward and backward selection suffer from local optimization problem—decisions can not be altered once a feature is selected or dropped [31].

Groups	System calls	Semantics
Lseek	lseek, _llseek	"Reposition read/write file offset"
Write	write, pwrite	"Write to a file descriptor"
Writev	writev, pwritev	"Write data into multiple buffer"
Read	read, pread	"Read from a file descriptor"
Readv	readv, preadv	"Read data from multiple buffer"
Open	open, openat	"Open a file"
Statfs	fstatfs64, statfs64, statfs, fstatfs	"Get filesystem statistics"
Stat	lstat64, stat, fstat, lstat, fstat64, stat64	"Get file status"
FSync	fsync, fdatasync	"Synchronize a file's in-core state with storage device"
Pipe	pipe, pipe2	"Create pipe"
Clone	clone, __clone2	"Create a child process"
Utime	utime, utimes	"Change file last access and modification times"
Dup	dup, dup2, dup3	"Duplicate a file descriptor"

Table 2: Grouping similar system calls according to OS semantics

Algorithms like exhaustive search are very time consuming, yet did not produce the best set of features that offer an accurate prediction model.

We also observed high correlation among different features, which is problematic for coefficient based feature selection methods like Ridge regression and Lasso. Elastic Net, however, works better with such scenarios and has been selected as our feature selection algorithm [48]. We also applied recursive elimination in order to have the least number of possible features with high predictive power. The only drawback of Elastic Net was that it deleted test duration from the set of important features after few rounds of the recursive elimination. We, however, used our domain knowledge and added test duration as one of the selected features; an application without doing anything can still consume energy if it is open. In fact, we observed significant improvement in prediction accuracy after including test duration in our feature set. It is important to note that we used 70% of the applications for the feature selection purpose. Table 3 shows the selected features for our prediction models.[2] Once the features are measured and normalized, we applied them to the machine learners to model energy consumption.

3.7 Algorithms to Model Energy Consumption

In order to build the proposed model of predicting energy consumption in *joules*, we have used four different machine learning algorithms. This is to select the best prediction algorithm (i.e., *GreenOracle*) that is not only accurate in estimating energy consumption, but also simple to use and interpret.

3.7.1 Ridge Regression

Linear regression is perhaps the simplest learning algorithm to build a regression model. Ridge regression is just an extension of simple linear regression with an added penalty expression which is used to restrict the size of the coefficients in order to avoid overfitting. The outline of the algorithm can be described as follows. Given a set of labelled instances $\{[\mathbf{X}^i, Y^i]\}$, ridge regression finds a coefficient vector $\boldsymbol{\theta} = (\theta_0, \theta_1, \ldots, \theta_n)$, which can find the best linear fit, $\mathbf{Y_p} = \boldsymbol{\theta}^T X$, where the predicted values $\mathbf{Y_p}$ minimizes the sum of the squared error. This can be formalized as in equation 2 [26]:

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}} [\sum_{i=1}^{m} (Y^i - \sum_{j=0}^{n} \theta_j X_j^i)^2 + \lambda \sum_{j=1}^{n} \theta_j^2] \qquad (2)$$

In our case, m is the number of versions, n is the number of selected features from the traces of system calls and CPU related information, $\mathbf{X}^i$s are the feature vectors, Y^is are the observed energy consumption, and $\mathbf{Y_p}$ is the vector of predicted energy consumption. The parameter λ is used for penalization in order to avoid overfitting the training data.

3.7.2 Lasso

One of the characteristics of ridge regression is that it does not eliminate unnecessary features—no feature will have a coefficient of zero. Lasso, on the other hand, drops features from a group of highly correlated features. The only mathematical difference between ridge and lasso is the penalty term in equation 2; lasso uses l_1 (i.e., $\sum |\theta_j|$) penalty instead of l_2 (i.e., $\sum \theta_j^2$) [26].

3.7.3 Support Vector Regression

Unlike the other algorithms that we implemented with Octave, we used the SVM^{light} [30] implementation with linear kernel for the SV regression. SVM^{light} is implemented based on ϵ-SV regression [45] where the main goal is to find a predictor function $f(x)$ that does not deviate more than ϵ from the true values. Success using linear kernel—instead of more complicated radial basis function, polynomial, and sigmoid kernels—is more beneficial, as linear features produce more interpretable results [27].

3.7.4 Bagging

In unstable learning, high variance is observed with little change in the training data [14]. As we test the accuracy of our models for different applications, and the training sets are a little bit different each time (the application under test is excluded), we need to verify how a little change in the training data affects the model. Bagging with ridge regression is used for this verification. We run ridge regression 100 times with replacement in the training set so that some of the applications are not included in a particular run. If the models are very different among different runs of bagging, our data collection is not adequate yet. Both the mean and median of the bagging predictions from 100 different runs are presented in our result analysis.

[2]In the table, an actual system calls starts with lower case letter

Features	Description
User	Number of CPU jiffies for normal processes executing in user mode
CTXT	Total number of context switches
Num_threads	Total number of threads created during execution
Intr	Total number of interrupts serviced during the test
Vsize	Virtual memory size
Duration	Length of the test case
recvfrom	System call to receive a message from a socket
Fsync	System calls (fsync &fdatasync) to " synchronize a file's in-core state with storage device"
setsockopt	System call for setting socket options
mkdir	System call for making a new directory
futex	System call for locking fast user-space
Write	System calls (write and pwrite) for writing to a file descriptor
sendto	System call to send a message to a socket
unlink	System call to delete a name from the file system to make the space reusable
Open	System calls (open and openat) to open a file

Table 3: Selected features from the traces of system calls and the CPU related information

3.8 Cross Validation

We evaluated the accuracy of our models using each application separately; when an application was under test, all the versions of that application were excluded from the training set. For parameter tuning, such as λ in equation 2, we separated out the versions of one of the applications from the training set; this specific set was used as the cross validation set. When we observed good accuracy in both training and cross set, the cross set was again combined with the training set to produce a final model. This model was then used to predict the energy of the versions of our application under test—application that was neither used for training nor for cross validation. The same tuned parameters ($\lambda = 0.001$ for ridge regression, for example) was then used for testing other applications' energy consumption. For example, when testing the accuracy of the application Firefox, we formed the training set with all of the versions from the other 23 applications. Now using $\lambda = 0.001$ for ridge regression, we trained the model and evaluated the prediction accuracy for 156 Firefox versions. As the accuracy was very similar across all versions of a particular application, we represented prediction accuracy of an application as the average accuracy across all versions. Similar process was followed for all other applications and algorithms.

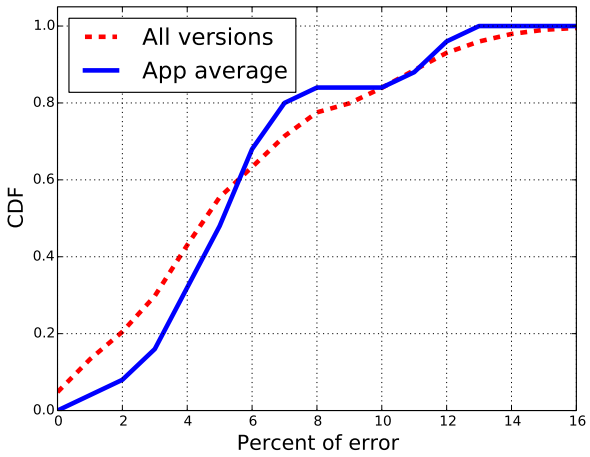


Figure 1: Percent of error with ridge regression

4. EXPERIMENT AND RESULT ANALYSIS

Table 4 shows the percent of errors when predicting the energy consumption in *joules* for all of the Android applications. A prediction of 95 *joules* against the ground truth of 100 *joules* is a 5% prediction error. All the presented results are for foreign applications; an application under test was never used in training nor in cross validation. The accuracy level varies across applications and algorithms. Although considering the average percent of error across all the applications, SV regression (with only 5.96% error) outperforms all others, simple ridge regression has shown the best performance when the worst case is considered. Ridge regression exhibits the least prediction accuracy for Exodus, an Image Board Browser, with 13.44% error. On the other hand, the worst performance of SV regression, bagging with mean, bagging with median, and lasso are 14.83%, 17.46%, 17.03%, and 19.13% of prediction errors (in *joules*) respectively.

Figure 1 illustrates the strength of the ridge regression based model more elaborately. Besides showing the cumulative distribution function (CDF) of the percent of errors for all of the 24 applications (percent of error for an application is the average of the percent of errors across all the versions of the application), it also depicts the error distribution for all the 984 versions separately. It is not surprising that the CDF curve with all the versions is different than the CDF curve only with the applications. There are lots of versions for which the energy consumption was predicted with ≈0% error. On the other hand, unlike the CDF curve of application where the worst percent of error is 13%, ridge regression has a worst case prediction error of 16% when all the 984 versions are considered. This is not surprising as our mapping mechanism—separate 10 runs for energy, system calls, and CPU usage pattern—can be a bit inaccurate, and the accuracy can vary across versions.

In spite of all the impediments with our data collection, this simple model is still very accurate in estimating the energy consumption of the Android applications under test. The CDF graph suggests that the energy consumption of approximately 85% of the applications and versions were predicted within 10% error. This is very similar to the performance of previous complicated but widely accepted tools and models—*eLens* [25] and Pathak's FSM based model [39]

Applications	% of prediction error (measured and predicted in *joules*) SVR	Ridge	Bagging (Mean)	Bagging (Median)	Lasso
Wikimedia	**3.99**	6.17	5.03	4.77	4.57
Sensor Readout	3.56	3.33	4.2	3.85	**0.73**
Bomber	7.94	7.12	8.20	**7.01**	7.21
Memopad	3.73	4.78	4.39	4.40	**3.68**
Calculator	11.36	**11.34**	11.92	11.87	14.61
Blockinger	3.61	4.09	**0.52**	1.76	4.00
Temaki	2.55	1.75	**0.89**	1.00	4.96
2048	4.05	5.11	**3.15**	3.08	4.02
ChromeShell	13.31	11.91	**10.73**	11.81	19.13
Pinball	2.32	3.93	6.31	6.07	**1.44**
Firefox	5.35	**4.83**	6.23	4.84	6.83
Budget	5.76	5.16	4.50	**4.46**	5.90
Acrylic Paint	**2.33**	4.44	5.26	5.26	2.84
VLC	7.54	5.86	**2.80**	3.20	7.18
Eye in Sky	**4.45**	6.96	9.34	10.78	7.99
AndQuote	14.83	**6.07**	11.04	10.11	12.10
Face Slim	6.58	**4.33**	7.87	8.56	6.02
24game	6.16	8.48	0.80	**0.13**	17.38
GnuCash	**4.33**	6.40	5.59	5.64	7.13
Exodus	13.96	13.44	**12.35**	12.53	12.97
Agram	6.39	6.69	5.43	**5.41**	5.82
Paint Electric Sheep	1.46	3.02	9.92	8.89	**1.01**
Yelp	**4.79**	11.95	17.46	17.03	9.49
Dalvik	2.77	**0.95**	3.65	3.08	5.03
Average	**5.96**	6.17	6.57	6.48	7.17

Table 4: Prediction accuracy of the proposed energy models: train on all but the application under test. The ground truths are the average of 10 runs and the predicted energy consumption is based on the average of 10 system call traces and 10 CPU usage traces. Error for a particular application is the average percent of error across all of its versions.

for examples. *eLens* and the FSM based models also have an upper bound of 10% error for most of the cases. With the current state—training set with 24 applications—we suggest the ridge regression based model as the best of our energy models, thus referred as the *GreenOracle*. This is encouraging as unlike SVR or bagging, models based on simple linear regression are easy to interpret, use, and reproduce.

A software developer, after capturing the set of invoked system calls and CPU usage patterns as described earlier, can directly apply our ridge regression based energy model (*GreenOracle*) as presented in Table 5 to estimate any Android application's energy consumption. The model for lasso is also presented to observe if the role of any particular feature varies towards energy prediction. These final models are developed using all the 24 applications with the tuned parameters after cross validation and exhaustive testing. Encouragingly, in both the models the role of a particular feature is same (either positive or negative) with little difference in scale.

It is important to mention that the negative coefficients in the models do not necessarily indicate their role in saving energy. In spite of applying one of the best feature selection techniques with highly correlated features, Elastic Net, we observed some features with high correlation still exists in our selected predictor set. For example, the system calls *sendto* and *recvfrom* (system calls for socket communication) are highly correlated (with correlation coefficient ≈ 0.7), but none of them were deleted from the set even

Features	Weight LR	LASSO	Minimum	Maximum
Offset	41.96	46.06	-	-
User	45.22	53.89	272.11	6686.20
CTXT	31.70	38.28	28497.93	370208.43
Num_threads	-20.89	-36.33	10.00	43.60
Intr	22.86	5.95	16837.67	193892.90
Vsize	-15.99	-16.71	477690265.60	637551820.80
Duration	84.95	88.60	42.00	200.00
recvfrom	-25.56	-24.99	94.79	6932.20
Fsync	47.99	45.49	0.00	234.20
setsockopt	15.45	16.69	0.00	195.80
mkdir	18.41	16.54	0.00	48.30
futex	-6.80	-6.01	910.70	148252.70
Write	-9.57	-7.08	43.80	12338.00
sendto	41.81	25.90	8.00	580.20
unlink	18.71	39.78	0.00	60.40
Open	18.80	17.02	8.00	1153.44

Table 5: Model description: after normalizing the features using the max and min, a developer can directly use the coefficients of the models to estimate the energy consumption of a new application.

after applying recursive elimination. In fact, the accuracy dropped significantly if we drop one of these features—to a greater extent for *sendto* and lesser extent for *recvfrom*. Models with such correlations are expected to have negative coefficients. As the models are developed with normalized feature values, maximum and minimum of all the features are also presented to enable normalization for a new application.

5. ARE THE MODELS USEFUL?

Considering the ridge regression model for example—with mostly an upper-bound of 10% error and 13% in the worst case—*GreenOracle* has two direct use cases: 1) developing an automated system to enable energy-rated mobile applications; 2) finding energy bugs incurred by any code changes in subsequent versions.

5.1 Energy-rated mobile applications

The concept of energy-rated mobile applications is yet to be adopted in spite of its urgency among the users. This is mostly due to the lack of tools and techniques required for such automated systems. Chenlei *et al.* [46] observed that significant reduction in energy consumption is possible when the users know how to select the most energy efficient application from a pool of applications with similar functionalities. The authors recommended the genesis of *Green Star*: Software Application Energy Consumption Ratings (SAECR). Johannes *et al.* [33] proposed an automated system where a new application is grouped with an existing cluster based on their functionalities. The energy consumption of the new application is measured and compared against other applications within the same cluster. The new application is then ranked based on its energy consumption. This not only improves user experience in selecting energy efficient applications, but also push the developers to consider energy efficiency in order to be competitive in the market. Such a system, however, requires a model which is able to estimate a new application's energy drains—a model with the capability of identifying different applications' energy consumption. In order to evaluate our models for such scenarios, we selected four applications from our training set that have very different energy requirement. And then we compared the actual energy consumption of these four applications (by picking a representative version from each) with the predicted values from our models. Figure 2 confirms that all of our models, in fact, are able to identify different applications' energy consumption very accurately; bagging with mean is omitted because of its very similar performance to median. As the actual energy consumption is the average of 10 different runs, the standard deviation of the measurements are also depicted.

5.2 Identifying energy sensitive code changes between subsequent versions

In continuous developments, the developers produce subsequent versions of the same application. In terms of energy efficiency, a simple code change can be colossal—both positively and negatively [38, 28]. The developers should be able to know if the changes committed for the new versions are going to cause more energy drains. This is where our energy models can be vital. The developers can use our models to estimate the energy consumption of the two versions of interest. If the new version consume more energy than

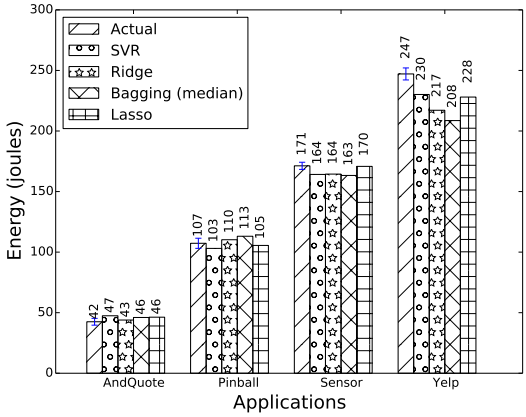
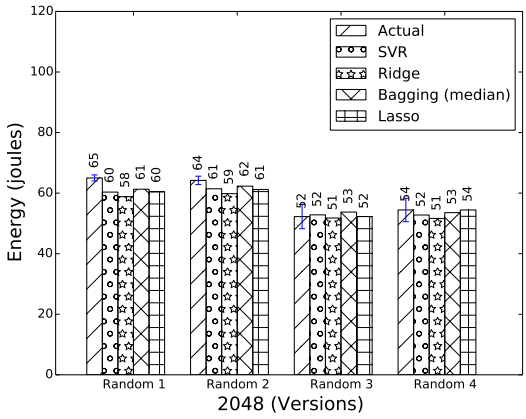


Figure 2: Models accuracy in segregating applications with very different energy requirements. Our proposed energy prediction approach is promising to enable energy-rated applications.

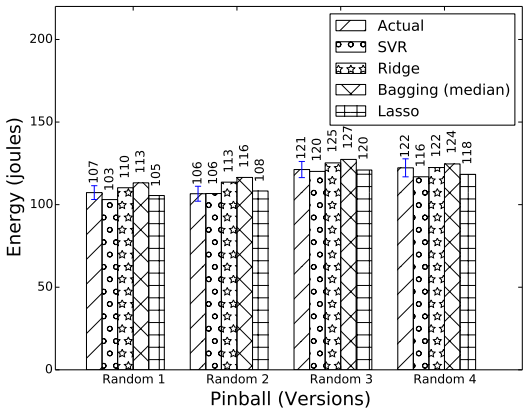
the previous one, the developers can simply investigate the changes made in the new version to find the possible energy bugs.

In order to be really useful for such scenarios, our models have to be able to identify if the energy consumption of an application has changed. For this evaluation, we selected four applications for which we found versions with totally different energy requirements than other versions. Figure 3 portrays the strength of our proposed approach in segregating versions with different energy consumption of the same application. In the 2048 Android game, we observed two different energy patterns among all the versions: versions tended to consume either around 65 *joules* or around 52 *joules*. We selected two versions from each cluster randomly and compared with our models' predictions. Figure 3 (a) clearly shows the accuracy of the estimates produced by all of our models. Similar observations can be made from Figure 3 (b) for versions of Pinball. In case of Wikimedia, all the versions have an energy consumption of around 166 *joules*, except two outliers with around *128* joules. Figure 3 (c) shows the efficacy of our proposed approach in separating those two outliers from other Wikimedia versions—the first Wikimedia versions in our dataset is used to represent others. In case of Agram, an application to generate anagrams with only three versions in our dataset, the later two versions have very similar energy drains, but are significantly different than the first one. This is clearly reflected in our prediction models in Figure 3 (d). These observations clearly indicate the accuracy of our proposed models in identifying significant changes in energy consumption between subsequent versions of applications.

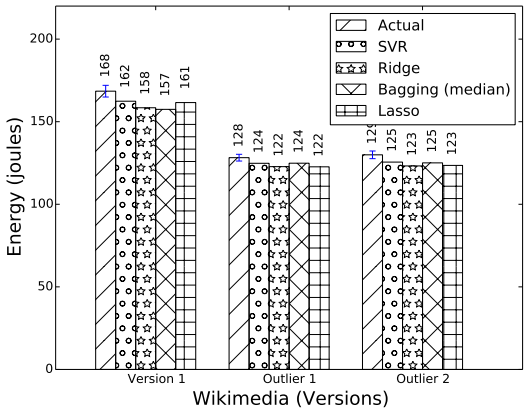
It is useful to understand why some of the versions are so different than others, in spite of their same functionality. This, however, requires a thorough understanding of different segments of the application's source code to know how different modules are connected. We selected Agram for this part of analysis as the functionalities of this application are simple, and thus the source code is easy to understand. Consequently, we ask what significant changes were made from version one to two in Agram, and how our models captured those changes?



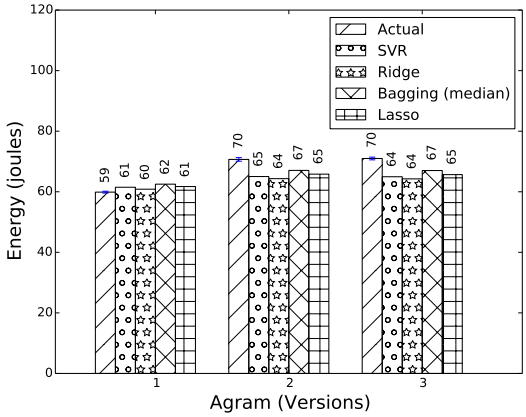
(a) 2048 Android puzzle game



(b) Pinball Android game



(c) Wikimedia (Wikipedia for mobile)



(d) Agram (generates anagrams)

Figure 3: Models' efficiency in differentiating versions with different energy consumption

Our models (ridge and lasso) suggest that the number of CPU jiffies, the total number of context switches, and the total number of interrupts have increased substantially for the next two versions of the Agram test case. Table 5 confirms that these three features have positve coefficients and thus any increase in these features contribute to more energy drains. Our first impression, especially for the changes observed with the number of context switches and interrupts, was that these changes should imply modification in thread related code. In order to verify this, we used *git diff* command to capture the code changes between version one and two. We observed a significant changes between these two versions, and the committed changes in fact support our hypothesis about thread related code. All the Java methods related to generating anagrams have been changed to synchronized methods. Figure 4 depicts such a synchronized method that returns the list of anagrams using an overloaded *generate* method. Interestingly, the efficiency of Java's synchronized methods have been castigated and reported as very resource expensive in different programming forum discussions [10, 40]. One commenter stated that lock requires more system calls and context switches that induce performance degradation [10]. This observation is encouraging as it clearly illustrates the effectiveness of our models in detecting energy bugs between subsequent versions.

We conclude that *GreenOracle* is not only able to foretell the changes in energy efficiency incurred by code changes, but is also able to provide pointers to the newly introduced energy buggy code.

6. DEVELOPER'S WORKFLOW TO ESTIMATE AND IMPROVE ENERGY CONSUMPTION

A developer can simply follow the following five steps to estimate an Android application's energy consumption in *joules* without dealing with any hardware instrumentation: 1) develop a test case using Android unit test for example; 2) run the test case in parallel to *strace* and capture counts

```
public synchronized ArrayList<String>
    generate(int n) {
    ...
    return results;
}
```

Figure 4: Agram synchronized method example

of different system calls; 3) run the test case to capture information from */proc/stat* and */proc/pid/stat* file systems to collect the CPU utilization and relevant information. In case of */proc/stat*, take the difference of before and after the test case; 4) normalize the selected features as presented in Table 5; and 5) use the ridge regression coefficients (i.e., *GreenOracle*) from Table 5 to estimate energy. After any modification in the source code, the developer can again use *GreenOracle* to check for energy consumption regression. In case of a significant change, feedback from *GreenOracle* can be used to locate possible energy bugs as we did for the Agram application.

7. TOWARDS IMPROVING THE ACCURACY OF OUR MODELS

Considering the average percent of error in predicting unseen application's energy consumption, the performance of bagging with ridge regression is very similar to the simple ridge regression (Table 4). For some applications, however, significant differences are observed. AndQuote and 24game are of such examples. This implies that our energy model with ridge regression was not yet completely stable; with little changes in the training set—exclusion or inclusion of some applications as occur in bagging—the coefficients can vary slightly. This articulates the importance of collecting more data to have an adamant and more robust energy model. For further verification, we measured the accuracy of ridge regression with an increasing number of applications in the training set. Figure 5 shows the distribution of accuracy for all of our Android applications with x number of applications in training. We observed that the accuracy varies based on the selected applications used for training; some of the applications cover more system calls than the others. As a result, we ran each of the scenarios 10 times and calculated the average percent of errors. In each run we select x number of applications randomly for training and predict the energy consumption of our application under test with the produced model. We increased the number of applications (i.e., x) from 1 to 23 and observed how the accuracy improved. With more applications in the training, the error distribution dwindles consistently.

Does big-data matter? Yes, it does. Controlling for the number of applications in training we can see that the error rates for energy prediction drop almost monotonically. This implies that we need to band together and collect measurements for more applications to produce a better energy model.

8. THREATS TO VALIDITY

Our application selection was manual and could introduce bias. The test cases we developed only executed some of the selected functionalities offered by the applications. With more test cases, more application features can be tested and new system calls can be added to the training set. The mapping mechanism (resource count to energy consumption) can be inaccurate although we ran each test 10 times to map the averages (a total of 30 runs for a single data point). External validity is harmed by the use of a single brand of smart-phone with a single version of the operating system.

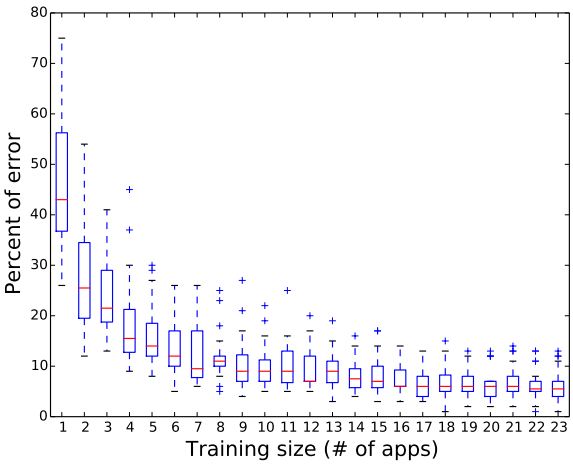


Figure 5: No of applications VS. performance

9. CONCLUSION AND FUTURE WORK

In this work we have presented an approach to model energy consumption that allows developers to estimate energy consumption without having to measure the energy of their own applications directly. The proposed *GreenOracle* model follows a MSR/big-data approach whereby CPU usage and system call counts of many applications under test are combined in order to estimate the energy consumption (*joules*) of an application under test. Through a thorough evaluation we demonstrated that *GreenOracle* can estimate joules mostly with less than 10% error, and the model can be distributed and run on unseen applications without hardware instrumentation. We also observed that the model continues to benefit from a variety of measured applications and tests of these applications.

We conclude that CPU usage statistics and system call counts are enough information to estimate the energy use of a test-run of an application based on a model tuned and trained on foreign applications.

Future work includes collecting more applications to create a public/crowd-sourced repositories of applications, test-cases, and traces in order to enable the creation of a truly big-data based energy model. Our long run goal is to develop an on-line energy model where an energy expensive system call can be directly mapped to the source code to enable bug fixing during the development phase [11].

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 Sharif Uddin, Samiul Monir, Christopher Solinas, Eddie Santos, Joshua Campbell, and Stephen Romansky for their insightful feedback.

10. REFERENCES

[1] Chromeshell apks. http://commondatastorage.googleapis.com/chromium-browser-continuous/index.html?prefix=Android/. (last accessed: 2015-May-22).

[2] Dalvikexplorer apks. https://code.google.com/archive/p/enh/downloads. (last accessed: 2015-Aug-22).

[3] F-droid: Free and open source android app repository. https://f-droid.org/. (last accessed: 2015-May-22).

[4] Firefox apks. https://ftp.mozilla.org/pub/mobile/nightly/. (last accessed: 2015-May-22).

[5] GreenOracle dataset. https://github.com/shaifulcse/GreenOracle-Data. (created on: 2016-Jan-29).

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

[7] /proc/stat explained. http://www.linuxhowtos.org/System/procstat.htm. (last accessed: 2014-May-22).

[8] THE /proc FILES YSTEM. https://www.kernel.org/doc/Documentation/filesystems/proc.txt. (last accessed: 2014-May-22).

[9] VLC apks. http://nightlies.videolan.org/build/android-armv7/backup/. (last accessed: 2015-Aug-22).

[10] Why are synchronize expensive in Java? http://stackoverflow.com/questions/1671089/why-are-synchronize-expensive-in-java. (last accessed: 2014-May-22).

[11] 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.

[12] 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.

[13] M. J. Alam, P. Ouellet, P. Kenny, and D. O'Shaughnessy. Comparative evaluation of feature normalization techniques for speaker verification. In *Proceedings of the 5th International Conference on Advances in Nonlinear Speech Processing*, NOLISP'11, pages 246–253, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] E. Alpaydin. Combining multiple learners. In *Introduction to Machine Learning (Second Edition)*. MIT Press.

[15] 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.

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

[17] 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.

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

[19] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014âĂŞ2019. Technical report, Cisco, February 2015.

[20] 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.

[21] eMarketer. 2 billion consumers worldwide to get smart(phones) by 2016. http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/1011694. (last accessed: 2016-Jan-07).

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] T. Hastie, R. Tibshirani, and J. Friedman. Support vector machines and flexible discriminants. In *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics.

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

[29] 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.

[30] T. Joachims. Making large-scale support vector machine learning practical. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods*, pages 169–184. MIT Press, 1999.

[31] M. Karagiannopoulos, D. Anyfantis, S. B. Kotsiantis, and P. E. Pintelas. Feature Selection for Regression Problems. http://www.math.upatras.gr/~dany/Downloads/hercma07.pdf. (last accessed: 2015-Oct-22).

[32] 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.

[33] J. Meier, M.-c. Ostendorp, J. Jelschen, and A. Winter. Certifying energy efficiency of android applications. In *4th Workshop on Energy Aware Software-Engineering and Development*, 2014.

[34] 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.

[35] I. Moura, G. Pinto, F. Ebert, and F. Castor. Mining Energy-Aware Commits. In *MSR 2015*, Florence, Italy, May 2015.

[36] 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.

[37] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What do programmers know about the energy consumption of software? *PeerJ PrePrints*, 3:e1094, 2015.

[38] 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.

[39] 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.

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

[41] P. Poole. Half of Us Have Computers in Our Pockets, Though You'd Hardly Know it. http://www.huffingtonpost.com/pamela-poole/smartphone-technology_b_2573671.html. (last accessed: 2014-Dec-22).

[42] 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.

[43] 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.

[44] 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.

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

[46] C. Zhang, A. Hindle, and D. German. The impact of user choice on energy consumption. *Software, IEEE*, 31(3):69–75, May 2014.

[47] 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.

[48] H. Zou and T. Hastie. Regularization and Variable Selection via the Elastic Net. http://people.ee.duke.edu/~lcarin/Minhua11.7.08.pdf. (last accessed: 2015-Oct-22).