Towards Developing Energy Efficient Mobile Applications: Models, Tools, and Guidelines

by

Shaiful Alam Chowdhury

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

Software energy efficiency has become a concern for the scale of operations in data centers and for the availability of battery-driven mobile devices. Developers now consider energy efficiency as one of the performance metrics. Unfortunately, developers are not trained enough and do not know how to produce energy efficient software. They demand guidelines and tools for developing energy efficient software. To help energy-aware developers (focusing on Android application developers), we make the following contributions.

1) Energy-aware application (app) developers, first and foremost, need to estimate their apps' energy consumption. Without any estimation, they would not know if their effort for energy optimization is effective. Considering Android as an example, we first present a reproducible machine learning-based energy consumption model. Results suggest that with more and more apps in training, we can improve the estimation accuracy of machine learning-based models. However, adding and running a new app requires writing a manual test script for the app, which hinders building an ever improving energy model. To alleviate this problem, we show that automatic random test generation with test selection heuristics can be used. These automatically generated tests can exercise the apps, so we can collect different resource usage (independent variables) and energy consumption of the apps (dependent variable).

2) We then argue that developers also need energy optimization guidelines: *what makes an app more energy efficient?* Along this direction, we provide

three different guidelines. First, we show that the new `HTTP/2` protocol offers reduced energy consumption when compared to the old `HTTP/1.1`. Also, the handshaking mechanism for adding transport layer security causes extra energy drain in mobile clients. Energy-aware web app developers should consider deploying the `HTTP/2` protocol for saving energy consumption. Second, we show that developers, if prudent, do not need to worry about the energy impact of software logging which is crucial for monitoring apps' health. They need to log less frequently by grouping small log messages together. Finally, we present a new design pattern that developers can adopt for making view updates more energy efficient. We also discuss the potential trade-offs (e.g., user experience, and software maintenance cost) that developers should be aware of before adopting a design choice.

Preface

This thesis is mainly based on five different publications, where I was the main contributor with the supervision of Professor Abram Hindle. I designed the methodology, conducted experiments, evaluated the results, and wrote the papers. In some cases, my other collaborators helped me with data collection and paper presentation.

Chapter 4 of this thesis has been published as:

- Shaiful Alam Chowdhury, Abram Hindle, "GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora", In 13th International Conference on Mining Software Repositories (MSR 2016), pages 49-60. May 14-15, 2016. Austin, Texas.

Chapter 5 of this thesis has been published as:

- Shaiful Alam Chowdhury, Stephanie Borle, Stephen Romansky, Abram Hindle, "GreenScaler : Training Software Energy Models With Automatic Test Generation", Empirical Software Engineering Journal, Springer, 2018. This paper has also been accepted as a journal first paper and was presented at the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019).

Chapter 6 of this thesis has been published as:

- Shaiful Alam Chowdhury, Varun Sapra, Abram Hindle, "Client-side Energy Efficiency of HTTP/2 for Web and Mobile App Developers", In 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), pages 529-540. March 14-18, 2016. Osaka, Japan.

Chapter 7 of this thesis has been published as:

- Shaiful Alam Chowdhury, Silvia Di Nardo, Abram Hindle, Zhen Ming (Jack) Jiang, "An Exploratory Study on Assessing the Energy Impact of Logging on Android Applications", Empirical Software Engineering Journal, Springer, 2017.

Chapter 8 of this thesis has been published as:

- Shaiful Chowdhury, Abram Hindle, Rick Kazman, Takumi Shuto, Ken Matsui, and Yasutaka Kamei, "GreenBundle: An Empirical Study on the Energy Impact of Bundled Processing", In 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019, technical track), pages 1107-1118. May 25-31, 2019. Montreal, Canada.

Below is the list of my other publications (including papers I co-authored) that I published during my PhD degree, but are not included in this thesis.

- Alexander Wong, Amir Salimi, Shaiful Chowdhury, Abram Hindle, "Syntax and Stack Overflow: A Methodology for Source Code Error and Fix Extraction", To appear in 35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019, short paper), Cleveland, USA.

- Abdul Ali Bangash, Hareem Sahar, Shaiful Chowdhury, Alexander William Wong, Abram Hindle, Karim Ali, "What do developers know about machine learning: a study of ML discussions on StackOverflow", In Proceedings of Mining Software Repositories (MSR 2019, Challenge track). May 25-31, 2019. Montreal, Canada.

- Stephen Romansky, Neil Borle, Shaiful Alam Chowdhury, Abram Hindle, Russ Greiner, "Deep Green: modelling time-series of software energy consumption", In 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017), pages 273-283. Sep 17-22, 2017. Shangai, China.

- Shaiful Alam Chowdhury, Abram Hindle, "Characterizing Energy-Aware Software Projects: Are They Different?", In 13th International Conference on Mining Software Repositories (MSR 2016, Challenge track). May 14-15, 2016. Austin, Texas.

- Shaiful Alam Chowdhury, Luke Kumar, Toukir Imam, Mohomed Jabbar, Varun Sapra, Karan Aggarwal, Abram Hindle, Russell Greiner, "A System-call based Model of Software Energy Consumption without Hardware Instrumentation", In Sixth International Conference on Green and Sustainable Computing Conference, December, 2015, Las Vegas, US.

- Shaiful Alam Chowdhury, Abram Hindle, "Mining StackOverflow to Filter out Off-topic IRC Discussion", In 12th Working Conference on Mining Software Repositories (Challenge track), May 16-17, 2015. Florence, Italy (Won the Best Mining Challenge Paper Award).

To all the tax payers of Bangladesh
For financing the costs of undergraduate studies in Bangladeshi public universities,

To my late father
For teaching me "simplicity",

To my mother
For teaching me "honesty".

Acknowledgements

First and foremost, I thank the almighty God (*Allah* in Arabic) for helping me throughout this journey.

I would like to thank my supervisor Professor Abram Hindle for his instrumental support. He was extremely helpful and kind. I sincerely thank my committee members (Dr. Thomas Zimmermann, Professor Eleni Stroulia, Professor Ken Wong, Professor Cor-Paul Bezemer, and my defence chair Professor Denilson Barbosa) for their insightful feedback. My lab mates were also very helpful, and they encouraged me during the difficult times. I am grateful to my family members (my mother, my wife Syeda Shamria Hasan, and my son Isham Ariz Chowdhury) for their continuous support.

I am grateful to the people of Bangladesh; thanks to their taxes which enabled free education in public universities.

I am also grateful for the following recognitions.

- Media Coverage: University of Alberta research could prevent app updates from draining smartphone batteries. Global News, `https://globalnews.ca/news/4643134/university-of-alberta-research-smartphone-apps-batteries/`
- NSERC Postdoctoral Fellowship, University of British Columbia, Canada
- Graduate Student Outstanding Undergraduate Teaching Award, University of Alberta
- Alberta Queen Elizabeth II Graduate Scholarship (Doctoral-level), University of Alberta
- Alberta Innovates -Technology Futures Graduate Student Scholarship, University of Alberta
- Winner: Computing Science Early Achievement Award (PhD), University of Alberta
- Devendra Jindal Graduate Scholarship, University of Alberta

- Runner-up: Computing Science Early Achievement Award (PhD), University of Alberta
- MSR Mining Challenge Award 2015
- Doctoral Recruitment Scholarship, University of Alberta, Canada

Contents

1 Introduction **1**
1.1 Software energy consumption and efficiency 2
1.2 Motivation 3
1.2.1 Estimating software energy consumption 3
1.2.2 Energy optimization guidelines 4
1.3 Contributions 5
1.3.1 Models/Tools for estimating apps' energy consumption 6
1.3.2 Enhancement in energy optimization guidelines 7
1.4 Thesis organization 9

2 Background **12**
2.1 Current, voltage, power, and energy 12
2.2 Power vs. energy 13
2.3 Energy measurement: GreenMiner 14
2.4 Tail energy 16
2.5 System calls for software energy modeling 18

3 Related Work **20**
3.1 Developers' knowledge of software energy efficiency 20
3.2 Modeling energy consumption 22
3.2.1 Instruction-based modeling 22
3.2.2 Utilization-based modeling 23
3.2.3 System call-based modeling 24
3.2.4 Other models 25
3.3 Energy optimization and testing 26
3.3.1 Energy efficient color selection 26
3.3.2 Cloud computing for saving mobile energy 27
3.3.3 Mobile energy efficiency in video streaming 28
3.3.4 Does ad blocking help to reduce energy drain? 29
3.3.5 Impact of code obfuscation and refactoring on energy . 30
3.3.6 Energy efficiency of Java collections 30

3.3.7 Detecting energy bugs and hotspots 31
3.3.8 Guidelines for energy-aware developers 33
3.3.9 Design patterns and energy consumption 34
3.3.10 Batch processing for energy efficiency 35
3.3.11 Energy specific testing 36

I Energy Estimation with Models 39

4 GreenOracle: Producing Reproducible Energy Models 40
4.1 Introduction 44
4.2 Background and related work 46
4.2.1 Power and energy 46
4.2.2 System calls 46
4.2.3 Modeling energy consumption 47
4.2.4 Energy optimization 49
4.3 Methodology 49
4.3.1 Collecting versions of Android applications 50
4.3.2 GreenMiner 51
4.3.3 Developing the test scripts 52
4.3.4 Collecting energy and resource usage of the applications 52
4.3.5 Grouping system calls 53
4.3.6 Feature scaling & feature selection 54
4.3.7 Algorithms to model energy consumption 55
4.3.8 Cross validation 58
4.4 Experiment and result analysis 58
4.5 Are the models useful? 62
4.5.1 Energy-rated mobile applications 62
4.5.2 Identifying energy sensitive code changes between subsequent versions 63
4.6 Developer's workflow to estimate and improve energy consumption 66
4.7 Towards improving the accuracy of our models 67
4.8 Threats to validity 68
4.9 Conclusion and future work 68

5 Leveraging Automatic Test Generation for Improving Energy Models 70
5.1 Introduction 75
5.1.1 Paper organization 77
5.2 Background 78

5.2.1 Power vs. energy 78
5.2.2 System calls and CPU time 78
5.2.3 Energy measurement: GreenMiner 79
5.2.4 Energy estimation: GreenOracle 81
5.2.5 Energy model building test generation 81
5.3 Code coverage heuristic 84
5.3.1 Methodology 85
5.3.2 Analysis of results 88
5.4 *GreenScaler* methodology 91
5.4.1 Collecting Android applications 92
5.4.2 Automatic test generation with resource-utilization heuristics 92
5.4.3 Collecting energy consumption and resource usage . . . 96
5.4.4 Algorithms for energy models 97
5.4.5 Feature engineering 99
5.4.6 Testing and cross validation 100
5.5 Evaluating resource utilization heuristics 101
5.6 Monkey vs. GreenMonkey 104
5.7 Evaluating *GreenScaler* 106
5.7.1 Evaluation on randomly generated tests 106
5.7.2 Evaluation on manually written tests 107
5.7.3 Qualitative evaluation of *GreenScaler* model 108
5.7.4 Evaluation on detecting energy regressions 110
5.7.5 Accuracy vs. commit size 113
5.7.6 Evaluating *GreenScaler* tool from developers' perspectives 118
5.8 The importance of more apps in training 119
5.9 Research directions for the software energy research community 119
5.10 Dataset 121
5.11 Threats to validity 122
5.11.1 Conclusion validity 122
5.11.2 Construct validity 123
5.11.3 Internal validity 124
5.11.4 External validity 125
5.12 Related work 126
5.12.1 Modeling energy consumption 126
5.12.2 Energy optimization 128
5.12.3 Energy testing 129
5.13 Conclusion and future work 130

II Guidelines **132**

6 Energy Efficiency of HTTP/2 **133**

6.1 Introduction . . . 136
6.2 Background . . . 138
6.2.1 Hyper Text Transfer Protocol (HTTP) and its limitations 138
6.2.2 SPDY and HTTP/2 . . . 139
6.2.3 Power and energy . . . 140
6.2.4 Tail energy . . . 140
6.3 Methodology . . . 140
6.3.1 GreenMiner . . . 140
6.3.2 Writing a test script . . . 141
6.3.3 Collecting Mozilla Firefox nightly versions . . . 142
6.3.4 Deploying a HTTP/2 server . . . 142
6.3.5 Workload . . . 143
6.3.6 Validation . . . 145
6.4 Experiment and result analysis . . . 148
6.4.1 World flags . . . 148
6.4.2 Gopher tiles . . . 149
6.4.3 Google and Twitter . . . 151
6.5 Discussion . . . 152
6.6 Threat to validity . . . 155
6.7 Related work . . . 155
6.7.1 Mitigation of energy bugs/hotspots in applications . . . 155
6.7.2 Performance of web protocols . . . 156
6.8 Conclusions and future work . . . 158

7 Energy Consumption of Logging in Android **159**

7.1 Introduction . . . 163
7.2 Background . . . 165
7.2.1 General approaches for software instrumentation . . . 166
7.2.2 Android logging . . . 167
7.2.3 GreenMiner . . . 168
7.3 RQ1: What is the difference in energy consumption for Android applications with and without logging? . . . 171
7.3.1 Motivation . . . 171
7.3.2 Experiments . . . 171
7.3.3 Analysis . . . 174
7.3.4 Summary . . . 181

7.4 RQ2: What are the factors impacting the energy consumption of logging on Android applications? 181
7.4.1 Motivation 181
7.4.2 Experiments 181
7.4.3 Analysis 183
7.4.4 Summary 187
7.5 RQ3: Is there any relationship between the logging events and the energy consumption of mobile applications? 187
7.5.1 Motivation 187
7.5.2 Experiments 188
7.5.3 Analysis 188
7.5.4 Summary 196
7.6 Threats to validity 196
7.6.1 Construct validity 196
7.6.2 Internal validity 197
7.6.3 External validity 198
7.7 Related work 199
7.7.1 Energy testing and modeling for mobile applications . . 199
7.7.2 Empirical studies on energy-efficient mobile development 200
7.7.3 Execution logs 201
7.8 Conclusion 203

8 GreenBundle: Addressing energy efficiency from design time 205
8.1 Introduction 209
8.2 Background 211
8.2.1 Energy efficiency is difficult to achieve 211
8.2.2 Model-View-Presenter 212
8.2.3 Events, bundling and dropping presenters 213
8.3 Methodology 214
8.3.1 The benchmark app 215
8.3.2 Energy measurements and test scripts 217
8.4 Results: benchmark app 219
8.5 Real world apps 224
8.5.1 Selection of applications 224
8.5.2 RQ 3: What are the energy impacts of bundling and dropping on real-world applications? 226
8.5.3 RQ 4: Can bundling and dropping help address users' feedback without harming apps' energy consumption? . 227
8.6 Understanding resource utilization patterns with bundling and dropping 229
8.6.1 RQ 5: Why do bundling and dropping save energy? . . 230

8.7 Maintainability analysis 231
8.8 Threats to validity 233
8.9 Related work 233
8.10 Conclusion & future work 235

III The Future 237

9 Conclusions & Future Work 238
9.1 Summary of the contributions 238
9.1.1 A reproducible energy model 239
9.1.2 Accurate energy models with automated test generation 239
9.1.3 Energy consumption of the `HTTP/2` protocol 240
9.1.4 Energy consumption of logging 240
9.1.5 Energy efficiency from the design time 241
9.2 Future work 242
9.2.1 A generic energy estimation model 242
9.2.2 Automatic energy regression testing 243
9.2.3 Other potential research 246
9.3 Concluding remarks 247

References 249

List of Tables

2.1 Specs of the Samsung Galaxy Nexus phones used for the experiments. 16

4.1 Description of the applications 51
4.2 Grouping similar system calls according to OS semantics . . . 54
4.3 Selected features from the traces of system calls and the CPU related information 56
4.4 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. 59
4.5 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. 61

5.1 Specs of the Samsung Galaxy Nexus phones used for the experiments [46]. 80
5.2 A summary of the existing Android testing tools for model building test generation. 83
5.3 Description of selected apps' master test suite coverage. 86
5.4 Correlation between code coverage and power with uncontrolled suite size. Suite size vs. power is also presented. 89
5.5 Correlation between code coverage and power with suite size fixed to 2. 90
5.6 Correlation between code coverage and power with suite size fixed to 13. 90
5.7 Categories of the 100 randomly selected AndroZoo apps. . . . 93
5.8 Grouping similar system calls according to OS semantics. . . . 99

5.9 Selected features (CPU and others, duration, colour, and system calls) from feature selection process to model energy consumption for Android apps. This table suggests that the major sources of Android energy consumption are CPU, context switches, test duration, screen color, file operations, and network operations. The weight represents the energy consumption for each unit (e.g., one CPU jiffy) of the features. The weights of each feature are discussed in Section 5.7.3. 101

5.10 99% mean confidence interval (percent of error in *joules*) of tests versus models. Results suggest that for mean confidence interval both the models have similar accuracy. The difference is negligible/small according to Cliff's delta. 103

5.11 99% mean confidence interval (percent of error in joules) of models based on Monkey and GreenMonkey generated tests. Model based on GreenMonkey tests is significantly more accurate than the model based on Monkey generated tests. 105

5.12 Description of the *GreenOracle* applications [48]. The table shows the 24 apps in the dataset with their types, numbers of versions, and the execution scenarios of the manually written test cases. 108

6.1 Description of the Workloads 143

6.2 P-Value for paired t-test among different settings for world flags with fgallery 149

7.1 Sample log events from the CALCULATOR application 168

7.2 Specs of the Samsung Galaxy Nexus phones used for the experiments. 170

7.3 The applications under test, selected from the *GreenOracle* [48] dataset. 173

7.4 Test scenarios and test results for the selected Android applications. 175

7.5 Wilcoxon Rank Sum Tests ($\alpha = 0.05$) comparing energy consumption between logging enabled versus disabled per version. $p \leq 0.05$ means that there is a statistically significant difference in the energy consumption between logging enabled and disabled, whereas $p > 0.05$ means otherwise. Cliff's δ magnitude across applications versions is from [103]. 176

7.6 Controlled experiments with varying logging rates and message sizes. 183

7.7 Percentage growth rates of energy consumption (joules) for the log generating tests. All the calculations below used the energy consumption of the idle tests as the baseline. 183

7.8 Summary of unique log events per application across all the versions. 189

7.9 Spearman's ρ correlation coefficient distribution between log event types and joules per application. Each column shows how many log events correlated with the correlation scale proposed by Hopkins *et al.* [105] 190

7.10 OS Level Log events shared by all the applications 193
7.11 Linear models of energy consumption based on log events across numerous Android applications. Top three models are shown only if they are significant ($p \leq 0.05$). 195

8.1 Percent increase of energy consumption compared with the energy consumption of 1 emitter and 1 event/second. For readability, nearest integer values are presented. 221
8.2 Energy savings (in percent) by different bundlers and droppers when compared with no bundling or dropping. Results are presented for just one emitter. 223
8.3 Description of the selected four real-world Android apps from F-droid. 224
8.4 DL values for before/after versions of each app (bundling only). 232

List of Figures

1.1 Thesis structure. 10

2.1 One of the four identical *GreenMiner* settings. Photo used with permission from the *GreenMiner* paper [102]. 15
2.2 A program that suffers from many tail energy leaks. 17
2.3 A program that suffers from one tail energy leak. 17
2.4 System calls act as the bridge between an application and the operating system. 18

4.1 Percent of error with ridge regression 60
4.2 Models accuracy in segregating applications with very different energy requirements. Our proposed energy prediction approach is promising to enable energy-rated applications. 63
4.3 Models' efficiency in differentiating versions with different energy consumption 64
4.4 Agram synchronized method example 66
4.5 Number of applications VS. performance 68

5.1 One of the four identical GreenMiner settings. Photo used with permission from the Green Miner paper [102]. 80
5.2 The process of developing *GreenScaler*. The model learns continuously with new apps using test selection by *GreenTestGen*. 91
5.3 Comparing performance between Monkey and GreenMonkey for building energy models. GreenMonkey outperforms Monkey in generating tests that are more suitable for building energy models. 105
5.4 *GreenScaler* (Lasso) outperforms *GreenOracle* with very large margin on the 472 AndroZoo apps with randomly generated tests. 106
5.5 *GreenScaler* (Lasso) outperforms *GreenOracle* even on *GreenOracle* dataset. Mean error was considered for apps with multiple versions. 109
5.6 *GreenScaler* maintains similar error distribution among different versions of an app. 111

5.7 *GreenScaler*'s efficiency in differentiating between versions with different energy consumption. Versions are sorted based on their committed times. Whenever there is significant energy difference between two versions of the same app, *GreenScaler* detects the difference. Developers can use *GreenScaler* to check for energy regression before releasing a new version. 112

5.8 *GreenScaler*'s sensitivity to commit size in detecting energy regression. Apparently, there is no (or very weak) relation between accuracy and commit size. 114

5.9 Comparing Java collections' energy consumption. Actual energy measurement from *GreenMiner* suggests that TreeList (ACC) is the most energy expensive and LinkedList (JCF) is the least energy expensive for inserting 50,000 integer elements. *Greenscaler* suggests the same and comes up with the exact same ranking. 116

5.10 Comparing Java download libraries: `Jsoup` and `URLConnect`. For the executed test, *GreenScaler* suggests that version with `Jsoup` consumes around 4 joules more than the version with `URLConnect`. This is very similar to the actual measurements from *GreenMiner*. 117

5.11 Model's accuracy against the number of apps used in training. The accuracy improves with more apps in training. This suggests that we can continuously improve the model by adding more apps with the random test generation process. 120

6.1 Comparing power usages for the same protocol with different settings 147

6.2 Power usage of different settings for world flags with fgallery . 149

6.3 Power usage for Gopher tiles with different RTTs 150

6.4 Power usage for Google and Twitter 151

6.5 Power usage over time 153

7.1 *GreenMiner* consists of an Arduino, a breadboard with INA219 chip, a Raspberry Pi, a USB hub, and a Galaxy Nexus phone connected to a Power Supply. Photo used with permission from the *GreenMiner* paper [102]. 170

7.2 Process to Investigate the Energy Impact of Logging (RQ1) . . 172

7.3 Wilcoxon Rank Sum *p*-values per application of energy consumed with logging and without logging. *p*-values less than 0.05 indicate that logging enabled and logging disabled consumed different amounts of energy. *p*-values were corrected for multiple hypotheses using Benjamini and Hochberg correction [35] 177

7.4 FeedEx Energy consumption over time. Versions 32 to 35 exhibit very different energy profiles compared to the previous versions. 180

7.5 Energy consumption against the number of log lines across different FeedEx versions. The graph depicts 2 measurements and the lines connects between adjacent versions. The line depicts how the FeedEx versions move through the space of log length and energy consumption. Essentially consecutive FeedEx versions use more and more energy. 180

8.1 UML class diagram of the benchmark app. 214

8.2 Energy consumption of the benchmark app with different numbers of emitters and event generation rates. Bars indicate the 99% confidence interval. 219

8.3 Energy consumption of bundling and dropping compared with the Nobundling versions for different number of emitters. Results for more than 32 emitters are not presented; due to resource limitations, the energy consumption is inconsistent for very high numbers of emitters and rates. Bars indicate the 99% confidence interval. 222

8.4 Energy consumption of bundling and dropping compared with the original versions of four real-world apps. Bars indicate the 99% confidence interval. 225

8.5 Energy savings of bundling/dropping for the Sensor app with higher sampling rates. Bundling/dropping time is fixed to 0.1 second. Bars indicate the 99% confidence interval. 228

8.6 Numbers of CPU jiffies and context switches for bundling and dropping compared with the original AcrylicPaint app. Bars indicate the 99% confidence interval. 230

9.1 eRED: Automatic test generation to detect energy regression between two versions. 245

Chapter 1

Introduction

Let us start with Isham's story. Isham is an independent Android app (application or software) developer. He develops apps and uploads them to the Google app store. Recently, Isham developed the second version of an on-line multi-player gaming app, after achieving moderate success with the first version. Within the first week of uploading, Isham's gaming app exceeded 10,000 downloads. Then, out of nowhere, the number of downloads dropped significantly. Isham was dumbfounded and looked for an explanation. He started reading the users' reviews and found that this version drains phone's battery much faster than the previous version. Isham started looking into this very foreign and ambiguous idea of energy efficient app development to save his client's battery life.

Isham found a survey revealing that a large fraction of the smart-phone users desire longer battery life more than other non-functional requirements [253]. Even organizations like Microsoft suffer from users' dissatisfaction when they deliver energy expensive software [127]. Isham immediately realized that the downfall of his app's popularity was not coincidental.

Isham needs an energy measurement system. With such a system, he can measure and compare the energy consumption of his updated app version with the previous version. But building an actual hardware measurement system demands money, time, and expertise [45], [101]—difficult for an individual developer like Isham. Isham wishes there was a software that he can just download and use for estimating his app's energy consumption. But what if, a software-based energy estimation tells Isham that his new app version consumes more energy? Isham realizes that he needs energy specific optimization guidelines, because traditional optimizations (e.g., reducing execution time) often do not work for reducing energy consumption. (Li *et al.* [137], Sahin *et al.* [211], and Hao *et al.* [92]).

This imaginative story of Isham's represents the actual circumstances of many real-world software and mobile app developers (Pang *et al.* [191], Manotas *et al.* [159], and Pinto *et al.* [198]), and bases the motivation of this thesis.

My thesis is that machine learning-based energy models can be developed to estimate software energy consumption with 90% accuracy. Developers can use these models to compare energy consumption of similar apps, or versions of the same app. My thesis is also that software developers can develop energy efficient software if they rely on guidelines developed through empirical evaluation: guidelines that may range from technology choice to design decisions. For example, just by selecting the HTTP/2 protocol rather than the old HTTP/1.1 developers can save ≈10% energy consumption in their clients' devices. Similarly, developers do not need to worry much about the energy impact of conservative software logging. Developers can reduce the logging frequency by bundling small log messages, because writing less than 10 log messages per second increases the energy consumption by only ≈1%. Finally, developers can (and should) address energy efficiency from the design time. A design choice, such as the bundled Model-View-Presenter, saves up to 40% energy without significantly impacting user experience and software maintenance cost.

1.1 Software energy consumption and efficiency

What exactly is software energy consumption? This is a rational question because it is the hardware components that actually consume energy to operate. These hardware components, however, are driven by software. Thus the energy consumed by a hardware component might vary based on the software that accesses and utilizes it—efficiently or inefficiently. For example, we can select the most energy efficient Java collection from a group of collections [95], given that we know their energy profiles. Energy efficiency, as defined by the

World Energy Council, is the reduction of energy consumed by a given service or activity [162]. Software energy efficiency research thus focuses on techniques and tools for aiding developers to produce energy efficient software—software that utilizes hardware components in energy efficient ways.

1.2 Motivation

Energy consumption is critical in the data-center and at the edge, on mobile devices such as smartphones. According to a data center developer, "*any watt that we can save is either a watt we don't have to pay for, or it's a watt that we can send to another server*" [159]. Server-side energy efficiency is crucial for organizations like Google for the scale of operations, as cooling becomes a very important operational factor [31]. For mobile devices such as smartphones, energy efficiency has direct impact on the devices' availability. It is unsurprising that smartphone users rate energy efficiency much higher than other non-functional requirements [253]. App developers thus need to be aware of their apps' energy consumption.

Energy efficient software development, however, is a complex process and requires support from different directions [159]. We categorize software energy research into two broad categories: estimating energy consumption, and optimization. The rationale is that to develop energy efficient software, developers need to measure or estimate their software energy consumption, and need to learn energy optimization techniques.

This thesis mainly focuses on mobile devices—Android smartphones to be more specific.

1.2.1 Estimating software energy consumption

For developing an energy efficient software, the first requirement is to measure the energy consumption of the software. For example, a developer needs to know if the new version of a software consumes more energy (and how much more) than the previous version before releasing the new version. Also, a developer might need to compare the energy consumption of her software with other already (if there is any) available software with similar functionalities. Unfortunately, energy measurement is difficult and expensive. Pinto *et al.* found that among all the categories of energy related discussions, measurements related questions are mostly unsolved. In addition, an actual hardware-based energy measurement system can be too expensive for an individual developer to afford—it can even cost around 40,000 CAD [45].

The energy research community was not silent and produced techniques and software-based tools for estimating software energy consumption [92], [184], [194]—focusing mostly on mobile devices. Unfortunately, most of these approaches are not generalizable—a tool developed for one specific mobile device is not usable or even reproducible on other devices [7]. This is because these approaches rely on measurements not available from all other devices (such as the PETrA by Nucci *et al.* [184]). Some of these tools [92] are also too complicated for the developers to use or even reproduce [7]. Developers need an energy estimation approach that is generalizable (can be reproduced for a different device), and a tool that is not only publicly available, but also easy to use.

1.2.2 Energy optimization guidelines

With the pressing need of energy efficient mobile apps, significant research has been done on guiding developers to work with software energy efficiency. Research includes fixing wake lock related problems [10], [28], [149], [192], [195], [250], finding energy efficient Java collections [95], [196], and automatic color transformation technique [142]. However, energy optimization research is a continuous process and need a lot more study [159]. For example, should developers switch to the newly introduced `HTTP/2` server for their web apps? Does it offer better energy efficiency compared to its predecessor `HTTP/1.1` for the mobile clients? This is an important question, as research suggests that network communication is one of the major sources of energy consumption for mobile devices [141]. Similar to writing features with network communications, developers frequently use software logging to keep track of unanticipated bugs/errors [123], [256], [261], [263]. Although the performance impact of logging has been studied [128] (e.g., execution time), the energy impact of software logging is unknown, and there is no guidelines on how to use logging without harming the energy consumption much. In this thesis, the energy efficiency of the `HTTP/2` protocol (on the client devices), and the energy consumption of logging are studied.

While optimizations through little tweaks (e.g., being selective between Java collections or screen colors for examples) are useful for developing energy efficient software, such optimizations can be insignificant in some real-world scenarios [211]—the energy consumption can be so high that a little energy saving through clever tweaking will not be worth developers' time. For such software, we need to make high-level design decisions that can significantly reduce energy consumption of the software. Also, these types of decisions will help developers to work on energy efficiency from the design time, before coding. This is important, because improving energy efficiency after receiving negative feedback from the users might not help to recover a developer's reputation. Unfortunately, the investigation of design choices for achieving energy efficiency is not explored widely. Energy efficient design choices deserve much more attention from the community, and motivates a part of this thesis.

1.3 Contributions

The contributions of this thesis are based on the problems that developers face towards developing energy efficient software—as presented in the previous section. This section summarizes the contributions in two different categories: modeling software energy models, and providing energy optimization guidelines for energy-aware developers.

1.3.1 Models/Tools for estimating apps' energy consumption

We propose a machine learning-based energy estimation approach for Android apps. It is a simple resource count-based energy consumption estimation model and is trained on a corpus of different Android apps' energy measurements. The biggest advantage of this approach is that it relies on common OS statistics that are available from any Linux system. Unlike most of the previous energy estimation approaches [7], this approach is thus reproducible for any Android device. Also, a publicly available, and easy-to-use open source tool is developed that the developers can download and use to estimate their apps' energy consumption. This contribution is an amalgamation of two sub-contributions: 1) evaluating the potential of a machine learning and resource count-based approach for producing a reproducible model (*GreenOracle*), and 2) Leveraging automated test generation approaches to increase the training-set size, and thus producing a more accurate energy estimation model (*GreenScaler*).

GreenOracle

GreenOracle is a machine learning-based Android energy model that trains itself using common OS statistics (e.g., CPU-utilization, file and network operations, and so on) from 984 versions of 24 third-party Android apps. *GreenOracle* is moderately accurate, and shows that the accuracy can be improved with the addition of new apps in training, rather than adding new versions of the apps already used in training.

GreenOracle was published in The 13th International Conference on Mining Software Repositories (MSR 2016) [48].

GreenScaler

GreenOracle shows that we can successfully build reproducible machine-learning based energy models. However, *GreenOracle's* recommendation about adding new apps in model training is difficult, because a manually written separate test script is required to run a new app, so the OS statistics (independent variable) and the energy consumption of the app (dependent variable) can be captured. *GreenScaler*, motivated by *GreenOracle's* recommendation, leverages automated test generation approaches and test selection heuristics for adding hundreds of apps in training without manually writing (infeasible) test script for each of the app. *GreenScaler* is much more accurate in energy estimation than *GreenOracle*, and publicly available as a tool [47].

GreenScaler was published in The Empirical Software Engineering Journal (EMSE 2018) [45].

1.3.2 Enhancement in energy optimization guidelines

The second major contribution of this thesis is to enhance the existing energy optimization guidelines, which is also a blend of three different sub-contributions: 1) helping energy-aware developers to understand the energy consumption of the `HTTP/2` and the `HTTP/1.1` protocols, 2) revealing the energy impact of logging on Android apps so developers can decide how much logging they can do without impacting energy much, and 3) providing a new energy efficient architectural design pattern (*GreenBundle*) that developers can follow from the design time, even before coding starts.

HTTP/2 Vs. HTTP/1.1

My study suggests that `HTTP/2` never performs worse (in terms of energy consumption) than `HTTP/1.1`. In fact, for networks with higher round-trip times (RTTs), `HTTP/2` has better energy consumption performance than `HTTP/1.1`. This suggests that mobile app developers should consider switching to `HTTP/2` server for reducing energy consumption of their mobile clients.

This study was published in the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering [50].

Logging vs. energy consumption

From a thorough empirical study, we found that developers need not worry about impacting energy consumption of their mobile apps if they conservatively employ logging. Small amounts of logging ($\leq$ 10 log messages per second) have little or no energy impact on the mobile apps. Under heavy logging, logging large amounts of data infrequently consumes much less energy than frequently logging smaller amounts of data.

This study was published in The Empirical Software Engineering Journal (EMSE 2018) [46].

GreenBundle

We propose *GreenBundle*, an energy efficient architectural pattern that leverages the existing Model-View-Presenter (MVP) pattern, and converts the presenter into a bundling presenter. The idea is, instead of updating the views immediately with each incoming event from the model, the presenter waits for a period of time (varies based on the apps), and then passes all the stored events in a batch to the views. The energy saving with *GreenBundle* even with real-world apps is encouraging. Also, with the *GreenBundle* pattern, energy saving can be a win-win situation; there are scenarios where *GreenBundle* can save energy without impacting app's maintenance cost and without negatively affecting users' perceived latency in view updates.

GreenBundle was published in The 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019) [84].

Contribution summary

- We present a reproducible machine learning-based energy estimation model for Android (*GreenOracle*), which relies on common OS statistics and system call traces.
- We show that automatic random test generation with test selection

heuristics can be successfully used for generating useful test cases for building more accurate energy estimation models (*GreenScaler*).

- We provide evidence that energy-aware app developers should switch to the new `HTTP/2` protocol for reducing their client devices' energy consumption.

- We suggest that developers should combine small log messages together for energy efficient logging, and they do not need to worry about energy efficiency if logging is less frequent.

- We propose a new energy efficient design pattern that developers can follow from the app design time. We also discuss the potential trade-offs that developers need to consider before making an energy efficient design decision.

1.4 Thesis organization

This thesis is structured based on the thesis contributions, as presented in Figure 1.1. After motivating the thesis objectives in Chapter 1, Chapter 2 explains the background concepts necessary to understand this thesis. Chapter 3 discusses the previous works on software energy consumption and their drawbacks.

We argue that energy-aware developers need to estimate the energy consumption of their software and require guidelines for developing energy efficient software. Along these directions, we grouped the rest of the chapters into three different parts.

Part I: Models is dedicated for showing how to make reproducible machine learning-based energy estimation models (Chapter 4), and how to leverage automatic test generation for making energy estimation models more accurate (Chapter 5).

Part II: Guidelines presents the contributions related to energy optimization guidelines. We show in Chapter 6 that `HTTP/2` is more energy efficient

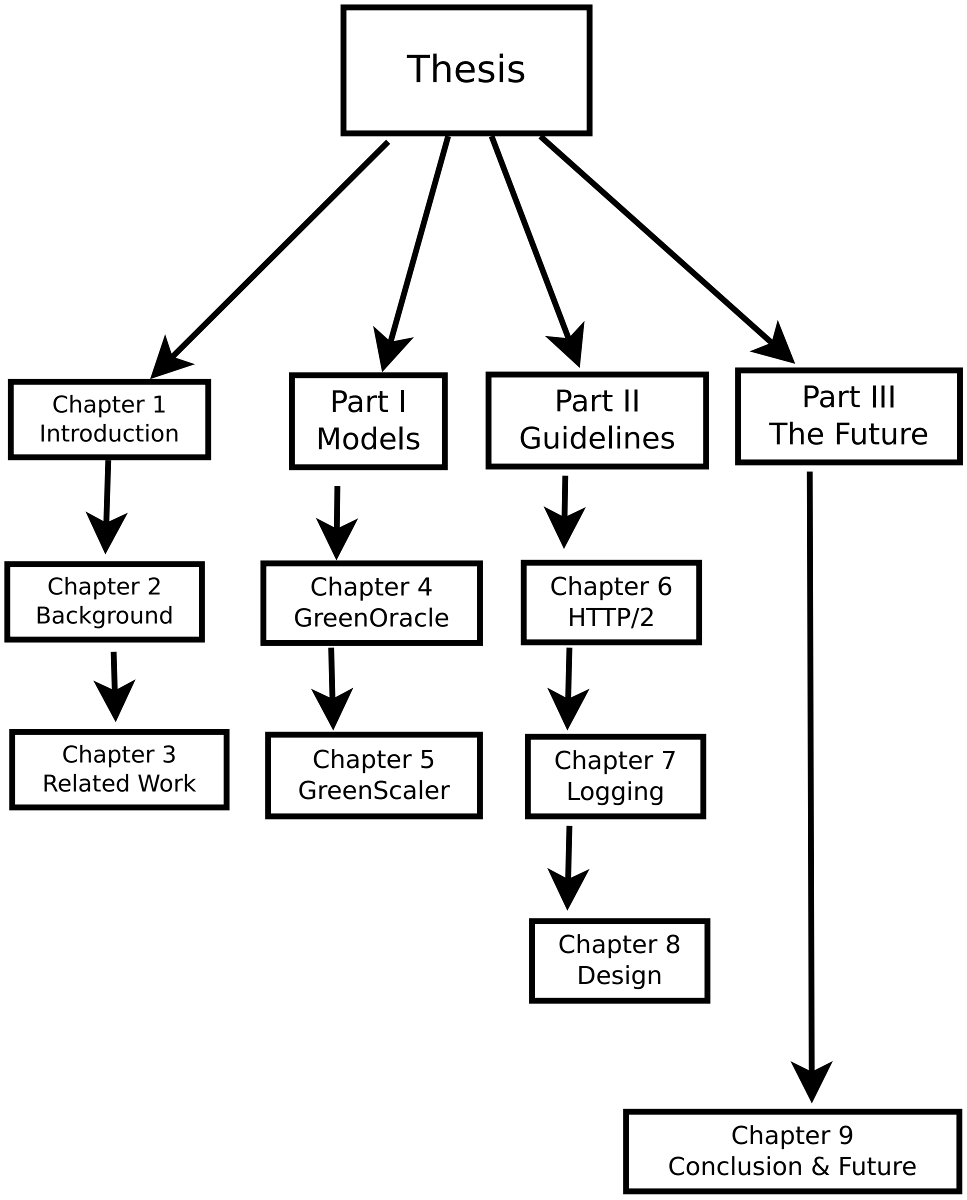


Figure 1.1: Thesis structure.

than `HTTP/1.1`. In Chapter 7, we discuss how developers can make software logging energy efficient. A new energy efficient design pattern is presented in Chapter 8.

Part III concludes this thesis along with a discussion of potential future work.

Chapter 2

Background

Important concepts that are used throughout this thesis are discussed in this chapter. First we describe how current and voltage are related to power, and why we focus more on energy than power. Then we explain the energy measurement system that was used for all the energy measurements used in this thesis. Next we explain a well-known energy bug, tail energy leak, because we refer to tail energy leak throughout this thesis. Finally, system calls are explained which are directly related to our energy modeling approaches.

2.1 Current, voltage, power, and energy

Electric current (I) is the rate of electron flow (or charged particles). The hardware components (in a mobile device, or in a desktop computer) operate using electric current. Consider the CPU as an example. A CPU consists of transistors, whereas a transistor is a semiconductor device that can amplify or switch an electrical signal (`On` and `Off`). With different configurations they form different logic gates that deal with boolean logic and can store binary information (`0s` and `1s`). Voltage (V), on the other hand, is the electric potential difference between two points in a circuit [107]. For example, the battery provides the electrical difference in a smartphone that induces electron flow. A smartphone becomes unavailable when the voltage is less than the required minimum voltage (low potential charge difference in the battery).

The one metric that captures both voltage and current is power ($P = V \times I$). Power (P) is the rate of work and is expressed in *watts*. Mobile devices run on batteries, and the availability of the devices depends on battery capacity (the amount of charge stored in a battery). More work indicates more power usage, which leads to reduced device availability. Also, more power usage (i.e., more electron flow) means more heat generation, which directly impacts the scale of data centers because of the cooling issues.

The power usage, however, is often not enough to understand how long a battery would last (for mobile devices) or how much heat (in a data center) would be generated. For that, the widely used metric is the energy consumption which not only includes the rate of work, but also considers the run-time of a system. Energy (E) is thus defined as the product of power and execution time (T) of a system ($E = P \times T$) and is expressed in *joules* [5], [49], [50]

2.2 Power vs. energy

In order to fully appreciate the difficulty of achieving energy efficiency, we need to understand the relation between power and energy (i.e., $E = P \times T$). As we can see, energy consumption does not necessarily decrease with the reduction in execution time. Reduction in the execution time T is beneficial only when the power (P) is constant or reduces as well. A reduced execution time, however, might put more workload on different hardware components (e.g., CPU). In order to cope with the workload, components like CPU might jump to more power consuming states and thus might make the whole system even less energy efficient. In fact, a previous study observed no correlation between execution time and energy consumption in Android apps [92]. This implies that software energy efficiency might not be achieved through conventional techniques. We need energy optimization guidelines, as well as techniques for measuring or estimating energy consumption so we know if a guideline indeed works.

2.3 Energy measurement: GreenMiner

In order to build machine learning-based energy estimation models (Chapter 4, and Chapter 5), we need actual energy measurements as the ground truths (dependent variable in supervised learning). Moreover, for developing energy efficient techniques, evaluation by measuring the energy consumption of those techniques are required. The *GreenMiner* energy measurement system was used for all the energy measurements that are used in this thesis. *GreenMiner* is fully described in Hindle et al. [102]. The following block quoted description of *GreenMiner* is directly used from my *GreenScaler* publication [45].

> *GreenMiner* provides accurate energy measurements for Android apps and is widely accepted in the software energy research community [4], [5], [48], [95], [101], [204]. The main components of this test-bed are a lab-bench power supply (a YiHua YH-305D), a test-runner computer (a Raspberry Pi model B computer) for controlling the experiments, an energy measurement IC (Adafruit INA219 breakout board), a micro-controller (Arduino Uno) for collecting energy measurements, and a system-under-test (a Galaxy Nexus phone) (Table 2.1). The Arduino and Raspberry Pi are powered by a USB hub. Each testbed costs approximately $250, each phone originally cost approximately $500, and the *Green-Miner* service is run on a seperate server ($1000). Development of the *GreenMiner* hardware and software itself was more than $32000 in developer time. The *GreenMiner* software is freely available for download [102].
>
> A test-runner, a Raspberry Pi, is connected to a particular system-under-test, a Galaxy Nexus. The test-runner pushes and runs tests on the Galaxy Nexus, and collects measurements from the Arduino. Afterwards the test-runner downloads statistics and other meta-data from the system-under-test. The responsible test-runner, a Raspberry Pi, then uploads the measurements to a central server

running the *GreenMiner* webservice. The current *GreenMiner* consists of four such identical testbeds to speedup and parallelize the data collection process. Figure 2.1 shows the innards of one of the four identical settings of the *GreenMiner*. The *GreenMiner* service is a continuous testing service whereby users may submit tests to be run and measured. After submitting a batch of tests to the *GreenMiner*, one of the phones is randomly selected for executing a test. As a result, four different tests can run in parallel to reduce the measurement time. *GreenMiner* maintains the same system state for each test by cleaning any installed apps that ran previously.

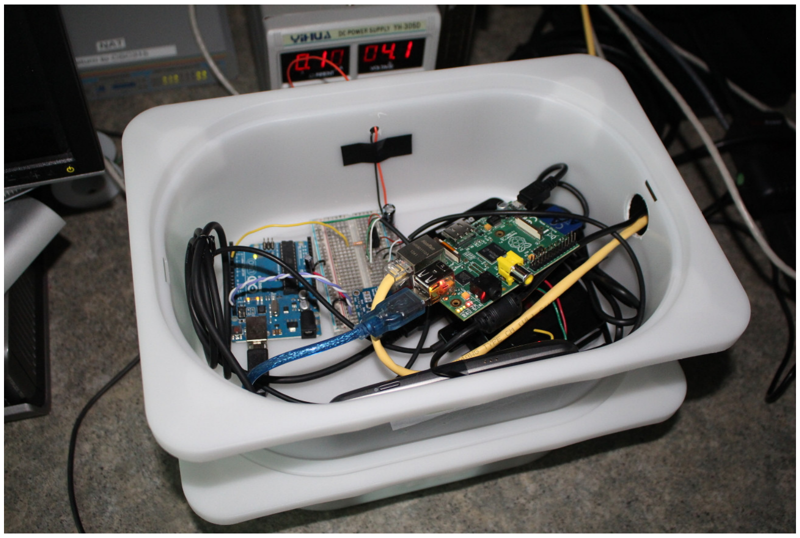


Figure 2.1: One of the four identical *GreenMiner* settings. Photo used with permission from the *GreenMiner* paper [102].

It is important to note that energy consumption varies (even if by a small amount) across different test runs. Consequently, all the previous *GreenMiner* based works reported the mean energy consumption of multiple test runs (when reporting the energy consumption of a test case) [5], [48], [95]. Similarly, all the reported energy measurements in this thesis are actually the mean energy consumption of multiple different runs (10 runs unless otherwise specified).

Table 2.1: Specs of the Samsung Galaxy Nexus phones used for the experiments.

Component	Specs
OS	Ice Cream Sandwich, 4.4.2
CPU	Dual-core 1.2 GHz Cortex-A9
GPU	PowerVR SGX540
Memory	16 GB, 1 GB RAM
Display	AMOLED, 4.65 inches
WLAN	Wi-Fi 802.11 a/b/g/n

2.4 Tail energy

Some components such as the NIC (Network Interface Card), and SD card suffer from tail energy. Tail energy is the energy that a hardware component consumes after finishing its task, because it stays in a high power state for sometimes before becoming inactive [5], [192], [194]. This is inefficient as the app consumes energy without doing any useful work in this period. One common approach for dealing with tail energy leaks is to perform operations in batches [154], [194]. The idea is that if multiple operations (related to a hardware component) are performed together in a batch, then the hardware component will have only one tail energy leak (at the end of completing all the operations in the batch).

To make it clearer, let us consider an example, where an app downloads 1000 images and then processes the images. The energy expensive way is to follow the download-process-download-process approach (download one image and process it immediately, before downloading and processing the next one). This is presented in Figure 2.2. In this approach, the network interface card (NIC) will be activated for downloading the first image. The app then will process the downloaded image, which can take few seconds. In the meantime, the NIC will become inactive with a transition from the active to inactive state. This transition causes a tail energy leak. This way, for 1000 images, the app will experience 1000 tail energy leaks. However, with the efficient way, presented in Figure 2.3, there will be only one tail energy leak. Because with just one NIC activation, the app will download all the images first, and then it will process all the images together in a batch. This simple solution was found to be significantly more energy efficient in real-world apps [192].

```
for i = 1 : 1000 {
  img = download (image_url[i])
  # A tail leak
  process (img)
}
```

Figure 2.2: A program that suffers from many tail energy leaks.

```
for i = 1 : 1000 {
  img[i] = download (image_url[i])
}

# A tail leak

for i = 1 : 1000 {
  process (img[i])
}
```

Figure 2.3: A program that suffers from one tail energy leak.

Tail energy complicates both energy measurements (if not careful) and energy modeling. For example, consider a test case `A` that we need to measure the energy consumption for. If the duration of the test case `A` is 20 seconds, an unsurprising mistake would be to measure the energy consumption for 20 seconds. But what if test case `A` was using a hardware component till the end, which has tail energy leaks? That tail energy leak will not be recorded in that 20 second period. This is why all the energy measurements that are used in this thesis had an extra 10 second period of waiting time so that the tail energy leaks were also captured (if there was any).

Tail energy also complicates software energy modeling, specially for utilization-based models (presented in Chapter 3) [194]. Utilization-based approaches only care about the utilization time of hardware components, which does not include the tail part.

2.5 System calls for software energy modeling

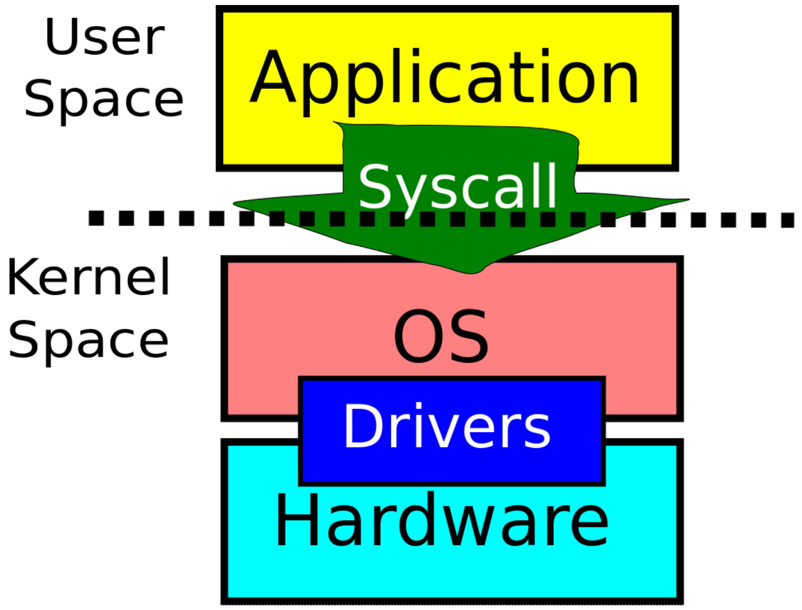


Figure 2.4: System calls act as the bridge between an application and the operating system.

For modeling software energy consumption, we need to capture the utilization patterns of different resources (a set of independent variables) that have impact on the energy consumption (the dependent variable). In this thesis, different system calls [230] (also known as monitor calls) are used as part of the independent variables for energy modeling (Chapter 4, and Chapter 5).

In a Unix-like system such as Android, a software or app is restricted to its own address space, and does not have direct access to hardware devices. It is only the operating system (OS) which has access to those devices and services. A system call is an interface (Figure 2.4) between the operating system (OS) and an app [4], [48], [194]. When an app requires a specific service (e.g., accessing the disk, creating a subprocess etc.), it calls a corresponding system call for that service. For example, in order to write to a file, an application uses the `write` system call. Similarly, the `recvfrom` system call is used to read messages from a socket. A system call causes an interrupt, which transfers the control to the kernel of the OS. The kernel then collects the system call's number and parameters from a predefined set of registers, and then executes the system call. After completing the execution of the invoked system call, the control is transferred back to the user space again.

System calls are related to the energy modeling Chapters (Chapter 4 and 5). Our hypothesis is that, if we can capture all the different system calls (and their counts) invoked by an application during a test run, we can estimate the resource utilization by that application. Given that resource utilization is directly related to energy consumption, the numbers of different system calls invoked by an application would therefore be a good estimation of the app's energy consumption. The `strace` [239] program was used to count the numbers of different system calls invoked by an app while executing a test case.

Chapter 3

Related Work

This chapter is divided into three sections in accordance with the motivation of this thesis: developers' knowledge on software energy consumption (to explain why energy efficiency research is important), previous energy estimation techniques and their drawbacks, and previous studies on energy optimization guidelines.

3.1 Developers' knowledge of software energy efficiency

As we have already discussed, achieving software energy efficiency is difficult. It is not just about reducing the run-time of a system. So it is important to understand what developers know and think about software energy efficiency.

Pang *et al.* [191] conducted an online survey among developers with a set of questions emphasizing on programmers' knowledge about energy efficient software development. The survey reveals that developers lack the knowledge on developing energy efficient software. It is not clear to them how to measure or estimate the energy consumption of their software—one of the motivations of this thesis. Programmers are also not sure what techniques are good or bad for software energy efficiency. The authors suggested that programmers, especially mobile application developers, should undergo proper training on developing energy efficient applications. Besides, automatic tools for suggesting power efficient features can be developed to help the programmers during software development.

A similar study by Manotas *et al.* [159] on a selected 464 practitioners from Google, Microsoft, ABB, and IBM revealed similar information—developers are uncertain about what improves software energy efficiency. This also implies the importance of accurate energy measurement or estimation. Practitioners are eager to learn different ways to improve energy efficiency (optimization guidelines). The authors concluded that new design approaches that support processing tasks in batches (to enable more idle time slots) would be helpful. This would also help making energy efficient software from the beginning, rather than struggling during the maintenance period. These observations motivated us to develop a new design pattern for energy efficient software—*GreenBundle*, as presented in Chapter 8.

Another similar study was conducted by Pinto *et el.* [198], but using energy consumption related questions from StackOverflow[1]. The authors observed a linear growth in energy related questions, revealing the growing interest in developing energy efficient systems. Compared to the other categories of StackOverflow questions, energy related questions received less number of successful answers—suggesting the inadequacy of energy experts in the community. Energy related questions, not surprisingly, were found to be closely related to mobile application development. Five different themes were identified by analyzing all the energy questions. These themes are *Measurements*, *General Knowledge*, *Code Design*, *Context Specific* and *Noise*. The *Measurements* related questions, for example, are questions asking how to measure energy consumption of different applications in different platforms. Supporting the earlier finding by Pang *et al.* [191] and Manotas *et al.* [159], *Measurements* category contained the least fraction of accepted answers.

Code Design related questions deal with programmers' decisions towards producing energy efficient applications. According to the StackOverflow users, the main causes for software energy consumption are unnecessary resource

[1]http://stackoverflow.com/

usage, faulty GPS behavior, background activities, excessive synchronization, background wallpapers, and advertisement. Although some of the understandings are true—at least in some contexts—the authors found some misconceptions among StackOverflow users regarding code design to achieve energy efficiency. For example, the idea of racing to idle is the assumption that a faster program consumes less energy than a slower program. This assumption, however, is flawed as different hardware components can have different states of operation. A CPU can operate in higher frequency (for heavier workload with reduced run-time), thus can consume even more energy.

Evidently, the research community needs to produce easy (and accurate) energy estimation approaches, and needs to provide guidelines for producing energy efficient software.

3.2 Modeling energy consumption

Previous software energy models can be categorized into three broad categories: instruction-based, utilization-based, and system call-based energy modeling.

3.2.1 Instruction-based modeling

Instruction-based modeling is estimating energy consumption using program instruction cost along with program analysis techniques.

In order to estimate software energy consumption using program instruction cost, Shuai *et al.* proposed *eLens* [92]—a tool that can estimate energy profiles at the instruction level, method level, and thus can estimate the energy consumption of the whole software system. *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 to create a new instrumented version of the software artifact and can generate sets of paths in the app 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. *eLens*, however, requires per-instruction power profile, which is not always available.

In addition to the requirement of per-instruction power profile, another major disadvantage of instruction-based modeling is its rigidness to one particular programming language. This approach also requires the availability of source code of the app under test. In contrast, the *GreenOracle* and the *GreenScaler* models do not require app source code or per-instruction power profile, making the models not only easier to use, but also more generalizable.

3.2.2 Utilization-based modeling

Utilization-based power models take into consideration the utilization statistics of individual components of a system like CPU, screen brightness, Wi-Fi etc. Regression analysis is conducted, to model app's energy consumption, by using the utilization statistics and the corresponding energy consumption.

Carrol *et al.* [40] studied the energy consumption of the Openmoko Neo Freerunner, an Android smartphone. They analyzed the energy usage patterns of different hardware components: CPU, memory, screen, graphics hardware, audio, storage, and different networking interfaces. After observing and capturing energy usages 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. The authors, unfortunately, do not offer any tool for the developers.

Shye *et al.* [227] developed a regression-based energy model. A logger app was employed that logs system performance metrics and user activities. The basic idea is to model the relationship between the captured system statistics and the energy consumption. The authors found that screen and CPU power consumption contribute highly toward the total energy drain. A similar utilization based energy model was proposed by Gurumurthi *et al.* [90]—using components' utilization statistics, their proposed model provided some useful insights into the major sources of power drains: disk is the largest consumer of energy (consuming approximately 34% of the system energy). This model, however, was not tested for mobile devices. Utilization based energy models were also studied by Flinn *et al.* [71], Zhang *et al.* [269], and Dong *et al.* [62].

Utilization based models, however, suffer from tail energy phenomenon [194]—modeling energy based on the active utilization time of hardware components does not consider the tail energy parts. The *GreenOracle* and *GreenScaler* models, on the other hand, do not model energy directly based on the active utilization times of hardware components, but based on simple OS-level statistics (number of CPU jiffies, number of system calls and so on). "This automatically alleviated the intricacy of separately modeling tail energy for every hardware components" [45].

3.2.3 System call-based modeling

System call-based modeling, in contrast to the direct utilization-based modeling, uses traces of system calls to understand the types and amounts of accessed resources by an app. A system call based approach is able to overcome the shortcomings of the utilization based approaches for several reasons [194]. Firstly, systems calls are the only gateways to provide access to different I/O components—capturing all the system calls invoked by an app thus provides the list of I/O components accessed by an app. Secondly, this approach of energy modeling does not suffer from the tail energy phenomenon.

Pathak *et al.* [194] proposed a complex Finite State Machine (FSM) based model using system call traces. Each state in the FSM represents a power state of a specific component or a set of components. A transition represents a significant change in the workload that causes the component to switch to a new power state (e.g., active state to tail power state, tail power state to base state and so on). A fully functional FSM for a specific smart phone can be designed by capturing the timing information in different states (e.g., how long a component stay in tail power state) and their corresponding energy measurements for all the hardware components. Aggarwal *et al.* [4], [5] 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. The authors, with high accuracy, proposed a rule of thumb that can be used by developers to predict if a new version is more energy efficient than the previous one: a significant change in the numbers of system calls invoked by an application implies a significant change in the energy consumption. This model, however, does not offer the actual energy consumption, and thus the developers would not know how bad the energy change is. None of these models consider screen colour and may profile other components inaccurately. The number of apps used for learning and validation was also very small compared to the proposed dataset.

3.2.4 Other models

PETrA is a recent energy estimation tool (published after *GreenOracle* and before *GreenScaler*). The following discussion about PETrA is used from our *GreenScaler* paper [45].

> Nucci *et al.* [184] proposed PETrA, an energy estimation tool that leverages Android tools such as `dmtracedump`. As PETrA is the state-of-the-art for estimating energy consumption of Android systems, we wanted to compare *GreenScaler*'s accuracy with PETrA. Unfortunately, PETrA relies on measurements that are not supported by all Android devices. For example, the `batterystats` program to collect which components were active during an app run, is not supported by the version of Android running on the *GreenMiner*'s Galaxy Nexus phones. In contrast to PETrA, *GreenScaler* is already open source and relies on information that is available on any Linux-based system.

Previous energy models do not generalize across different platforms and operating systems. Also, some of the models are hard to be actually used by the developers. We need an energy estimation model with tool support that is reproducible and easy to use.

3.3 Energy optimization and testing

Smartphone users value longer battery life than other non-functional features, and the research community is trying to help the users by providing recommendation systems while selecting an app from a group of similar apps [209]. With such recommendation systems being developed, developers are now more compelled for producing energy efficient systems. Significant research has been done on software energy efficiency and energy testing for finding or resolving energy bugs. This section focuses on different areas of software energy efficiency.

3.3.1 Energy efficient color selection

Display of a smartphone is one of the most energy consuming components [40]. Organic light-emitting diode (OLED), used by many modern smartphones, is most energy efficient in dark color and expensive in light color mode. This is a concern for energy efficiency as light colored background is adopted by many popular web applications. An automated system to transform a web application to an energy efficient version (by changing its color) would allow the users to use the original or the energy optimized app.

Li *et al.* [142] proposed automatic color transformation (energy inefficient to energy efficient) for web apps. The objective is to convert web apps' colors for making them energy efficient, but without impacting the readability. The proposed approach is based on several graphs containing useful information that help in converting the colors. For example, the adjacency relationship graph is formed for showing siblings and child-parent relationship between nodes. Then there are different types of color conflict graphs: the background color conflict graph, text color conflict graph, and image color conflict graph.

These graphs help rank the importance of color difference between elements. In the ranking system, the color difference between a child and the parent element is more important than the color difference among the siblings (for maintaining the readability of the web apps).

3.3.2 Cloud computing for saving mobile energy

A study by Othman *et al.* [188] claimed that up to 20% reduction in energy consumption is achievable by offloading tasks from a local mobile device to a fixed server. This approach, however, is not helpful when offloading a task is more expensive than processing the task locally. Moreover, asymmetric communication—more receives than sends—is needed to achieve such energy efficiency, because transmitting is more energy expensive than receiving. The main challenge is to know the energy profiles of a task for both local and remote executions. Also, for some applications the response time might be too crucial to take the advantage of a remote server (due to the communication time between the client and the server). The authors used history data for their simulation—previous energy usage by the task when ran locally. If the estimated cost of communication is less than the history cost, the device would go forward for data offloading. Results suggest that slower mobiles with more bandwidth benefit more from such offloading.

A similar study by Miettinen *et al.* [169] found that most of the mobile applications are unfortunately suitable for local processing. Mobile cloud computing can help explore new computationally expensive but useful mobile applications, only if offloading the task to the cloud offers less energy drain than processing the same task locally. Based on the authors recommendations, task offloading should be based on workload characteristics, and on the underlying communication technologies. Also, bulk data transfers (batch processing) are helpful for saving energy consumption.

3.3.3 Mobile energy efficiency in video streaming

Video streaming in smartphones has become energy expensive [74]. For video streaming, Trestian *et al.* [245] examined the impact of different network related aspects on mobile device's energy consumption. An Android device was used under an IEEE 802.11g network for profiling energy consumption under different scenarios. In general, the authors examined the impact of several factors in video streaming on mobile energy efficiency: video quality, selection of TCP or UDP as the transport layer protocol, link quality, and so on. For the experimentation, local video playbacks were performed for a range of video qualities (i.e., bad to excellent). Selecting the lowest quality video makes battery life double than selecting the highest quality video. In fact, 34% energy consumption was saved just by switching from excellent quality video to good quality video. This is encouraging as the surveyed users noticed negligible difference between these two quality levels. Surprisingly, TCP based streaming consumed less energy than UDP based streaming, in spite of the extra features that TCP maintains for providing reliable data transfer. This was due to the less network load with TCP (with congestion control) than the UDP based systems.

A similar study by Gautam *et al.* [74] suggests that applying algorithmic pre-fetching helps saving energy. This is, however, based on the assumption that the algorithm is accurate in selecting appropriate videos based on a user's predilection. An inaccurate algorithm might backfire by downloading lots of videos that the user might never watch.

Mohammad *et al.* [106] studied how intelligent video buffering techniques can be employed to significantly improve energy usage of mobile devices. The author reported four main sources of energy consumption in pre-fetching/buffering video segments: 1) downloading segments at lower rate than the possible maximum rate—the time intervals between subsequent packets causes energy drains, 2) persistent TCP connections—because they keep client's radio awake, 3) tail energy—time gap between downloading subsequent small sized segments leads to more tail sections, and 4) buffering unneeded segments—segments that users do not watch most often. The authors proposed a method for efficient video streaming based on crowd sourced viewing statistics. The assumption is that a video's previous audience retention information provides useful insights about the future users watching patterns of the same video. This approach, based on the scenarios, offers up to 80% energy savings compared to the default Android YouTube application.

3.3.4 Does ad blocking help to reduce energy drain?

Study reveals that 65%-75% of energy usage in mobile applications can only be from the third-party advertisement functionalities [192]. Mohan *et al.* [174] observed that advertising can significantly degrade energy efficiency—consuming 65% of the communication energy and 23% of the total application's energy. This is because of the regular refreshes that causes the network radio to become active. These observations raise an interesting question: can we reduce energy drains of mobile devices by adopting ad-blocking techniques? Or the overhead for ad-blockers dwindles the benefit so much so that the device consumes even more energy? In desktop settings with the Windows operating system, Simons *et al.* [231] observed 3.4% energy savings with an ad-blocker.

A more detailed study was conducted by Rasmussen *et al.* [204], using different hosts files and AdBlock Plus. A hosts file contains (located at /etc/hosts in many Linux systems) a list of IP addresses and aliases for each IP address so that a request for a web server first go to this file instead of going to the DNS. This way any unwanted web pages (i.e., advertisements) can be mapped to an invalid address—thus preventing those pages from loading. Energy efficiency of ad blocking was evaluated with different settings, with a set of 100 popular websites. The observation was not conclusive: ad blockers saved energy for some settings only. The authors suggested further optimization by removing entries (i.e., IP addresses from host files) that are not usually accessed, and by sorting entries based on the frequency of requests.

3.3.5 Impact of code obfuscation and refactoring on energy

Piracy has been reported as a crucial problem for Android apps [213]. Code obfuscation is the process of making source code less human readable for inhibiting piracy, which is widely adopted by the developers (specially by the mobile app developers). Code refactoring, on the other hand, has completely different objective—making source code more readable and maintainable. Developers apply code factoring to help their team members, while code obfuscation is applied for the outside world. As both of these approaches are widely adopted by the app developers, it is important to understand their energy impact so the developers can make informed decisions. Sahin *et al.* studied the impact of code obfuscation [213] and refactoring [212] on energy consumption of several Android apps. They found that code obfuscation does impact energy consumption, but the differences could be too small for the users to notice. This suggests that developers do not need to worry much about applying code obfuscation—thus can prevent piracy without significantly impacting apps' energy consumption. For the refactoring study, the authors examined different refactoring options: extracting methods, converting local variables to fields, and so on. The impact of code refactoring is not consistent across different applications (refactoring can increase or decrease the energy consumption).

3.3.6 Energy efficiency of Java collections

Some studies concentrated on writing energy efficient code during the development phase. Along that direction, energy profiles of the frequently used Java collection frameworks were studied [95], [196]. Hasan *et al.* [95] proposed a recommendation system for selecting the most energy efficient Java collection classes for different scenarios in Android (for example, is it an insert to a list? if yes, does the insert take place at the beginning or in the middle of a list? and how many elements are involved?). The authors profiled energy consumption of different classes from three different types: List (e.g., ArrayList, and LinkedList); Map (e.g., HashMap, and TreeMap); and Set (HashSet, TreeSet, and LinkedHashSet). This recommendation system enables developers to select the most energy efficient Java collection for a given scenario. The authors found that even up to 300% energy can be saved by replacing an energy inefficient collection by the most efficient one. In a similar vein, Pereira *et al.* [196] measured the energy consumption for different Java collections, but they did it for Linux server instead of Android phones. Manotas *et al.* [160] proposed the SEEDS framework that can automatically select and replace an energy inefficient Java collection by an energy efficient collection.

3.3.7 Detecting energy bugs and hotspots

An energy bug—in contrast to the traditional programming bugs that produce incorrect/unwanted results—leads to reduced battery life [268]. The sources of energy bugs can be of different types: *no sleep or wake lock bug*—e.g., waking up the CPU, but not putting it back to the sleep mode; *loop bug*—waiting for an event to happen, periodically inspecting changes in a variable, and thus unnecessarily using CPU cycles. An energy hotspot, however, points to a segment of an application where the energy consumption is significantly higher than other segments [28]. While an energy hotspot can be normal (and unavoidable) because of an app's functionality, detecting hotspot might enable a developer to think about possible optimizations.

In order to detect energy bugs and hotspots in Android applications, a test generation framework was offered by Banerjee *et al.* [28]. As I/O components are accessed by invoking system calls, capturing those system calls can help to find I/O related energy bugs or hotspots in a particular application. The authors emphasized on the careful test case generations so that most of the expensive system calls are traced. The framework is summarized in three sequential steps: 1) for each application, an Event Flow Graph (EFG) [166] is formed so that each EFG can capture the set of possible user interaction sequences in a particular application; 2) by following the EFG, some event traces are generated randomly, and the associated system calls are also captured; 3) the same process of event trace generation is repeated until most of the important system calls are captured or the time budget expires. In order to detect hotspots/bugs, energy consumption to Utilization ratio (E/U) was calculated. A particular application is defined to have energy bugs if the E/U after running a test case is higher than the E/U before running the test case. Energy hotspots, however, are captured using anomaly detection in time series data. The authors then manually evaluated the hotspots and bugs suggested by their framework and found very few false positives.

Wake locks are frequently used in Android apps for critical operations (e.g., banking transactions) so that the involved hardware components (e.g., Wi-Fi) do not go to the sleeping state due to the Android's aggressive power saving policy [149]. Pathak *et al.* [193] reported wake lock related bugs as the most common energy bug. Liu *et al.* [149] found that around 60% of the Android apps that use wake locks, suffer from wake lock related bugs. This is why developers should be extra careful while using wake locks. Liu *et al.* [149] provided three simple guidelines for using wake locks: 1) is it worth using a wake lock (for a scenario) considering the loss in energy efficiency? 2) which component actually need to be awake? and 3) what are the program points that the wack locks are most suitable to be acquired and released? The authors also discovered eight common pattern of wack lock bugs. Interestingly, the most dominant pattern is the *unnecessary wakeup* pattern. This means that developers are not sure at what points the program should acquire and release a lock. They either acquire a lock too early or release a lock too late. The same problem was observed by Alam *et al.* [10], who then proposed an energy efficient data flow analysis based automatic approach for appropriately placing wack lock acquire and release related code. To help developers with the wake lock related problems, Wang *et al.* [249] proposed a tool called *WLCleaner* that leverages the *dumpsys* tool to dump power consumption related information at runtime. This information was then used for locating wake lock related bugs.

3.3.8 Guidelines for energy-aware developers

Linares-Vásquez *et al.* [144] analyzed 55 Android apps to capture energy greedy APIs in Android. While all the energy greedy APIs with their energy profiles were listed (limited by the 55 apps), the authors found that almost 60% of the energy hungry APIs belong to the GUI & Image manipulation, and the database categories. The authors produced a set of actionable guidelines for the energy-aware app developers. For example, the Model-View-Controller pattern is discouraged for apps with many views. Also, information hiding is found energy expensive; developers are recommended to have public fields, instead of private, if the fields are accessed frequently. This will reduce the number of calls to the *getter* and *setter* methods and will reduce app's energy consumption.

Banerjee *et al.*, in an invited talk, discussed some interesting issues about energy efficiency [29]. For small data transmission for example, GSM is more energy efficient than Wi-Fi (due to the connection initialization cost). But for larger transmissions Wi-Fi gets better than GSM. The authors also discussed the trade-off between QoS and energy efficiency of mobile applications. For instance, the users can be provided with the option to reduce the GPS update frequency when the battery power becomes low. These types of design choices depend on the application itself, and the developers must decide on such design issues before start developing the application. The main idea of such application design is to maintain standard QoS while minimizing the energy consumption as much as possible. The authors also gave special importance on using asynchronous task provided by Android. By default, all the components of Android applications run in a single thread (the UI thread). This leads to performance degradation as well as energy inefficiencies. For example, if a network operation is delayed, all the subsequent operations (if under the same thread) are also delayed, which indirectly affects the energy efficiency of that application.

A more detailed road-map for developing energy efficient mobile applications was given by *Ding et al.* [138]. Some of the selected guidelines are presented as follows.

1. In approximately 50% of the applications, the actual application code was responsible for less than 31.1% of the total application energy consumption. This suggests that most applications spend most of the energy in idle states. Efficient color scheme can be employed to improve the energy expenses in idle states [142].

2. Application developers should be more concerned on the optimization of using APIs rather than their own code—system APIs consume significantly much more energy than the user code. This complements the findings of Linares-Vásquez *et al.* [144].

3. In case of using hardware components, the developers should concentrate more on optimizing network operations (e.g., performing `HTTP` requests). Network operations are extremely energy expensive, yet frequently found in Android apps.

4. Instruction inside loops are not only frequent in all types of applications, but also can be very expensive. Optimization techniques need to be applied for such cases.

3.3.9 Design patterns and energy consumption

Are there any specific design patterns that can make mobile application energy efficient? Like energy specific testing, this is another least explored area in software energy research. Cruz *et al.* [57] leveraged mining software repositories (by analyzing commits, issues, and pull requests) to understand what common patterns are followed by energy-aware Android and iOS app developers. They made a catalog of 22 common practices that the developers follow. Interestingly, some of these revealed patterns complement earlier findings from the research community. For example, making apps with dark backgrounds [45], [142], and processing tasks in batches [84], [154], [192]. This type of catalog is definitely helpful for the energy-aware developers. However, some patterns practiced by the developers are controversial. For example, the *race to idle* pattern. While racing to idle can help in some cases, it can also increase the energy consumption if the corresponding hardware components need to wake up frequently. It is best for the developers to actually measure or estimate the energy consumption of their apps for such optimizations—optimizations that can both improve or worsen the energy efficiency of an app.

Sahin *et al.* [210] studied the energy consumption of 15 existing design patterns from the 3 broad categories: creational (e.g., buidler, singleton), structural (e.g., composite, decorator), and behavioral (e.g., observer, visitor) design patterns. The authors found that some of the design patterns improved the energy efficiency whereas some others did the opposite. There was, however, no solid recommendation from the study (as it was a preliminary study on design patterns). For example, the observations could be highly impacted by the choice of the subject apps. As a result, developers do not have enough guidelines about what kind of patterns are energy efficient for what kind of scenarios.

3.3.10 Batch processing for energy efficiency

Tail energy leak (discussed in 2.4) is one of the most discussed topics in the software energy research community [50], [57], [117], [192], [194]. The common solution to alleviate tail energy leaks is to perform operations in batches (e.g., bundled I/O operations) [57], [192].

Pathak *et al.* [192] observed that a real-world photo uploading app was consuming too much energy because of the tail energy consumption of the network component. The app was uploading photos, but not in bundle. After uploading a photo, the app computes the hash of the next photo and then uploads it. This means that after uploading each photo, the network component goes to the tail state, and wait for the next photo to be ready. So if there are 100 photos to upload, there will be 100 tail energy phases for the network component. This serious tail energy leak can be prevented if the processing (calculating hash) of all the photos are done first and then the app uploads all of the photos in a batch.

Because of the overhead associated with each request, sending the same data in a single bundled larger `HTTP` request is much more energy efficient than sending the same data over many smaller `HTTP` requests. To take the advantage from this observation, Li *et al.* [141] proposed an automated bundling approach that can decide at runtime which `HTTP` requests are safe to bundle (using program analysis). With this approach, the authors were able to reduce energy consumption for real-world apps by around 15%.

Similar to network operations, bundling has been found helpful for other I/O operations. Lyu *et al.* [154] has shown that grouping multiple database auto-commit transactions into a single transaction improves energy efficiency. In general, their proposed approach can automatically detect database write operations that occur repeatedly (e.g., inside a loop) so that it can bundle them into a single auto-commit. While evaluating on marketplace apps, significant energy consumption was reduced with this approach.

3.3.11 Energy specific testing

In order to automatically test an app, a set of test scripts that run and test the app are required. A test script represents a test case of an app. For example, for a gaming app, a test case can be a script that opens the app, shows the playing options, select the difficulty level, and starts the game. Developers often manually write test scripts for testing their software or apps. However, writing test scripts manually is expensive. This is why software testing community has been heavily working on automated test case generation [15], [156], [157], [161], [172]. Energy specific testing, however, is different than traditional testing [117]. For example, in spite of its wide acceptance for test generations and test selections in traditional software testing [16], [17], [37], [82], [180], the code coverage heuristic does not perform well in energy specific testing [45], [117]. Covering more code does not mean covering energy expensive parts of an app. Unfortunately, energy specific testing research has not seen much progress, and this is one of the energy research areas that the community needs to focus on.

Jabbarvand *et al.*[117] proposed an energy-aware test-suite minimization approach. The idea is that running only the test cases that cover energy hungry APIs are enough. Other test cases that do not execute energy expensive paths are unnecessary for energy specific testing. The authors found that a simple greedy algorithm performs really well in minimizing the test suites (around 80% test cases can be dropped). The list of energy greedy APIs reported by Linares-Vásquez *et al.* [144] was considered for ranking the test cases. This is a limitation, because the API list of Linares-Vásquez *et al.* [144] was formed only using 55 Android apps, thus many other energy expensive APIs might be ignored. Subsequently, test cases that are energy expensive but do not cover APIs from that list will not be in the minimized test suite. This approach also rely on an existing test suite. The quality of the minimized test suite totally depends on the existing suite (i.e., master test suite).

In a separate work, Jabbarvand *et al.*[117] proposed a framework that can evaluate the effectiveness of a test suite in revealing energy bugs. The general idea is to apply mutation in source code (for introducing energy bugs), and then evaluate a given test suite to see if the bugs are revealed. The authors concluded that existing test generation techniques are not good enough for finding most of the energy related bugs, so more attention should be given on energy-specific test case generation. In that direction, the authors proposed a search-based energy testing approach for Android [115], which utilizes an evolutionary algorithm to generate a test suite emphasizing more on finding energy bugs. The authors could not compare the effectiveness of their approach with a tool that also specializes in energy testing. This implies the importance of more energy testing research.

From previous works, we still do not know which Internet protocol is the most energy efficient (presented in Chapter 6). We do not know, how developers' logging practices impact software energy consumption (presented in Chapter 7). Most importantly, there is no study that investigates how developers should aim for energy efficiency from the design time, before they start the implementation phase (presented in Chapter 8).

Part I

Energy Estimation with Models

Chapter 4

GreenOracle: Producing Reproducible Energy Models

This chapter shows the initial step for achieving the first objective, discussed in 1.3.1, of this thesis: building energy models that are reproducible for other linux-based systems (specially for Android devices). The idea is that if we can build accurate software energy models using a feature set that is available across different platforms and operating systems, we can follow the same methodology for reproducing the models in other systems.

This chapter was published as:

- Shaiful Alam Chowdhury, Abram Hindle, "GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora", In 13^{th} International Conference on Mining Software Repositories (MSR 2016), pages 49-60. May 14-15, 2016. Austin, Texas [48].

GreenOracle is a count-based energy estimation model, which is built using the Samsung Galaxy Nexus Android phone. This model uses different OS statistics (e.g., number of CPU jiffies, number of context switches), and individual counts of different system calls (described in Section 2.5) as the independent variables. Energy measurement from the *GreenMiner* (discussed in Section 2.3) is used as the dependent variable. *GreenOracle* is trained on 24 Android apps. However, the training set was enlarged by adding a total of 984 versions from those 24 apps.

Why is GreenOracle reproducible? The independent variables of *GreenOracle* can be accessed from any Linux-based system. For example, for tracing the OS statistics, the Linux */proc* pseudo-file system (*/proc/pid/stat* [241], and */proc/stat* [1]) was used. For capturing the system calls invoked by an application, the `strace` program was used. The `strace` program can be run similarly in other Linux systems.

The takeaways from this chapter include:

- By leveraging techniques from mining software repositories, we can collect a significant number of third party apps and their versions for building machine learning-based software energy estimation models.
- Machine learning-based software energy models, if built on a feature set that is available across different platforms and operating systems, are reproducible.
- The *GreenOracle* model shows that the estimation accuracy of machine learning-based models improves with the addition of new apps in the training set. The problem is, in order to add a new app we need to manually write a new test script for the app (difficult). This encourages new research ideas for adding new apps into energy estimation model training without much manual effort (e.g., by leveraging automated test generation).

My role in GreenOracle: I, with the help of my supervisor, made plans for the methodologies, data collection, experimentation, and evaluations. I also wrote the *GreenOracle* paper [48] with the guidance of my supervisor.

Impact: As of writing, *GreenOracle* has been cited 26 times according to the Google Scholar. The citations include different contexts, such as successful use of system calls for modeling energy [197], and applications of mining software repositories in energy research [118]. This chapter provides energy consumption of 984 versions from 24 Android apps with human written meaningful test cases. This dataset was directly used by two different research [46],

[207].

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.

4.1 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 [199]. According to eMarketer, the number of smart-phone users will exceed two billion by 2016 [63]. This enormous adoption led to a significant increase in mobile data traffic, and is predicted to increase 10-fold between 2014 and 2019 [54]. 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 [101]; most software developers are not aware of how their source code changes might have drastic repercussions on their application's energy consumption [191].

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 [177]. 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 [191]. Among different energy related topics such as energy optimization, software developers mostly suffer from the measurement related issues [198]. Yet the number of models and tools to estimate software energy consumption and to locate energy bugs are noticeably low [4], [191]. Some of those models [92] 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 [5], [49], [194] 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 *GreenMiner* [101].

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 instrumentation. 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 4.6.

4.2 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.

4.2.1 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* [5]. 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 [50].

4.2.2 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 [5], [194]. 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.

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

4.2.3 Modeling energy consumption

Energy modeling based on component's utilization

Carrol *et al.* [40] 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.* [227] 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.* [90], disk was found to be the largest consumer of energy. Utilization based energy models were also studied by Flinn *et al.* [71], Zhang *et al.* [269], and Dong *et al.* [62].

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 [194]—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.

Instruction based modeling

In order to estimate software energy consumption using program instruction cost, Shuai *et al.* proposed *eLens* [92]. *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. [219] 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.

Energy modeling from system call traces

Pathak *et al.* [194] 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.* [5] 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 [49]. 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.

4.2.4 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 smartphones' energy drains [28]. This is partly because of the tail energy leaks that are common to exist when energy bugs are not considered carefully [192], [194]. In order to reduce the tail energy leak, bundling I/O operations together has been suggested [192].

For suitable jobs—when offloading data itself is not very expensive—transferring task to fixed servers can be compelling towards saving energy [188]. Unfortunately, a separate study revealed that for most of the mobile applications data offloading is too expensive to offer any gain in energy saving [169].

Automatic color transformation was offered by Li *et al.* [142] 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.* [74]. Rasmussen *et al.* [204] observed that ad-blockers, in spite of their own energy consumption, can help in reducing energy drains.

4.3 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 4.3.1). 2) Energy measurements along with the resource usage patterns were captured for all of the versions (Section 4.3.2, 4.3.3, and 4.3.4). 3) A grouping mechanism was used in order to deal with some system calls that have different names but similar characteristics (Section 4.3.5). 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 4.3.6). Finally, models are developed using machine learning algorithms for regression (Section 4.3.7) and validated using a separate cross validation set (Section 4.3.8). 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.

4.3.1 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 4.1).

F-Droid [68], 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 4.1 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.

Table 4.1: Description of the applications

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 [70]
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 [51]
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 [248]
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 [59]

4.3.2 GreenMiner

GreenMiner [102], a hardware-based energy profiler, was used for energy and resource profiling. *GreenMiner* 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 *GreenMiner* are available in prior work [102], [204].

4.3.3 Developing the test scripts

A separate test script was developed for each of the applications 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 the rudimentary Unix shell available on Android [102].

4.3.4 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. *GreenMiner*, 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 different runs of the same test case [5], [49]. 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* [241], and information global to the system behavior was captured from */proc/stat* [1]. 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 *GreenMiner*, 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 [86].

4.3.5 Grouping system calls

We observed that in spite of their very similar characteristics, some system calls come with different names [49]. 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 calls *lseek* whereas another one calls *_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 [147]. System calls that are

Table 4.2: Grouping similar system calls according to OS semantics

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"

unique in their functionality were not grouped with others. All the grouped system calls are presented in Table 4.2. 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.

4.3.6 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 4.1 was used for our feature normalization [11], where **x** is a feature vector.

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})} \tag{4.1}$$

After applying grouping, the dataset has 98 features from system call traces along with the 21 features that we got from the *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 contribute to the actual energy consumption. Some feature selection algorithms like forward and backward selection suffer from the local optimization problem—decisions can not be altered once a feature is selected or dropped [130]. 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 [270]. 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 4.3 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.

4.3.7 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.

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

Table 4.3: Selected features from the traces of system calls and the CPU related information

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

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 4.2 [96]:

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}} [\sum_{i=1}^{m} (Y^i - \sum_{j=0}^{n} \theta_j X_j^i)^2 + \lambda \sum_{j=1}^{n} \theta_j^2] \qquad (4.2)$$

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.

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 4.2; lasso uses l_1 (i.e., $\sum |\theta_j|$) penalty instead of l_2 (i.e., $\sum \theta_j^2$) [96].

Support vector regression

Unlike the other algorithms that we implemented with Octave, we used the SVM^{light} [125] implementation with linear kernel for the SV regression. SVM^{light} is implemented based on ϵ-SV regression [247] 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 [98].

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.

4.3.8 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 4.2, 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. A similar process was followed for all other applications and algorithms.

4.4 Experiment and result analysis

Table 4.4 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,

Table 4.4: 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.

	% of prediction error (measured and predicted in *joules*)				
Applications	**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

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 4.1 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

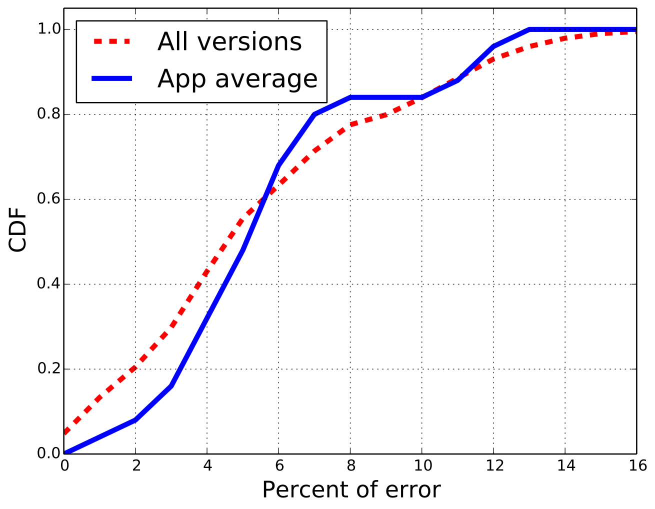


Figure 4.1: Percent of error with ridge regression

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* [92] and Pathak's FSM based model [194] 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 regres-

Table 4.5: 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.

	Weight			
Features	**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

sion based energy model (*GreenOracle*) as presented in Table 4.5 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 the 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 $\approx$ 0.7), but none of them were deleted from the set even 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.

4.5 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.

4.5.1 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.* [266] 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.* [165] 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 4.2 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.

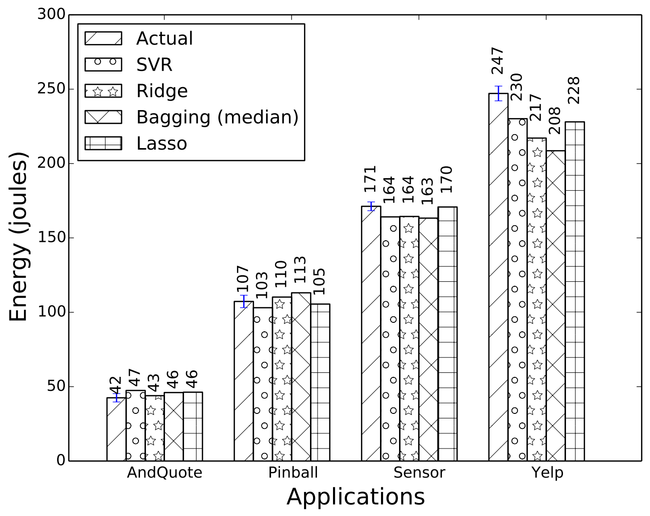
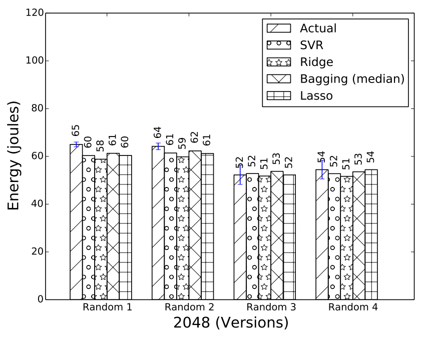


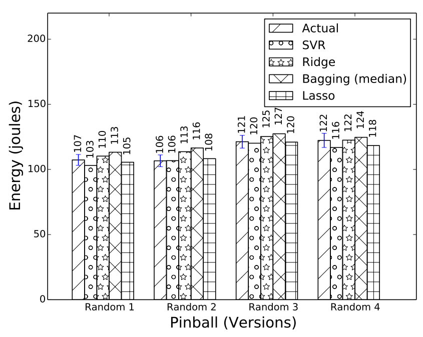
Figure 4.2: Models accuracy in segregating applications with very different energy requirements. Our proposed energy prediction approach is promising to enable energy-rated applications.

4.5.2 Identifying energy sensitive code changes between subsequent versions

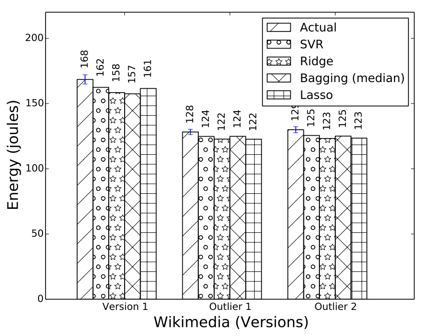
In continuous developments, the developers produce subsequent versions of the same application. In terms of energy efficiency, a simple code change can be colossal—both positively and negatively [101], [192]. The developers should be able to know if the changes committed for the new versions are going 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 the previous



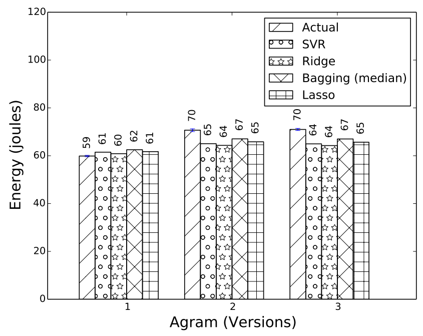
(a) 2048 Android puzzle game



(b) Pinball Android game



(c) Wikimedia (Wikipedia for mobile)



(d) Agram (generates anagrams)

Figure 4.3: Models' efficiency in differentiating versions with different energy consumption

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 4.3 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 4.3 (a) clearly shows the accuracy of the estimates produced by all of our models. Similar observations can be made from Figure 4.3 (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 4.3 (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 4.3 (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?

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 4.5 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 the *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 4.4 depicts such a synchronized method that returns the list of ana-

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

Figure 4.4: Agram synchronized method example

grams 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 [198], [236]. One commenter stated that lock requires more system calls and context switches that induce performance degradation [236]. 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.

4.6 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 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 4.5; and 5) use the ridge regression coefficients (i.e., *GreenOracle*) from Table 4.5 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.

4.7 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 4.4). 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 4.5 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.

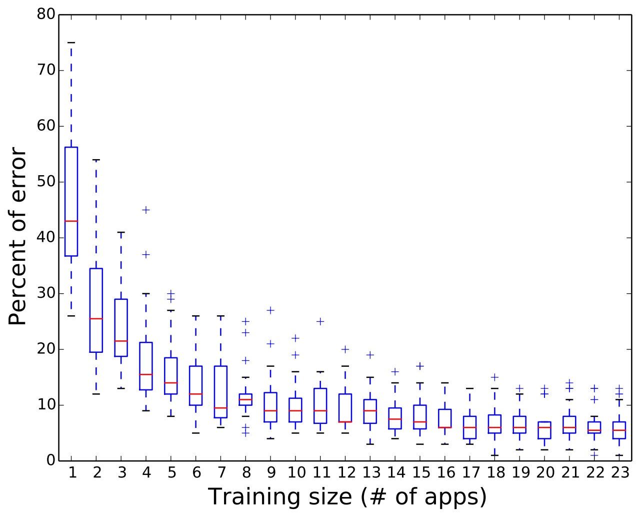


Figure 4.5: Number of applications VS. performance

4.8 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.

4.9 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 [4].

Chapter 5

Leveraging Automatic Test Generation for Improving Energy Models

With a significantly more accurate energy model, this chapter completes the first objective of this thesis (i.e., reproducible and accurate energy estimation models).

This chapter was published as:

- Shaiful Alam Chowdhury, Stephanie Borle, Stephen Romansky, Abram Hindle, "GreenScaler : Training Software Energy Models With Automatic Test Generation", Empirical Software Engineering Journal, Springer, 2018. This paper has also been accepted as a journal first paper and was presented at the 41^{st} ACM/IEEE International Conference on Software Engineering (ICSE 2019) [45].

This chapter shows how we can improve the accuracy of machine learning based energy models with automatic test generation. In the previous chapter (Chapter 4), there was a crucial observation that if we can add more apps in training (Figure 4.5), we can improve models' accuracy. However, adding one app requires manually writing one separate test script for that app. This is difficult and not scalable. This is where the usefulness of automated test generation becomes conspicuous.

We show that we can use a random test generation technique (simple and supports black-box testing) for generating test cases that can be used for running apps for collecting independent and dependent (i.e., energy consumption) variables for building energy models. Random test generation, however, might produces test cases that are weak—a test case that does not do anything to consume a significant amount of energy. This led us to create multiple test cases within a given time budget, and then to use different test selection heuristics. This way only the strongest test, based on a given test selection heuristic, is selected per app. We have evaluated three different test selection heuristics: 1) Code coverage, 2) CPU-utilization, and 3) *GreenOracle*-based heuristic.

The takeaways from this chapter include:

- With an appropriate test selection heuristic, we can employ random test generation for producing tests towards building more accurate software energy models.
- Code coverage, in spite of its wide adoption in traditional software testing, does not perform well for energy model building test selection.
- CPU-utilization is easy to use and performs similar to the complex *GreenOracle*-based heuristic in selecting tests for building energy models.
- The model we built with automated test generation (i.e., *GreenScaler*) performs great in detecting energy regressions. *GreenScaler* is useful for answering questions like: *Does the new version of an app consume more energy than the previous version?*

The *GreenScaler* tool and data are publicly shared [47]. With the data, researchers can further experiment with other machine learning algorithms for improving energy estimation accuracy. Developers can download and use the *GreenScaler* tool to estimate the energy consumption of their apps.

My role in GreenScaler: I, with the help of my supervisor, made plans for the methodologies, data collection, experimentation, and evaluations. My co-authors (Stephen Romansky and Stephanie Gil) helped me with the long and tedious measurement process (for collecting the energy consumption and resource usage for hundreds of apps). I also wrote the *GreenScaler* paper [45] with the guidance of my supervisor.

Impact: *GreenScaler* enables developers to estimate their Android apps' energy consumption without any cost. This contribution attracted different media coverage, including the coverage by the Global News [133]. *GreenScaler* was made available on-line on 20 July, 2018. According to the Google Scholar, *GreenScaler* has been cited 6 times (including the preprint) already. The most encouraging observation about *GreenScaler* is that it is actually used by researchers for estimating Android apps' energy consumption [24], [25]. According to Ayala *et al.*, *GreenScaler* is not only accurate, but also easy to use (verified by their independent study [25]), which compelled them to select *GreenScaler* over other estimation tools.

Abstract

Software energy consumption is a performance related non-functional requirement that complicates building software on mobile devices today. Energy hogging applications (apps) are a liability to both the end-user and software developer. Measuring software energy consumption is non-trivial, requiring both equipment and expertise, yet researchers have found that software energy consumption can be modelled. Prior works have hinted that with more energy measurement data we can make more accurate energy models. This data, however, was expensive to extract because it required energy measurement of running test cases (rare) or time consuming manually written tests. In this paper, we show that automatic random test generation with resource-utilization heuristics can be used successfully to build accurate software energy consumption models. Code coverage, although well-known as a heuristic for generating and selecting tests in traditional software testing, performs poorly at selecting energy hungry tests.

We propose an accurate software energy model *GreenScaler*, that is built on random tests with CPU-utilization as the test selection heuristic. *GreenScaler* not only accurately estimates energy consumption for randomly generated tests, but also for meaningful developer written tests. Also, the produced models are very accurate in detecting energy regressions between versions of the same app. This is directly helpful for the app developers who want to know if a change in the source code, for example, is harmful for the total en-

ergy consumption. We also show that developers can use *GreenScaler* to select the most energy efficient API when multiple APIs are available for solving the same problem. Researchers can also use our test generation methodology to further study how to build more accurate software energy models.

5.1 Introduction

Does software energy consumption matter? The answer is *yes*. Mobile device users prioritize longer battery life when investing in their next purchase [99], [191], [253]. Mobile device users also complain about battery life: recently Microsoft acknowledged that a software bug, unrelated to battery hardware, induced short battery life on Surface Pro 3 tablets [127]. Accordingly, developers are trying to write more energy efficient code to meet the need of consumers [44], [158], [198]. Research has shown that energy efficiency can be improved significantly with small code optimization [48], [95], [137], [192], [196]. To develop energy efficient software, developers need feedback about the energy consumption of their software. Unfortunately, developers are not sure how to measure and optimize the energy consumption of their apps [159], [191].

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, developers can use an externally developed and robust model, built from physical measurements of third party apps, to accurately estimate their own software's energy consumption. Measurements of third party apps, however, are hard to find as we need repeatable test cases and corresponding energy measurements. *These test cases are costly to build manually and are the main limitation of empirically derived models* [48]. 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 apps. *GreenScaler* leverages a continuous process of test generation to build an ever more robust corpus of energy measurements. As of writing, *GreenScaler* learns from a wide variety of 472 real world Android apps, 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.

The contributions of this paper are summarized as follows.

1) *We propose a process of continuously building an ever more accurate software energy consumption model using automatic test generation and test selection heuristics.* The success of automatic test generation for building accurate software energy models is significant. We can continuously improve model's performance by adding more apps in training. New research ideas can be explored with this approach. Researchers can investigate further for producing better energy models. For example, can we improve model's accuracy by building domain specific models (building a separate model for gaming apps for example)?

2) For using random tests, we need test selection heuristics. *We evaluate three test selection heuristics to understand which one is the most effective for selecting energy tests.* From empirical results we show the following. i) Code coverage is not a good heuristic for selecting tests to produce energy models. ii) A simple CPU-utilization heuristic performs similar to a complex energy-estimating test heuristic. To the best of our knowledge, we are the first to evaluate test selection heuristics for producing software energy models.

3) *We propose the GreenScaler model that can accurately estimate software energy consumption of apps without hardware instrumentation. GreenScaler* is trained and tested (with leave-one-out approach) on 472 apps with randomly generated test scripts. The model shows an upper error bound of 10% when compared with the ground truths, except for few extreme cases. As *GreenScaler* is built on randomly generated tests, it is also important to evaluate its accuracy on human written meaningful tests. For such manually written tests of 984 versions from 24 real world Android apps, the upper error bound of *GreenScaler* is always less than 10%. To the best of our knowledge, no previous software energy model was evaluated on such a large number of apps.

4) *We show that GreenScaler is accurate in finding energy regressions between versions of the same app, regardless of the amount of change in the source code. GreenScaler* detects energy regression even for a single API change, when such a change has significant impact on the app's energy con-

sumption. This is directly helpful for the developers who want to examine if a new version consumes more energy than the previous version. With *Green-Scaler*, researchers can build API recommendation systems for energy-aware developers.

5) *We publicly release our dataset and tools* to enable replication and extension [47]. The dataset contains measurements that took us nearly two years to collect, including time for test generation and time for actual energy measurements. Our automatic test generation and selection tool can be used to add more apps for building better energy models. Developers can directly use our energy prediction tool to estimate their apps' energy consumption.

5.1.1 Paper organization

The main focus of this paper is to build an accurate software energy model for Android systems—a model that learns from hundreds of apps. However, it is laborious to write tests to drive those hundreds apps. So we need automatic test generation. We study the previous test generation techniques in Section 5.2, with the description of other important concepts related to this paper. We show that Android Monkey is the best available test generation technique for building software energy models. However, Monkey has its own drawbacks—generates too many redundant events and does not offer us an way to control the distribution of individual events. So we made our own Monkey, *GreenMonkey*. GreenMonkey is still a random test generation technique, and might produce test cases that do not exercise energy consuming operations. We generated several test cases for each app to select the best one. To select the best test case, we need test selection heuristic. Which test should we select? Test that covers more code? We study the effectiveness of code coverage heuristic in selecting test cases that exploit energy expensive resources, and found that code coverage would not be a good heuristic for selecting energy consuming test cases (Section 5.3). Instead, we focused on resource-utilization heuristics. Section 5.4 describes the whole *GreenScaler* methodology of building software energy models with resource-utilization heuristics. The rest of the paper is about evaluating our model from different perspectives (Section 5.5 to Section 5.8). In Section 5.9, we discuss the future research avenues with our model building approach, followed by the description of our dataset (Section 5.10), Threats to Validity (Section 5.11), Related work (Section 5.12), and Conclusion (Section 5.13).

5.2 Background

This section explains the important concepts that are frequently used in this paper. It also describes the motivation for energy model building automated test generation.

5.2.1 Power vs. 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 T ($E = P \cdot T$) and expressed in *joules* [5], [49], [50]. Understanding the difference between power and energy is important to develop energy efficient system. A misconception exists among developers: improving execution time automatically improves energy efficiency [159], [191]. Improving execution time reduces T in the equation. However, with the reduced execution time the CPU workload may also increase, which can switch the CPU to its highest frequency, which is also its highest power using state, thus negatively affecting the overall energy consumption. Furthermore T can be reduced by parallelizing a task across multiple cores, which could induce even higher power use. This is also confirmed by a previous study that shows that less execution time does not necessarily indicate less energy consumption [92].

5.2.2 System calls and CPU time

System calls act as the bridge between an app and the OS. For example, `socket` is a system call responsible for creating communication endpoints, whereas `read` takes the responsibility to read from a file handle. Counting system calls of different types can thus provide an estimation of the amount of different resource usage by an app [4], [48], [194].

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 [124]. A CPU can operate in different power consuming states, which complicates software energy modeling [159]. A CPU jiffy, however, can be of different time intervals based on the CPU states. Thus considering CPU jiffies as CPU time would mitigate some intricacy involved in software energy modeling.

5.2.3 Energy measurement: GreenMiner

For measuring energy consumption and resource usage, we used *GreenMiner*, which is fully described in Hindle et al. [102]. *GreenMiner* provides accurate energy measurements for Android apps and is widely accepted in the software energy research community [4], [5], [48], [95], [101], [204]. The main components of this test-bed are a lab-bench power supply (a YiHua YH-305D), a test-runner computer (a Raspberry Pi model B computer) for controlling the experiments, an energy measurement IC (Adafruit INA219 breakout board), a micro-controller (Arduino Uno) for collecting energy measurements, and a system-under-test (a Galaxy Nexus phone) (Table 5.1). The Arduino and Raspberry Pi are powered by a USB hub. Each testbed costs approximately $250, each phone originally cost approximately $500, and the green miner service is run on a separate server ($1000). Development of the GreenMiner hardware and software itself was more than $32000 in developer time. GreenMiner software is freely available for download [102].

A test-runner, a Raspberry Pi, is connected to a particular system-under-test, a Galaxy Nexus. The test-runner pushes and runs tests on the Galaxy Nexus, and collects measurements from the Arduino. Afterwards the test-runner downloads statistics and other meta-data from the system-under-test. The responsible test-runner, a Raspberry Pi, then uploads the measurements to a central server running the GreenMiner webservice. The current Green-

Table 5.1: Specs of the Samsung Galaxy Nexus phones used for the experiments [46].

COMPONENT	SPECS
OS	Ice Cream Sandwich, 4.4.2
CPU	Dual-core 1.2 GHz Cortex-A9
GPU	PowerVR SGX540
MEMORY	16 GB, 1 GB RAM
DISPLAY	AMOLED, 4.65 inches
WLAN	Wi-Fi 802.11 a/b/g/n

Miner consists of four such identical testbeds to speedup and parallelize the data collection process. Figure 5.1 shows the innards of one of the four identical settings of the GreenMiner. The GreenMiner service is a continuous testing service whereby users may submit tests to be run and measured. After submitting a batch of tests to the GreenMiner, one of the phones is randomly selected for executing a test. As a result, four different tests can run in parallel to reduce the measurement time. *GreenMiner* maintains the same system state for each test by cleaning any installed apps that ran previously.

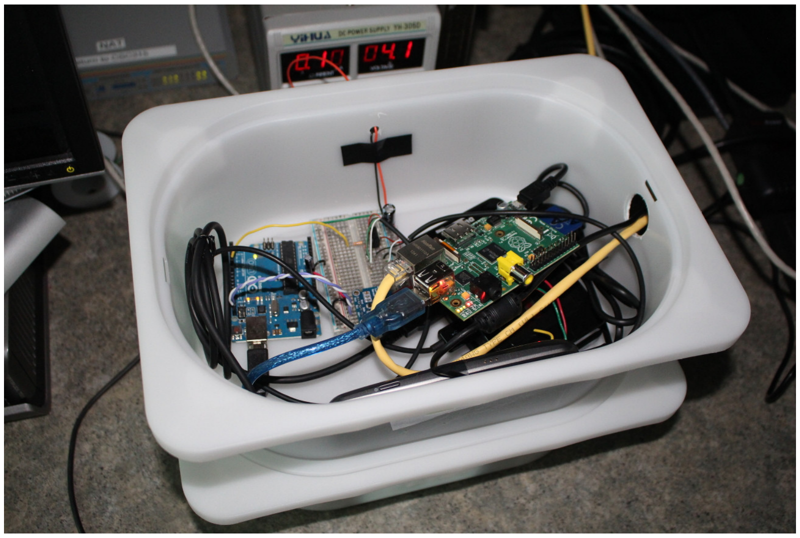


Figure 5.1: One of the four identical GreenMiner settings. Photo used with permission from the Green Miner paper [102].

Variations in energy consumption and resource utilization are observed in different measurements for the same test. Consequently, all the *GreenMiner* based previous work repeated any specific measurement multiple times [5], [48], [95]. Similarly, all of our tests for measuring energy and resource usage were run 10 times and the mean value was used.

5.2.4 Energy estimation: GreenOracle

There exist different types of software energy models: instruction based models [92], [219], utilization based models [40], [90], [227], and others [4], [5], [48], [194] (described in Section 5.12). We followed the philosophy of count based energy modeling similar to our previous *GreenOracle* [48]. *GreenOracle* modeled software energy consumption based on the counts of different resource usages: number of CPU jiffies, number of different invoked system calls and so on. *GreenOracle*, however, has some limitations. This model was built only using 24 Android apps. Although the size of the training set was enlarged by adding different versions from those apps, this did not improve the model significantly. This is not surprising as only one manually written test case was used for each app. As a result, different versions of the same app might have executed the exact sequence in source code, offering very similar information on resource usage and energy consumption for a given test. An accurate energy model, however, requires training from a wide variety of workloads. In this paper, we show that a model based on such a small dataset is not accurate in estimating energy consumption of apps from very different domains.

5.2.5 Energy model building test generation

To measure the energy consumption of an app we need to drive the app with some kind of test or benchmark. To run an app on *GreenMiner*, we need a test script to replay operations on the app. An example of a test case is: open Firefox, load a Wikipedia page and scroll over the page for five minutes—as if a user is reading the page.

In our previous *GreenOracle* [48] model, we demonstrated that adding new apps in training improves the accuracy of software energy models. Unfortunately, adding more apps requires manually writing test cases for each app, which makes it infeasible to have an energy model trained on hundreds of apps. Manual software testing is difficult and expensive [16], [132], which motivated a significant number of research in automated software testing [94], [201], [221], [258], [267]. Automatic test generation is when tests are created automatically through algorithmic means to drive the software under test. There has been a significant number of research dedicated for Android test generation [15], [156], [157], [161], [172].

Test generation can be completely random [16] such as Android Monkey [175] where random events are injected to an app. Some random test generation strategies, such as Dynodroid [156], extract the layout of the GUI components from an app screen and generate events based on the extracted components. *Search heuristics* are also employed to guide the test generation process, known as Search Based Software Testing (SBST) [94]. For example, code coverage is a *search heuristic* where the objective is to generate test cases to maximize code coverage [82]. Techniques like Sapienz [161] use multi-objective optimization (maximize fault detection and minimize test sequence length) in order to guide the test generation. Genetic algorithms can be used in SBST to find the optimal set of test cases: from a set of candidate solutions (test cases), the test generation process applies mutation on individual candidates, or cross over across two or more candidates, or a combination of both to find better solutions [157], [161].

In order to produce the *GreenScaler* model, a model trained on hundreds of AndroZoo apps (described later), a test generation tool is required that is publicly available, and does not require app's source code for test generation; the AndroZoo database only contains the executables (i.e., APKs) without any source code. Choudhary *et al.* [43] and Mao *et al.* [161] performed detail surveys on the most well-known Android test generation tools. Table 5.2 shows the summary of the surveyed tools to identify the ones that are suitable for the *GreenScaler* model building. Only 7 out of the 19 surveyed tools are potentially suitable for the model building test generations. It is important to

Table 5.2: A summary of the existing Android testing tools for model building test generation.

Tool	Available?	Works without source code?	Suitable for *GreenScaler*?
AndroidRipper [15]	*Yes*	*Yes*	✓
DroidFuzzer [260]	*No*	*Yes*	
NullIntentFuzzer	*Yes*	*Yes*	✓
IntentFuzzer [215]	*Yes*	*No*	
Monkey [175]	*Yes*	*Yes*	✓
MonkeyLab [146]	*No*	*Yes*	
Dynodroid [156]	*Yes*	*Yes*	✓
ACTEve [18]	*Yes*	*No*	
TrimDroid [172]	*Yes*	*Yes*	*N/A. Generates the whole test suite*
A^{3}E-DFS [26]	*Yes*	*Yes*	✓
SwiftHand [42]	*Yes*	*Yes*	✓
ORBIT [259]	*No*	*No*	
PUMA [93]	*Yes*	*Yes*	✓
EvoDroid [157]	*No*	*No*	
SPAG-C [143]	*No*	*Yes*	
Thor [3]	*Yes*	*Yes*	*N/A. Requires existing test suite*
JPF-Android [167]	*Yes*	*No*	
CrashScope [176]	*No*	*Yes*	
Sapienz [161]	*Yes*	*Yes*	N/A. Authors do not share source code

find the best performing tool among these 7 that can be used for the energy model building test generation. Given the time cost of test generation and the collection of energy and resource usage measurements, it is infeasible to use all the available tools for the energy model building process.

Choudhary *et al.* [43] concluded that evaluation of existing test generation tools can be biased by the apps selected for evaluation. So they did a thorough study on most of the well-known Android test generation tools that are publicly available. The authors found that some tools are hard to use, and might demand continuous communication with the actual authors which is often not feasible. Interestingly, after their rigorous evaluation, Choudhary *et al.* [43] concluded that random test generation—Monkey and Dynodroid—outperform all the existing Android test generation techniques by a large margin, including all of the *GreenScaler* suitable tools mentioned in Table 5.2. AndroidRipper exhibits the worst code coverage and requires *major effort* to use. PUMA, although requires little effort to use, its code coverage and framework compatibility are very poor compared to others. Instead of the very similar performance, Monkey is much simpler and 5x time faster in test generation than Dynodroid [156]. Unlike Dynodroid, monkey does not have any framework compatibility issue [43], making Monkey as the most suitable tool for generating tests for a wide variety of Android apps.

Monkey, however, is notorious for some of its limitations including app irrelevant events [156], [161] like volume control, screen capture etc. Moreover, although Monkey allows setting distributions of different groups of events [175], it does not allow the user to define the distribution of events (e.g., generating 60% `tap` events). Inspired by the success and drawbacks of Monkey, we propose a very similar random test generation tool, GreenMonkey. GreenMonkey is no different than Monkey, except control over the distribution of events is allowed and app irrelevant events such as volume control are discarded. In section 5.6, we show that this little modification indeed improves model's performance. As a result, we continue our test generations with GreenMonkey instead of Monkey.

With automated test generation, we can generate a test to drive a given app so that we can collect resource usage (independent variables) and energy consumption (dependant variable) to build energy models. However, the generated test case might not be exercising any energy expensive resources, and thus will not provide any useful information for the models. As a result, we aim to generate more than one tests for each app and select the one that has the highest potential of exploiting energy hungry resources. So we need test selection heuristics. In the next section, we evaluate the effectiveness of code coverage heuristic in selecting test cases for energy model building.

5.3 Code coverage heuristic

In traditional software testing, code coverage is one of the most used metrics to evaluate the effectiveness of a test generation approach [16], [17], [37], [82], [180]. In general, a test with higher coverage is expected to reveal more faults in a system [82], [180]. However, Inozemtseva and Holmes [114] found that coverage is not strongly correlated with test suite effectiveness for finding faults. In contrast to traditional testing, the objective of our test heuristic is to select test cases that exploit different energy consuming hardware components. A model built on test cases that do not observe energy expensive work, would not be accurate. Such a model would fail to estimate the energy consumption of a foreign app that accesses different energy consuming hardware components.

In this paper, we evaluate the potential of the code coverage heuristic to select tests that can be used for producing energy models. Does covering more code necessarily indicate exercising more energy expensive resources? To answer this question, we investigate the correlation between coverage and power usage. If test suites with high code coverage implies high power usage, then code coverage would be a valid test generation heuristic. Otherwise, different avenues of test generation heuristics would need to be explored. Our methodology is a near replication of Inozemtseva and Holmes [114], but we focus on power usage instead of fault detection ability.

5.3.1 Methodology

In order to determine the suitability of code coverage as a heuristic for selecting energy consuming tests, we require: 1) a set of Android apps with available test cases; 2) a process to generate test suites of different sizes; 3) coverage and energy consumption of the generated test suites; and 4) a statistical method to calculate the correlation between coverage and power usage.

Selected applications

The difficulty of finding open source Android apps with JUnit test suites limited the number of potential apps. To match the work of Inozemtseva and Holmes, the apps need to have a coverage of nearly 50% or more for either class, method, line, or block. This further narrowed down the available choices. We finally selected three open source Android apps: Klaxon—a pager; Password Hash—generates passwords; and Storyhoard—a choose your own adven-

Table 5.3: Description of selected apps' master test suite coverage.

Feature	**Klaxon**	**Password Hash**	**Storyhoard**
Source Lines of Code	1601	541	3749
Executable Lines of Code	799	247	1959
Master Suite Size	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%)

ture app. Table 5.3 shows the characteristics of the selected apps. The varied numbers of lines in source code, number of methods, classes and blocks help better to understand the relationship between code coverage and power usage.

Generating test suites

A test suite is a collection of sampled test cases from the master suite. The master suite contains all the test cases for an app written by the developers. Following the similar approach of Inozemtseva and Holmes [114], we generated test suites of different sizes by sampling the existing master suite (collection of all JUnit test cases). For example, a test suite of size 3 means there are 3 test cases in the test suite. In the study of Inozemtseva and Holmes, the selected sizes for generating random test suites were 3, 10, 30, 100, 300, 1000, and 3000. In their subject Java projects, the largest and smallest number of tests were 7,947 and 628 respectively. In our selected Android apps, however, the largest master suite (from the Storyhoard app) has only 75 test cases. As a result, we could not select sizes similar to Inozemtseva and Holmes.

In order to get reliable statistical results by generating a large number of test suites, we selected more sizes within short intervals. We started with test suite size 2 and then repeated the procedure with test suites of sizes 4, 7, 10, 13, 16, 27, 40, 64, and 73. Once the sizes were decided, a Python program was written to randomly choose test cases from an app's master suite and create different sized test suites from them. For each test suite size, 100 test suites were generated with random sampling from the master suite. This allowed us to have a diverse collection of test suites of various sizes and coverage levels. Algorithm 1 illustrates the whole process of generating test suites.

Algorithm 1: Generating 100 test suites. Each test suite will have a given number (based on the suite size) of randomly sampled test cases.

```
input  : master_suite, suite_size
/* The master suite of an app and a given suite
   size.                                          */
output: collection_test_suites
/* 100 test suites each with a fixed number (i.e.,
   suite_size) of randomly sampled test cases.    */

1  collection_test_suites ← [];
2  for suite_number ← 1 to 100 do
3      test_suite ← [];
4      for test_number ← 1 to suite_size do
5          test_case ← Random(master_suite);
           /* select a test case randomly from the
              master suite.                       */
6          test_suite.append(test_case);
7      end
8      collection_test_suites.append(test_suite);
9  end
10 return collection_test_suites;
```

Capturing coverage and energy consumption of each test suite

In order to capture the coverage of each test suite, we used a third-party tool `emma` [65] on the source code of each app. Emma provides four types of code coverage: line coverage, method coverage, class coverage, and block coverage. We captured all the coverage reports for our analysis.

Energy was measured using the *GreenMiner* by averaging 10 runs of each test suite. However, during the energy measurement process, "*coverage true flag*" was disabled to avoid any overhead incurred from coverage calculation.

Kendall's τ as a measure of effectiveness

We calculated the Kendall's τ correlation coefficients between coverage and power usage for the generated test suites. Kendall's τ does not assume any distributions of the data—unlike Pearson's correlation coefficient it does not assume that the two variables (i.e., coverage and energy consumption in our case) have a linear relationship. Similar to Inozemtseva and Holmes, we calculated the coefficients with both uncontrolled and controlled suite size.

Uncontrolled suite size: Combine the measurements of coverage and power usage from all the generated test suites and calculate the Kendall's τ correlation coefficient between coverage and power usage.

Controlled suite size: Combine the measurements of coverage and power usage from the generated test suites with a particular suite size (e.g., all suites with size 2), and calculate the Kendall's τ correlation coefficient.

5.3.2 Analysis of results

For uncontrolled suite size, Table 5.4 shows the Kendall's τ correlation coefficients between code coverage and power usage (*watts*) for all the three apps. Power usage against test suite size is also presented. Table 5.5 and 5.6 show the correlations when the suite sizes are fixed to 2 and 13 respectively (i.e., controlled suite size). We did not include results for other suite sizes (e.g., correlations for suite size 4, 7, 10 and so on) as the observations are similar.

We observe good/strong correlation between energy and code coverage, especially when we do not control for test suite size (Table 5.4). This is not surprising; larger code coverage usually means larger execution time which has direct impact on the total amount of energy consumption (i.e., $E = P \cdot T$). For example, from our results the highest correlation is observed for the Storyhoard app when suite size is not fixed (minimum 0.78 for class coverage and maximum 0.83 for method coverage, Table 5.4). However, this is the same setup when we observe the highest correlation between energy consumption and test duration (correlation coefficient 0.95). The lowest correlation is found for the same app when suite size is fixed to 13 (Table 5.6). Interestingly, for the same setup, the correlation between energy and test duration is the lowest. This implies that good correlation between coverage and energy consumption does not necessarily indicate that the test cases are exercising energy hungry resources—test duration might be the major factor for the observed good correlation. In order to use coverage as the heuristic for energy model building test generations, we need to observe good correlation between coverage and power usage, instead of coverage and energy consumption.

The correlations between coverage and power usage are weak for Password Hash and Storyhoard when suite size is not controlled (Table 5.4). In case of Klaxon, we observe moderate correlation. However, for controlled suite sizes, the correlation for Klaxon drops significantly (Table 5.5 and 5.6). Results from our subject Android apps indicate that covering more code does not necessarily indicate more exercise of power expensive source code portions. This supports the intuition that all code is not equally heavy in power usage. For example, code that makes an `HTTP` request might consume more energy than a larger segment of code without high CPU usage or network operations [141].

Table 5.4: Correlation between code coverage and power with uncontrolled suite size. Suite size vs. power is also presented.

		Correlation with Coverage				**Correlation**	
Applications		**Method**	**Class**	**Line**	**Block**	**Suite size**	**Duration**
Klaxon	**Power**	0.57	0.58	0.58	0.58	0.59	N/A
	Energy	0.67	0.67	0.69	0.69	0.67	0.91
Password Hash	**Power**	0.06	0.08	0.02	0.01	−0.05	N/A
	Energy	0.56	0.53	0.60	0.61	0.68	0.94
Storyhoard	**Power**	0.22	0.23	0.20	0.19	0.20	N/A
	Energy	0.83	0.78	0.82	0.81	0.85	0.95

In order to build *GreenScaler*—modeling software energy against the indirect measurement of resource usage—we need test cases that are more likely to exploit different resources and energy consuming portions of the code. This short study led us to focus experimentation on resource-utilization heuristics rather than investing time on code coverage based test generation. Build-

Table 5.5: Correlation between code coverage and power with suite size fixed to 2.

Applications		**Correlation with Coverage** **Method**	**Class**	**Line**	**Block**	**Correlation Duration**
Klaxon	**Power**	0.13	0.40	0.31	0.34	N/A
	Energy	0.12	0.41	0.31	0.35	0.79
Password Hash	**Power**	0.33	0.38	0.31	0.31	N/A
	Energy	0.69	0.67	0.67	0.66	0.85
Storyhoard	**Power**	0.45	0.52	0.35	0.13	N/A
	Energy	0.47	0.51	0.37	0.17	0.90

Table 5.6: Correlation between code coverage and power with suite size fixed to 13.

Applications		**Correlation with Coverage** **Method**	**Class**	**Line**	**Block**	**Correlation Duration**
Klaxon	**Power**	0.26	0.27	0.26	0.27	N/A
	Energy	0.52	0.51	0.52	0.53	0.81
Password Hash	**Power**	0.21	0.21	0.13	0.13	N/A
	Energy	0.40	0.40	0.23	0.26	0.91
Storyhoard	**Power**	0.26	0.38	0.15	0.13	N/A
	Energy	0.21	0.26	0.13	0.12	0.47

ing automatic test cases for hundreds of apps with a heuristic, running them on *GreenMiner* for collecting energy consumption and resource usage counts, and then applying/tuning/validating machine learning models demand several months.

Findings: Code coverage relates more to test duration than to power usage. With code coverage as the heuristic, test with longer execution time might be selected for model building, in spite of its weakness in exercising energy expensive portions of the source code. This implies that coverage will not be a good heuristic if employed for selecting tests to model software energy consumption. We need different heuristics that use actual resource-utilization, and thus capture energy expensive portions of source code.

5.4 *GreenScaler* methodology

In this section, we describe the complete *GreenScaler* methodology of using resource-utilization heuristics for generating tests to build continuously refined software energy models.

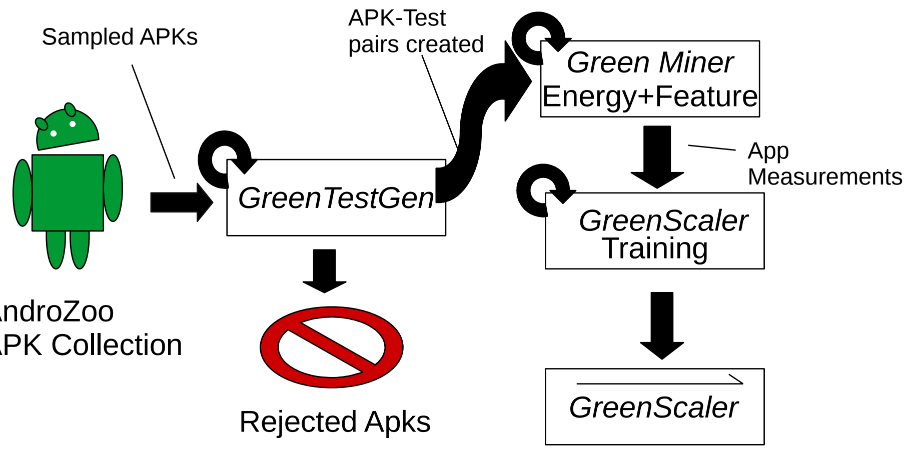


Figure 5.2: The process of developing *GreenScaler*. The model learns continuously with new apps using test selection by *GreenTestGen*.

The process of producing an energy model from a large corpus of energy measurements is to: 1) collect Android apps (Section 5.4.1); 2) generate tests for the collected apps (Section 5.4.2); 3) collect energy consumption measurements, system calls measurements, and other process counters while running an app's test (Section 5.4.3); 4) add measurements to the training corpus; 5) and finally train our model. Figure 5.2 summarizes the process of developing *GreenScaler*. We also need to evaluate the effectiveness of this process, so model selection (Section 5.4.4), feature engineering (Section 5.4.5), and validation (Section 5.4.6) are also described.

5.4.1 Collecting Android applications

We sampled apps randomly using the database provided by AndroZoo [13], and collected about 500 apps. AndroZoo provides millions of apps for Android research that were collected from 3 different app stores: Google Play, Anzhi, and AppChina. Some of the apps did not install or run properly. After removing those, 472 apps were used to develop the proposed *GreenScaler* energy model. AndroZoo does not provide app categories, but using the `screencap` program, described in Section 5.4.2, we manually investigated a sample from the 472 collected apps, and observed apps from different domains: media players, games, utility, etc. Some sampled apps (radio, streaming, and online games) heavily used the network. Table 5.7 shows the categories of 100 randomly selected apps. The categories of these apps were defined based on two of the authors consensus. The unknown category contains non-English apps that are difficult to categorize. It is clear from Table 5.7 that our sampled apps are from very different domains. This is important for building a robust energy model that should work across different types of Android apps.

5.4.2 Automatic test generation with resource-utilization heuristics

The futility of code coverage heuristic encouraged generating test cases based on resource-utilization heuristics. We evaluate two such heuristics for building software energy models: CPU-utilization and E-heuristic (estimated energy utilization).

CPU-utilization heuristic: Select the test case with the highest CPU time,

Table 5.7: Categories of the 100 randomly selected AndroZoo apps.

Category	**Count**	**Category**	**Count**
Game & Puzzles	23	Book	6
Utility	11	Entertainment	5
Unknown	11	Media	5
Business & Website	10	Finance	2
Education	9	Health	2
Communication	8	News	1
Tools	7	**Total**	100

as CPU is a major source of energy consumption. We can argue that the CPU-utilization heuristic may select test cases that are biased to CPU utilization only and may ignore the utilization of other resources like network, file etc. Consequently, a model built on such test cases might fail to estimate energy consumption of apps with high network or file operations. As a result, we selected another heuristic, E-heuristic, that exploits other energy heavy resources as well.

E-heuristic: Select the test case with the highest estimated energy consumption based on an actual software energy model—*GreenOracle* [48]. In this approach, a test case is selected that has the highest estimated energy consumption based on all the different hardware components utilization. In a nutshell, an existing energy model is used to generate tests towards producing an even more accurate energy model.

For generating tests based on these two heuristics, we propose *GreenTestGen*. For a given app, *GreenTestGen* creates a number of test cases with the help of GreenMonkey. With GreenMonkey, each test case consists of different randomly selected events. *GreenTestGen* runs all the test cases for the app, and selects the one that maximizes a given heuristic function. It is important to note that any test generator (e.g., Monkey, Dynodroid) can be used with *GreenTestGen* just by replacing GreenMonkey. GreenMonkey and *GreenTestGen* are fully depicted in Algorithm 2 and 3 respectively.

Algorithm 2: GreenMonkey

```
input  : no_of_events
output: test_script
1 EVENTS_POOL={adb shell events};
2 test_script ← SequenceOfEvents (no_of_events,
   DISTRIBUTION_OF_EVENTS, EVENTS_POOL);
3 return test_script;
```

Algorithm 3: GreenTestGen

```
input  : An AndroZoo app, a testGenerator
/* calls GreenMonkey as the default test generator
    */
output: Test case for the app
1  if CrashCheck (App)==True then
2      Exit();
3  end
4  no_of_events ← Random(10, 40);
5  play_time ← 0;
6  max_heuristic_value ← 0;
7  while play_time ≤ 30 mins do
8      test_script ← testGenerator(no_of_events);
9      heuristic_value ← Execute(App, test_script);
10     if heuristic_value>max_heuristic_value then
11         max_heuristic_value ← heuristic_value;
12         best_test ← test_script;
13     end
14     update play_time ;
15 end
16 ScreenCap(App, best_test);
```

GreenMonkey: A pool of `adb` events—such as `tap x y`, `swipe x1 y1 x2 y2`, `text string or number`, `ENTER`, `DEL`, `tapmenu` etc.—was created where the values of pointer locations, strings and numbers are selected randomly (line 1 of Algorithm 2). The pointer locations were restricted to a specific range to avoid clicking on the phones' `HOME` and `BACK` buttons. Different events have different impacts on generating useful test cases. We did manual observations on 30 randomly chosen apps. Not surprisingly, `input tap` was found as the most contributing event toward generating useful test cases. Events like `swipe`, `input text`, and `keyevents` were assigned similar priority, followed by `tapmenu`. So instead of selecting events uniformly randomly, GreenMonkey is biased so that a test script contains more `tap` events than any other events, whereas the event `tapmenu` occurs the least (DISTRIBUTION_OF_EVENTS, line 2). A test script is thus a set of different events—separated by a 2 second sleep time—and the number of events in a test case might differ among the apps.

***GreenTestGen*:** When an app is selected, *GreenTestGen* first checks if the app installs and runs properly (line 1, Algorithm 3). In case of a success, the app is then run to find the best test case—the test that maximizes a selected heuristic function. *GreenTestGen* selects the number of events randomly—from 10 to 40 events so that a test is neither too short nor too long (line 4). A test script that is too short (few events) 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 (same number of events), any machine learning model would ignore duration as an important feature, which would be devastating for predicting an unknown app's energy with a very different test duration. Each app is then run for a fixed 30 minutes before selecting the best test case (line 7).

GreenTestGen calls a test generator (GreenMonkey as the default, algorithm 2) to create a test case with the selected number of events (line 8). After running the test script and measuring the heuristic value (line 9), *GreenTest-*

Gen creates another test with the previously fixed number of events and run it to evaluate if the heuristic value (e.g., CPU time) is increased, in which case it updates the best test case as the most recent one. At the end, the best test case is the one with the highest heuristic value (line $10 - 12$). Before running the test on *GreenMiner*, we also added a 10 seconds idle time at the end of the selected test case. This is to capture any associated tail energy leak [5], [194] that can occur at the end of running the test (tail energy is explained in Section 5.12.1).

After spending 30 minutes to generate the best performing test case, *Green-TestGen* replays the best test case in order to capture the screenshots using `screencap` program and save the images (line 16). This allows us to investigate each app's behaviour and type if needed—construction of Table 5.7 for instance.

5.4.3 Collecting energy consumption and resource usage

For collecting the energy consumption and resource utilization statistics for all the apps, we used *GreenMiner* (described in Section 5.2.3). All of our measurements, for energy and resource usage, were separate from each other so that the actual energy measurements are not affected by the programs capturing resource usage. As mentioned earlier (Section 5.2.3), we repeated each test 10 times and took the mean value of the measurements, for both energy and resource usage.

We used the `strace` program for tracing all the different system calls invoked by an app. The `-c` option was enabled so that we only capture the summary counts of each system call. We also enabled the `-f` option so that system calls invoked by the child processes are captured as well. A script was written that starts running just before the test case of the app under test (AUT) starts its execution. The script then waits and checks for the availability of the AUT in the current running process list. Once it finds the process in the list, it immediately starts the `strace` program with the process *id*. The `strace` program stops and writes the summary counts in a file, when the test case of the app under test finishes.

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 [48]. 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 took the difference of counts between after and before running a test. The local information is collected after a test run is completed.

One more important feature that was ignored in some other software models (e.g., *GreenOracle* [48], and PETrA [184]) is an app's interface colour. In case of OLED screen, up to 40% reduction in screen-based energy consumption is achievable by switching interface with white background to dark background [142]. With such dependency on colour, an energy model would be inaccurate if it does not consider screen colour information. We used `screencap` program to capture screenshots while running a test case for an specific app. The script is very similar to the script we used for system calls, except it runs the `screencap` program instead of the `strace` program. Motivated by Dong *et al.* [61], we calculated the average red, green, and blue values (RGB) for all the pixels across all screenshots. Each of these three averages is then multiplied by the test duration—as we model energy consumption instead of power. For example, if we capture three screens, and the average red pixel values are 100, 150, and 200 in those three screens, the calculated red value is 150—i.e., $(100 + 150 + 200)/3$. And the red value used by a model is 150 multiplied by the test duration.

5.4.4 Algorithms for energy models

We have to train a model based on resource usage (independent variables) and energy consumption (dependent variable). We compared three machine learning algorithms and chose the best performing and most interpretable one for 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 methods of regularization. The biggest advantage with these algorithms are that they are very easy to interpret.

Ridge: 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 5.1 [96]:

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}} [\sum_{i=1}^{m} (Y^i - \sum_{j=0}^{n} \theta_j X_j^i)^2 + \lambda \sum_{j=1}^{n} \theta_j^2] \tag{5.1}$$

In our case, m is the number of apps, 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 regularization in order to avoid overfitting the training data.

Lasso: One of the characteristics of ridge regression is that it does not eliminate unnecessary features—it retains features with tiny coefficients. Lasso, on the other hand, drops features from a group of highly correlated features. The only mathematical difference between ridge and lasso is the regularization term in equation 5.1; lasso uses l_1 (i.e., $\sum |\theta_j|$) regularization instead of l_2 (i.e., $\sum \theta_j^2$) [96].

Support vector regression: SVR, in contrast to Ridge and Lasso, is more complex and in many cases can exhibit better performance than simple linear regression [247]. Interpretation of such a model, however, is difficult and can be complicated for the developers to find which features are contributing more toward energy consumption. To mitigate this, we only used the linear kernel instead of the more complicated sigmoid, radial basis function (RBF), and polynomial kernels. SVM^{light} implementation was used for SVR that is based on ϵ-SV regression [247] which finds a function $f(x)$ that does not deviate more than ϵ from the ground truth.

5.4.5 Feature engineering

Some system calls are similar in functionality. For example, both `fsync` and `fdatasync` do the similar file synchronization work—"synchronize a file's in-core state with storage device"[147]. If we treat these system calls the same, then apps that use either can benefit from training. As a result, similar system calls are grouped together similar to our previous work on GreenOracle [48]. All the grouped system calls are presented in Table 5.8. In general, if an app invokes 10 `fsync` and 10 `fdatasync`, a new feature Fsync (group name) was used with 20 counts in our model.

Table 5.8: Grouping similar system calls according to OS semantics.

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"

Compared to the number of apps, the number of features in our dataset is quite large—22 from CPU and pertinent information, 4 for R, G, B, duration, and 99 from grouped and individual system calls. This large number of features leads to model overfitting. Among the three algorithms we used, SVR is hard to interpret and does not help in feature selection. Lasso with $l1$ regularization yields a more sparse coefficient vector (i.e., many features with coefficient 0) than Ridge, thus more suitable for feature selection. However, with high correlation among features, Lasso selects many features with negative coefficients, which made our previous *GreenOracle* model [48] less interpretable and less accurate.

We addressed this issue with the recursive feature elimination method with Lasso. After the first iteration, we manually removed the features with low coefficients. We followed this procedure until we got a set of features with reasonably high coefficients. This procedure subsequently deleted highly correlated features. Only 80% of the measurements from AndroZoo (i.e., 377 randomly selected apps out of 472) were used for feature selection. Table 5.9 describes the final set of selected features. We got this same set of features for the both resource-utilization heuristics based test sets (CPU-utilization and E-heuristic). This small number of features makes a model easy to interpret.

Learning algorithms perform slowly and suffer from low accuracy with high variance in feature values [109]. In our case, we indeed observed such high variance. This was solved by using 0-1 normalization, as in equation 5.2, where $\mathbf{x}$ is the actual and $\hat{\mathbf{x}}$ is the normalized feature vector.

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})} \tag{5.2}$$

5.4.6 Testing and cross validation

We applied 10-fold cross validation for all the three algorithms to tune the regularization parameters, known as model validation phase before testing with the test data. 10-fold cross validation is helpful when the data size is small. Because it does not require dividing the data into three parts: training data, validation data, and testing data. With 10-fold cross validation, we divided the training data into 10 segments (each containing the same number of apps). At phase i, segment i is used as the test data, whereas the other 9 segments are combined to make the training data. This way, 10 models are evaluated with 10 different test data partitions. We observe the model's performance to see if we need to adjust the regularization parameters (i.e., the penalty sizes). The process stops when no more improvement is possible and

Table 5.9: Selected features (CPU and others, duration, colour, and system calls) from feature selection process to model energy consumption for Android apps. This table suggests that the major sources of Android energy consumption are CPU, context switches, test duration, screen color, file operations, and network operations. The weight represents the energy consumption for each unit (e.g., one CPU jiffy) of the features. The weights of each feature are discussed in Section 5.7.3.

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	7.604e-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	5.260e-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

when the performance of the 10 different models are similar. The coefficients of the regularization (e.g., λ for ridge and lasso) for all the three algorithms were finalized during this cross validation phase. It is important to note that, we only used 80% of the apps for the cross validation. Also, for evaluating a model's accuracy (starting from Section 5.5), an app under test was always excluded from the training set.

5.5 Evaluating resource utilization heuristics

This section evaluates and compares the two resource-utilization based test generation heuristics: CPU-utilization heuristic (CPU time) and E-heuristic (energy estimation).

Using the 472 collected AndroZoo apps, we built two energy models.

- *Model* $_{CPU-H}$: trained with all the selected features from Table 5.9, but exclusively using measurements from tests generated by the CPU-utilization heuristic.

- *Model* $_{E-H}$: trained with the same feature set, but exclusively using measurements from tests generated by the E-heuristic.

Both the models were trained with Lasso because of its superior performance over the others (discussed later). Regularization parameters were used from the cross validation phase (Section 5.4.6). Comparing the accuracy of these two models would tell us which test generation heuristic produces better tests for developing software energy models.

We evaluate the models' accuracy (percent of error in *joules*) following the leave-one-out approach [97]. This ensures that an app under test was never seen in training. Each model was evaluated on two different sets: measurements from the CPU-utilization heuristic based tests and from the E-heuristic tests. This produces two error distributions for each model.

We applied the Anderson-Darling normality test [243] and found that none of the error distributions are normally distributed. This led us to select a non-parametric test to decide if the error distributions from *Model* $_{CPU-H}$ and *Model* $_{E-H}$ are statistically different. We used the Kruskal-Wallis test [104], which does not assume data is normally distributed, and found that these two models (from two different heuristics) produce statistically different error distributions ($\alpha = 0.05$ and $p = 0.01$). In order to find which model offers better accuracy, we used the pairwise Wilcoxon-rank-sum test [32] to calculate the 99% confidence interval of mean percent error in *joules*. We also calculated Cliff's delta [55] to measure the effect size between the two error distributions (presented in Table 5.10).

When applied to CPU-utilization heuristic based tests, *Model* $_{CPU-H}$'s mean confidence interval is lower than *Model* $_{E-H}$ with negligible effect (Cliff's delta). *Model* $_{E-H}$, however, similarly outperforms *Model* $_{CPU-H}$, with slightly better accuracy, when evaluated for the E-heuristic based tests. In other words, each model slightly outperforms the other when evaluated on measurements arising from its own test heuristic. We select *Model* $_{CPU-H}$ as *GreenScaler* for

Table 5.10: 99% mean confidence interval (percent of error in *joules*) of tests versus models. Results suggest that for mean confidence interval both the models have similar accuracy. The difference is negligible/small according to Cliff's delta.

Test Heuristic	**Model $_{CPU-H}$ confidence interval**	**Model $_{E-H}$ confidence interval**	**Wilcoxon p**	**Cliff's delta**
CPU-Heuristic	3.60 - 4.50	4.62 - 5.72	2.20e-16	*negligible*
E-Heuristic	5.36 - 6.26	3.02 - 4.03	2.21e-16	*small*
Combined Tests	4.63 - 5.24	3.90 - 4.68	2.20e-16	*negligible*

the following reasons:

1) For mean confidence interval both the models perform similarly. However, for upper error bound *Model* $_{CPU-H}$ is better. *Model* $_{CPU-H}$ and *Model* $_{E-H}$ estimate 94% and 90% apps within 10% error respectively. Also the mean error of the worst 5% estimations with *Model* $_{CPU-H}$ is 13%, in contrast to 16% for *Model* $_{E-H}$.

2) *Model* $_{CPU-H}$ is built on test cases generated with a simple CPU-utilization heuristic. This is much simpler and easier than a complex energy model heuristic. CPU-utilization heuristic requires only capturing CPU time for a test from the Linux file system. On the other hand, the model based heuristic also requires tracing all the invoked system calls by an app by running a separate `strace` program.

3) We were concerned that CPU-utilization tests would ignore other resources. Thus we compare calls to resources (i.e., system calls) with the Kruskal-Wallis test on the number of `recvfrom` (network receive), `sendto` (network send), and `fsync` (file operations) between CPU-utilization and E-heuristic based generated tests. We found that the distributions of these system call counts between the two test sets are not statistically different ($p >> \alpha$ where $\alpha = 0.05$). This suggests that CPU-utilization heuristic based tests exploit other resources similar to the E-heuristic tests. We further examined a sample of the CPU-utilization heuristic based tests and found that accessing other resources can impact CPU utilization. For example, for an on-line video player, CPU utilization is highest when a test starts playing a video

across the network. Thus CPU-utilization based tests *do* use other hardware components.

Findings: Tests based on CPU-utilization heuristic exploit other resources similar to those exercised by the complex E-heuristic based tests. Consequently, energy models built on test cases from both heuristics perform similarly. For simplicity, we recommend CPU-utilization heuristic for generating tests to build software energy models.

5.6 Monkey vs. GreenMonkey

In Section 5.2.5, we mentioned the drawbacks of Monkey and proposed GreenMonkey to mitigate Monkey's problems. Before generating tests and collecting measurements for 472 AndroZoo apps, we conducted a short study to verify if these little changes can indeed improve the performance of energy model building test generation. We selected 100 random apps for training, and 50 different apps for testing models' accuracy. These are two subsets of the 472 AndroZoo apps. With CPU-utilization test generation heuristic, we used *GreenTestGen* with Monkey and GreenMonkey to generate tests for these 150 apps. Two energy models were built: one with Monkey generated tests and another with GreenMonkey. Both the models were tested on the 50 selected apps. Figure 5.3 clearly shows that the energy model built on GreenMonkey generated tests outperforms the energy model built on Monkey generated tests. In case of GreenMonkey based model, 90% of the apps' energy was estimated within only 5% error. On the other hand, only 64% of the apps were estimated within 5% error by the model based on Monkey. Table 5.11 also confirms that this difference is statistically significant.

These early findings led us to generate tests for all the 472 apps with GreenMonkey instead of the Android Monkey.

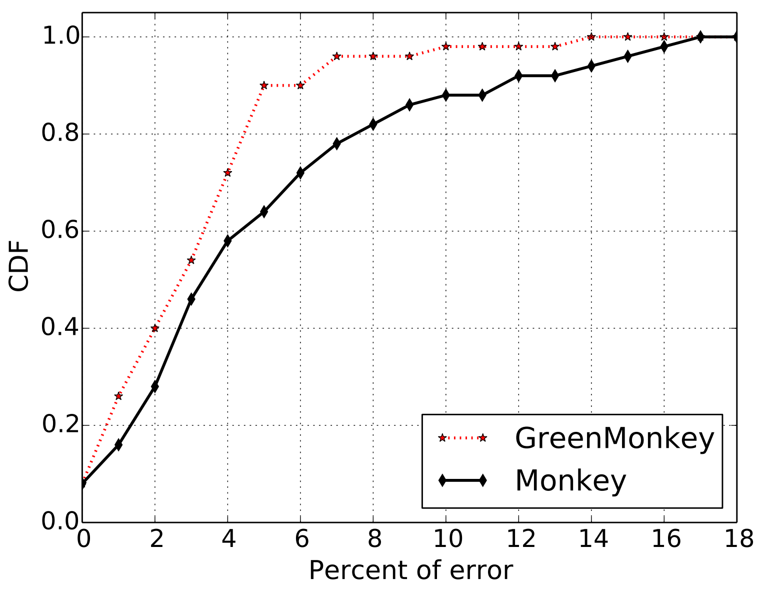


Figure 5.3: Comparing performance between Monkey and GreenMonkey for building energy models. GreenMonkey outperforms Monkey in generating tests that are more suitable for building energy models.

Table 5.11: 99% mean confidence interval (percent of error in joules) of models based on Monkey and GreenMonkey generated tests. Model based on GreenMonkey tests is significantly more accurate than the model based on Monkey generated tests.

Models	**confidence interval**	**Wilcoxon** p
Monkey tests	3.60 - 7.00	7.80e-10
GreenMonkey tests	2.80 - 4.60	7.80e-10

5.7 Evaluating *GreenScaler*

In this section, we evaluate the performance of *GreenScaler* from different aspects: accuracy of *GreenScaler* on randomly generated tests and manually written tests, *GreenScaler*'s ability to explain different sources of energy consumption, and *GreenScaler*'s ability to detect energy regressions. We also observe *GreenScaler*'s sensitivity to the size of changes in SLOC between versions of different apps. This is to check if *GreenScaler* detects energy regression only for large commit sizes. Next, we evaluate *GreenScaler* from developers' perspectives.

5.7.1 Evaluation on randomly generated tests

We compare the performance of *GreenScaler* (based on CPU-utilization heuristic tests) with the previous *GreenOracle* model [48]. We applied all the three machine learning algorithms from Section 5.4.4.

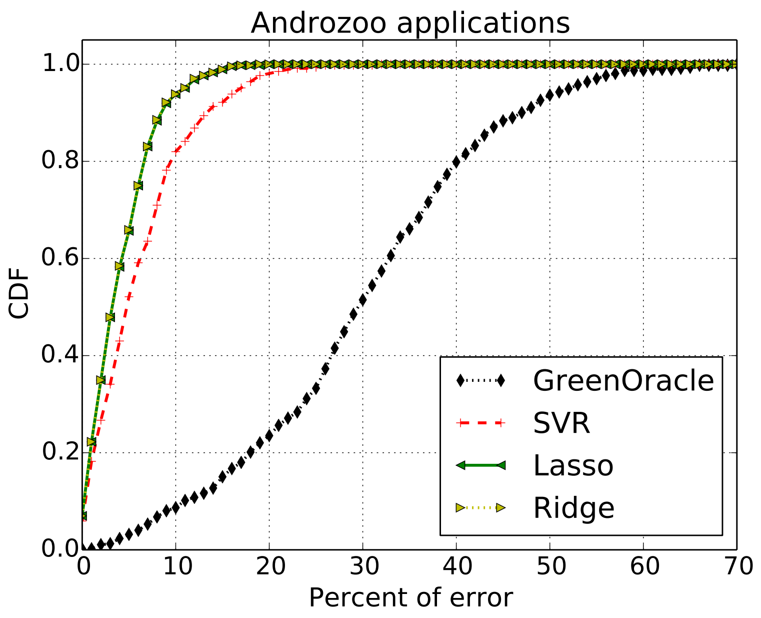


Figure 5.4: *GreenScaler* (Lasso) outperforms *GreenOracle* with very large margin on the 472 AndroZoo apps with randomly generated tests.

GreenScaler has significantly less estimation error than *GreenOracle*. Figure 5.4 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 apps. The significantly worse performance of *GreenOracle* is not surprising, as it was trained only on 24 apps, which led to the selection of inappropriate features with inaccurate coefficients. With the new set of 472 apps and accurate feature set, all the three models outperform *GreenOracle* by a large margin. Lasso and Ridge perform very similarly and show better accuracy than SVR. In case of Lasso, for example, almost 94% of the apps' energy estimations had an upper bound of 10% error. For outliers, the upper bound was only ≈15% error compared to the 70% worst case error with *GreenOracle*. The indistinguishably similar performance of Lasso and Ridge is because of the very small (close to zero) regularization coefficient obtained from the cross validation phase. With no regularization, there is no difference between Lasso and Ridge. With the very small number of features, none of the models overfit the training set, which led to a negligible regularization coefficient. However, during the feature selection phase, Lasso was very different than Ridge and helped us to find a good performing feature set. Therefore, we select Lasso for our *GreenScaler*. In other words, the final *GreenScaler* model is built on CPU-heuristic based test generations with Lasso.

5.7.2 Evaluation on manually written tests

The randomly generated tests with utilization heuristics, although good for energy model building, might not observe any meaningful sequence of actions. The *GreenScaler* model is built on such test cases. So far, we do not know how a model built on random test cases performs on meaningful human written test cases. As a result, it is important to evaluate *GreenScaler*'s performance on human written test cases. Our previous energy model *GreenOracle* [48] is trained on 24 Android apps, with 984 versions in total, where the test cases were written manually. These test cases represent how an average user might interact with these 24 apps, and were written based on the consensus of several computing science grad students from the Software Engineering Research Lab,

Table 5.12: Description of the *GreenOracle* applications [48]. The table shows the 24 apps in the dataset with their types, numbers of versions, and the execution scenarios of the manually written test cases.

Applications	**Type**	**No. of versions**	**Test Scenario**
Firefox	Browser	156	Loads a Wikipedia page and scrolls over the page.
Calculator	Android Calculator	97	Does simple and complex calculations.
Bomber	Bombing game	79	Starts the game and drops bombs at fixed intervals.
Blockinger	Tetris game	74	Moves, rotates blocks randomly
Wikimedia	Wikipedia mobile	58	Searches and loads the Bangladesh page, and scrolls.
Sensor Readout	Read sensor data	37	Reads and draws graphs for different sensors' data.
Memopad	Free-hand Drawing	52	Opens a canvas, draws an object.
Temaki	To do list	66	Creates, updates, searches, and deletes a to-do list.
2048	Puzzle game	44	Tries different moves to solve the problem.
ChromeShell	Browser	50	Opens a web page and scrolls.
Vector Pinball	Pinball game	54	Throws several balls, and plays with them.
Budget	Manage income & expense	59	Calculates by depositing and withdrawing money.
Acrylic Paint	Finger painting	40	Draws objects.
VLC	Video/Audio player	46	Loads and plays a video for 2 mins.
Eye in Sky	Weather app	1	Searches for Edmonton, and looks for temperature.
AndQuote	Reading quotes	21	Reads some famous quotes.
Face Slim	Connect to Facebook	1	Connects with Facebook, and browse the help page.
24game	Arithmetic game	1	plays some random tries.
GnuCash	Money Management	16	Opens an account and saves transactions.
Exodus	Browse 8chan	3	Reads some selected threads.
Agram	Word anagrams	3	Generates single and multiple anagrams.
Paint Electric Sheep	Drawing app	1	Draws objects.
Yelp	Travel & Local app	12	Finds a restaurant and reads users' reviews.
DalvikExplorer	System information	13	Reads system's information.

University of Alberta, Canada. Table 5.12 shows the test scenarios for all the *GreenOracle* apps. The complete and the subsets of *GreenOracle* dataset were used in several published papers on software energy consumption [4], [5], [46], [48], [49], [207].

Encouragingly, *GreenOracle*, even for its own dataset, is outperformed by *GreenScaler* (Figure 5.5). The upper error bound for the new model is 10% (i.e., with Lasso), in contrast to the 13% error with *GreenOracle*. This suggests that *GreenScaler*, although built on measurements from randomly generated test cases, can accurately estimate energy consumption of manually written tests.

5.7.3 Qualitative evaluation of *GreenScaler* model

With the leave-one-out approach, described in Section 5.5, we have developed 472 different linear models with the CPU-utilization test generation heuristic. These models, however, are almost identical, as excluding one app from

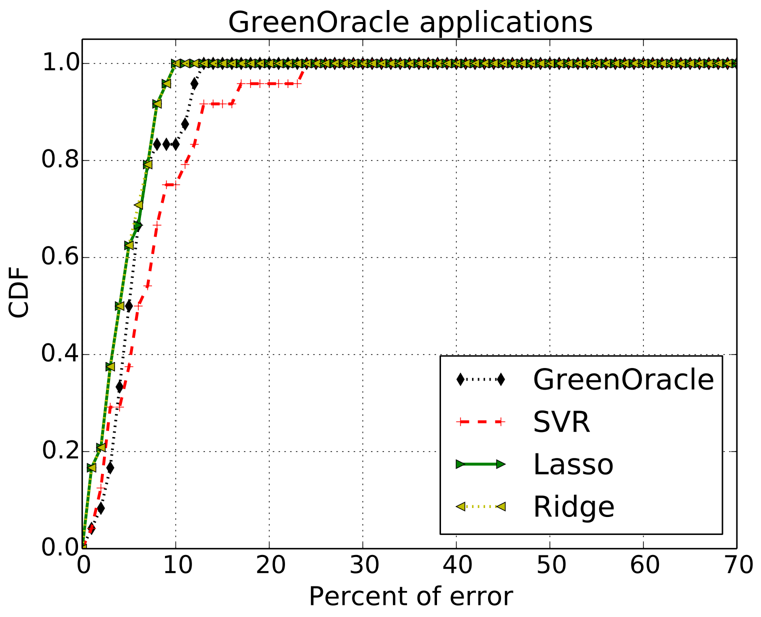


Figure 5.5: *GreenScaler* (Lasso) outperforms *GreenOracle* even on *GreenOracle* dataset. Mean error was considered for apps with multiple versions.

training does not affect a model. We chose one of these models randomly to represent the *GreenScaler*. Table 5.9 shows the final *GreenScaler* energy model (i.e., weights of the selected denormalized features). The accuracy of the model stems from what the table shows. The model suggests that the main sources of energy consumption in Android systems are CPU usage, test duration, screen colour for OLED screen, file operations (`fsync`, `dup`, and `poll`), and data communication (`sendto`, `recvfrom`, and `bind`). The very high coefficients for CPU usage and test duration are similar to the findings of Miranskyy *et al.* [171], who found that energy consumption was often highly correlated with CPU and run-time on database systems. According to the model, transmitting (`sendto`) is more expensive than receiving (`recvfrom`), which is complemented by previous research [169]. Moreover, in terms of pixel colour intensity, blue is the most expensive and green is the least expensive, which is also observed by Dong *et al.* [61].

Findings: *GreenScaler* considers many of the known major sources of energy consumption on an smart-phone. This includes CPU time, context switches, page faults, test duration, and interfaces' colors. *GreenScaler* relies on system calls to estimate energy consumption by other components like disk, networks.

5.7.4 Evaluation on detecting energy regressions

GreenScaler is a model, and similar to any previous energy estimation approaches [92], [184], [194], *GreenScaler* is not 100% accurate. Considering the estimation error, we ask: *will GreenScaler be useful to app developers?* We argue that usefulness is how *GreenScaler* model performs where a developer would actually use it: during the implementation and maintenance of their own app, comparing version against version. In this section, we evaluate *GreenScaler*'s ability to detect energy regression between versions at different levels—two versions separated by a single commit or two versions from two subsequent releases. If *GreenScaler* is successful in detecting energy regression, developers can know if their changes have negative effect on the app's energy consumption.

The main strength of *GreenScaler* is that it maintains similar shape between estimations and ground truths for all the versions of any particular app. We selected four apps from *GreenOracle* dataset that have lots of versions. With multiple versions, we have a separate error distribution function for each app. Figure 5.6 shows that although *GreenScaler* accuracy varies among the apps, the error distribution is very similar among all the versions for the same app.

This observation is significant: it indicates that *GreenScaler* should accurately estimate the energy consumption difference between two versions of an app. To further demonstrate the adeptness of *GreenScaler* for such cases, we select six apps from *GreenOracle* dataset. Unlike other apps, these six apps contain versions with very different energy profiles. Moreover, two of these six apps (Yelp and Agram) contain versions that are actual releases, whereas versions from other four apps are separated by a single commit.

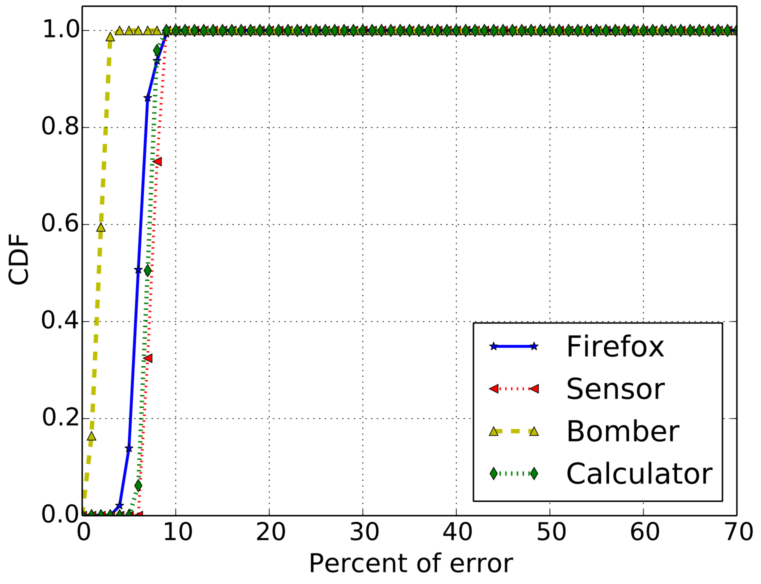
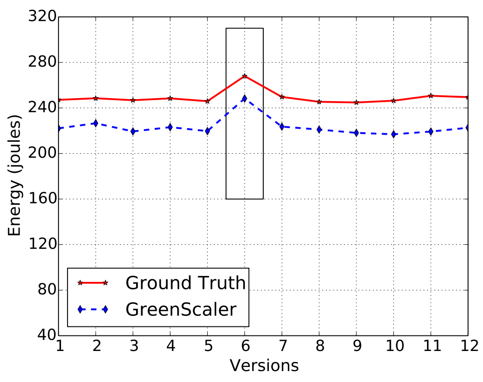


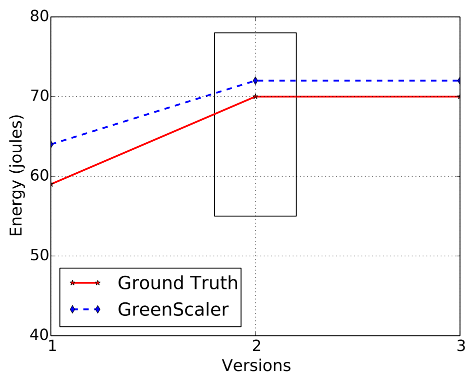
Figure 5.6: *GreenScaler* maintains similar error distribution among different versions of an app.

Figure 5.7 shows that for all the six apps, *GreenScaler* successfully separates out the energy inefficient versions. For Yelp, a travel & local information app, only one version has a very different energy profile. *GreenScaler* distinguished that version accordingly. Memopad, a drawing app, exhibits three interesting energy profiles throughout its life time—it became more and more energy efficient over time in contrast to Agram and Pinball. Our proposed model accurately distinguished those three phases.

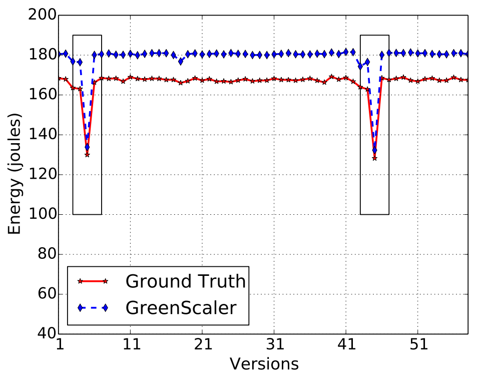
We also investigate if *GreenScaler* can help developers to understand the type of modification that impacted the energy consumption. *GreenScaler* does not locate source code responsible for energy regression. However, the simplistic philosophy of *GreenScaler*—simple counts of different features—helps understanding the type of energy expensive modification. We provide two such examples. For Agram, an app to generate anagrams, the *GreenOracle* dataset contains only three versions. Figure 5.7(b) shows that version 2 and 3 consume more energy than version 1. *GreenScaler* suggests that the number of context switches has increased significantly from version 1 and stays similar to 2 and 3. Our first impression was that code for thread interaction could have



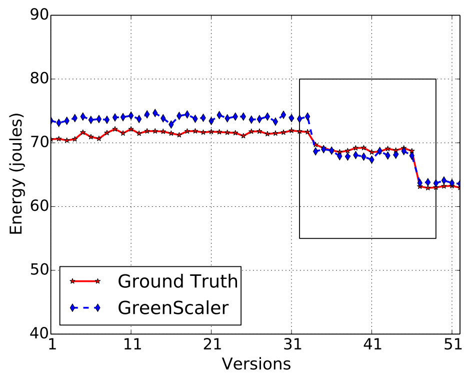
(a) Yelp



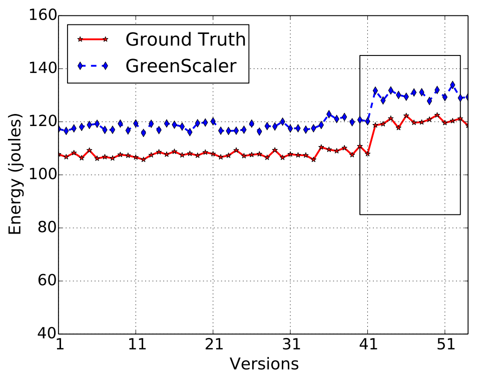
(b) Agram



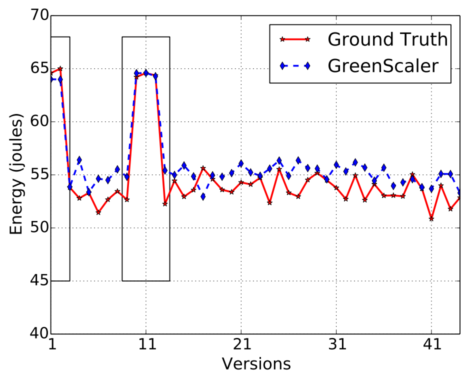
(c) Wikimedia



(d) Memopad



(e) Pinball



(f) Game_2048

Figure 5.7: *GreenScaler*'s efficiency in differentiating between versions with different energy consumption. Versions are sorted based on their committed times. Whenever there is significant energy difference between two versions of the same app, *GreenScaler* detects the difference. Developers can use *Green-Scaler* to check for energy regression before releasing a new version.

been modified. We used `git diff` and found that Java methods for generating anagrams were indeed synchronized. It is well-known that unoptimized `synchronized` methods fight excessively for shared locks, which leads to more context switches and CPU usage [198], [236]. Similarly, we investigated the continuous improvements of Memopad in terms of energy consumption. The only significant difference in our model among all the versions of Memopad was their RGB counts, clearly suggesting the background colour was changed over time. Indeed we found three distinct background colours. White background (the most expensive for OLED screen) was used for versions up to 33, which was modified to more efficient yellow, followed by even more efficient red. This articulates how significant a simple choice of background colour can be for devices with OLED screens, as also observed by previous research [142].

Findings: *GreenScaler* accurately identifies energy inefficient versions for a given app—for versions separated by a single commit and multiple commits (i.e., subsequent releases). Developers can use our tool [47] to evaluate if a new version of an app is more energy expensive than the previous one. In case of energy regression, they can consult *GreenScaler* to understand the type of modification that might have impacted the energy negatively. For example, if a new version calls `fsync` more than before, a developer can focus on file I/O related code.

5.7.5 Accuracy vs. commit size

For our subject apps, *GreenScaler* was successful in detecting energy regression. However, this does not tell us if *GreenScaler* is sensitive enough to detect regression even when the code change is minimal—e.g., when two versions differ by a single line of source code. *Does GreenScaler's accuracy in detecting regression depend on the commit size?* To answer this question, we used five apps (with all the versions) from Section 5.7.4, as these apps show energy regression with source code modifications. We could not use Yelp as we do not have access to its source code.

We calculated the commit sizes (the sum of the number of additions and number of deletions in SLOC) between all the successive versions. We then calculated the differences between mean energy consumption of the successive versions (difference between mean of 10 energy measurements of version x_i and mean of 10 energy measurements of version x_{i+1}, difference between mean of 10 energy measurements of version x_{i+1} and mean of 10 energy measurements of version x_{i+2}, and so on). Similarly, the differences between mean energy estimations (with *GreenScaler*) of all the successive versions were calculated. This way we calculated the absolute estimation error of *GreenScaler* for a commit size in case of energy regression detection. For example, for a commit size of 10, if the difference in mean energy consumption between two versions of an app is 2 joules and the difference in mean estimated energy consumption is 5 joules, this is a 3 joules of estimation error for a commit size 10. We combined the data from all the five apps and show the result in Figure 5.8.

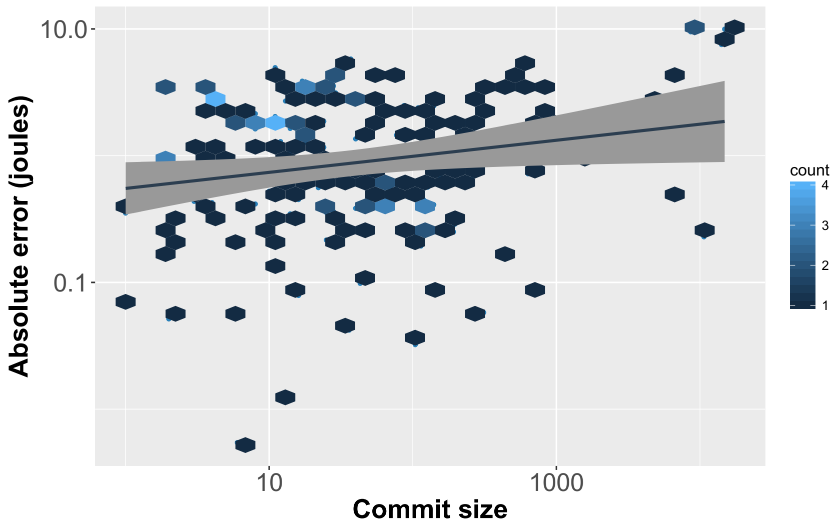


Figure 5.8: *GreenScaler*'s sensitivity to commit size in detecting energy regression. Apparently, there is no (or very weak) relation between accuracy and commit size.

Figure 5.8 suggests that there is no notable relationship between the commit size and *GreenScaler*'s accuracy in detecting energy regression. We also calculated the Kendall's τ correlation coefficient between commit size and absolute error in joules. The coefficient is only 0.08. This implies that *GreenScaler* accuracy does not rely on the commit size. Thus, the model is expected to identify energy regression regardless of the number of changes (small or large) made in the source code. We provide more empirical evidence as follows.

GreenScaler's accuracy in detecting energy regressive API changes

Can *GreenScaler* detect energy regression for a single API change? We consider two case studies with experiments for the evaluation: 1) changing a Java collection data structure, and 2) changing the HTTP library for HTTP requests.

Hasan *et al.* [95] showed that selecting the most energy efficient Java collection can have enormous effect in reducing software energy consumption. In order to evaluate *GreenScaler*'s sensitivity on a single line of code change, we selected four different Java collections: TreeList from Apache Commons Collections (ACC); TreeMap, LinkedHashMap, and LinkedList from Java Collections Framework (JCF). These four collections have different energy profiles [95] and are suitable for our study. We developed an Android app that has only one activity. This activity does only one thing: insert 50,000 elements into a Java collection. This way, we created four different versions of the app with the four collection APIs. For example, in one version the app inserts 50,000 elements into a TreeMap, and in the next version the TreeMap is replaced by the LinkedHashMap—a single line of source code modification.

We measured the actual energy consumption of these four different versions (10 times each) with the *GreenMiner*, and ranked them according to their mean energy consumption. We then compared this ranking with the ranking obtained from our *GreenScaler* model by taking the mean of 10 energy estimations for each collection. Figure 5.9 shows the energy consumption ranking (ranking from actual measurements as well from model's estimation) of the four selected Java collections. In case of energy consumption estimation, *GreenScaler* has the smallest percentage of error (2.63%) for TreeList (ACC) and the largest percentage of error (10.75%) for LinkedHashMap. In spite of these estimation errors, *GreenScaler* is very accurate while detecting energy consumption differences between two successive versions ordered by their ranking, even when the difference is very small. For example, the actual difference between the versions with LinkedList and LinkedHashMap is only 0.5 joules, whereas the difference is 0.56 joules with *GreenScaler*. As we mentioned earlier, this is one of the main strengths of *GreenScaler*—it can identify energy inefficient versions. It is also important to note that although there exists variation in different energy measurements with *GreenMiner* (box-plot), the variation is very small. This is also true for the estimations from *GreenScaler*.

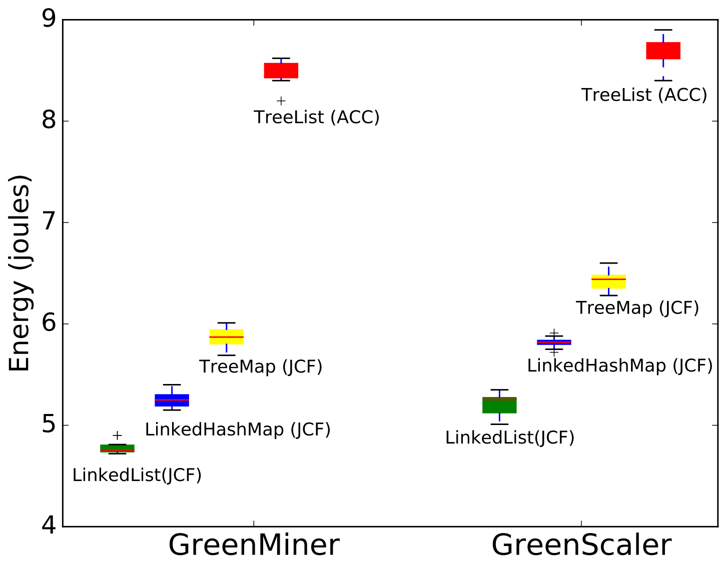


Figure 5.9: Comparing Java collections' energy consumption. Actual energy measurement from *GreenMiner* suggests that TreeList (ACC) is the most energy expensive and LinkedList (JCF) is the least energy expensive for inserting 50,000 integer elements. *Greenscaler* suggests the same and comes up with the exact same ranking.

Collections are typically CPU and memory bound, thus we provide another case study that employs the network. We compare the performance of *GreenScaler* in identifying energy consumption difference between two Java HTTP libraries. In this case, our single activity app downloads the homepage of *CNN.com* 20 times at one second interval. One version of the app uses the `Jsoup` library for the HTTP requests, which is replaced by the `URLConnect` library to produce the second version of the app. Figure 5.10 shows the comparison with 10 energy measurements and 10 energy estimations for each version. *GreenScaler* estimated the energy consumption within around 12% error (by calculating the mean of energy measurements and estimations) in both cases. However, the estimated difference (around 4 joules between the means) is very similar to the ground truths. This difference can be significant for an app that continuously communicates over a network.

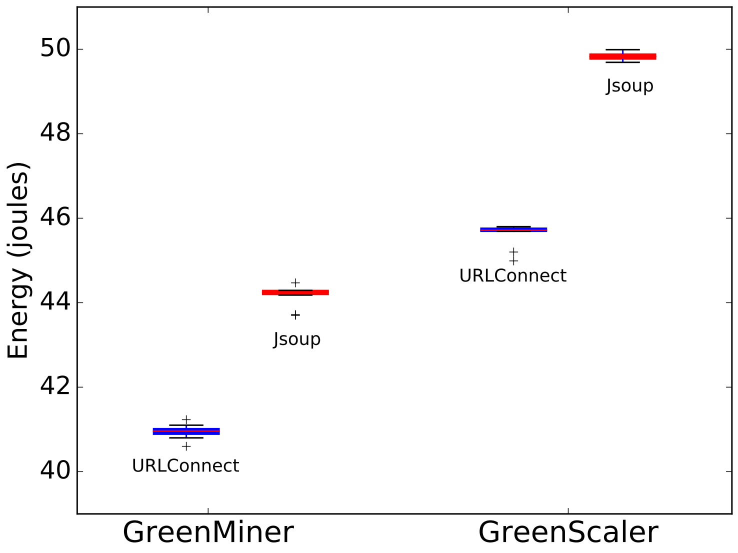


Figure 5.10: Comparing Java download libraries: `Jsoup` and `URLConnect`. For the executed test, *GreenScaler* suggests that version with `Jsoup` consumes around 4 joules more than the version with `URLConnect`. This is very similar to the actual measurements from *GreenMiner*.

GreenScaler's ability to detect energy regressive API changes is significant for energy-aware app developers. Previous studies [138], [144] have found that energy-aware app developers should take extra care while selecting and utilizing APIs that are energy hungry—e.g., APIs for making `HTTP` requests. It is evident that *GreenScaler* can help developers for selecting energy efficient APIs. However, to perform accurate comparison between two APIs, developers should write concise test cases that exercise other irrelevant components the least. A very recent work by Song *et al.* [232] discusses about writing such test cases.

5.7.6 Evaluating ***GreenScaler*** tool from developers' perspectives

One of the end products of this research is the *GreenScaler* tool [47], a model-based tool support that Android developers should be able to use to estimate their apps' energy consumption. We apply a qualitative evaluation of *GreenScaler* as follows.

1) *Accurate and reliable:* Previous research [43] shows that Android tools' evaluation are biased by the apps used for evaluation. As a result, very different performance might be observed when those tools are evaluated on a different set of apps. *GreenScaler* is trained and tested on 472 real-world Android apps. In addition, 24 Android apps with 984 versions were tested with *GreenScaler*. To the best of our knowledge, no previous energy model was tested on such a wide variety of apps.

2) *Easy to use: GreenScaler* works without any need of expensive hardware instrumentation. Developers do not even need to instrument their apps to run with *GreenScaler*. The only required tools to run *GreenScaler*, Android Debug Bridge (`adb`) and `aapt`, come with the Android development framework. *GreenScaler* does not suffer from Android framework compatibility issues. As *GreenScaler* works without the need of source code of an app, it works on both native and non-native Android apps. Previous studies observed that these issues make many of the Android tools unusable [23], [43].

3) *Regression detection: GreenScaler* is accurate on detecting energy regression; a developer can immediately verify if an updated app's version is more energy expensive than the previous one. In case of energy regression, developers can make a trade-off between energy efficiency and other functionalities. Energy-aware developers are willing to sacrifice some other features if that helps in reducing energy consumption [159]. In addition to evaluating source code changes, developers can employ *GreenScaler* model with manually created benchmarks to compare energy consumption of different third-party libraries to select the most efficient one.

5.8 The importance of more apps in training

Will more apps help? Figure 5.11 shows the reduction in error as more apps are used in training. From 472, 50 apps were sampled randomly as test instances. Using the rest of the apps, accuracy of *GreenScaler* is tested using different training sets with different number of training apps (50, 100, 150 and so on). The accuracy can vary for the same training size based on the selected apps—some apps 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 apps. Figure 5.11 shows the combined error distribution for each training size.

Apps with high estimation errors (outliers) exist for all the training sizes. This high error, however, dwindles continuously as we add more apps in our training. In fact, with 400 apps 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 apps in training set, the least number of outliers with 400 apps suggest the possible improvement of *GreenScaler* with adding even more number of apps. Evidently, adding more apps in training improves the upper-error bound of *GreenScaler*. With a large number of apps, we can evaluate the performance of more complex approaches like deep learning. This is why automatic test generation is so useful. *GreenTestGen* enables a process that allows *GreenScaler* to improve continuously.

5.9 Research directions for the software energy research community

We provided empirical evidence on the success of automatic test generation for building software energy models. This observation encourages more research avenues to explore. We provide two such examples.

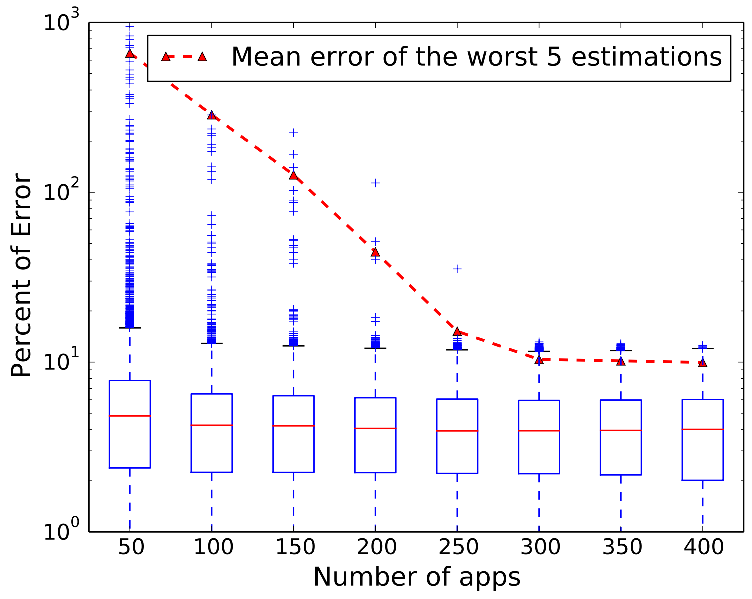


Figure 5.11: Model's accuracy against the number of apps used in training. The accuracy improves with more apps in training. This suggests that we can continuously improve the model by adding more apps with the random test generation process.

Domain specific energy models and deep learning

In this paper, we focused on building a generic single energy model that can be used to estimate energy of Android apps from any category. With random test generation process aided by test heuristics, we can collect measurements for more apps and build domain specific models. Although the AndroZoo database does not contain app categories, we can collect apps from repositories (e.g., F-droid or Google App Store) where categories are available. Instead of building a single model, a cluster of models (separate models for games, communication, utility and other categories) can be built.

Do domain specific models offer better accuracy than a one-for-all model like *GreenScaler*? We do not know the answer, and that is why it could be an interesting future work. Similarly, how about models that are built on resource usage similarity rather than similarity in app category? With more and more apps in training, should we employ deep learning for building energy models? Deep learning usually requires large training data. While techniques like early stopping and dropout layers [79] can help alleviate the problem, the *GreenScalar* methodology provides a method of continually generating and adding more measurements of more apps to achieve an appropriate amount of measurements for deep learning. This leads to a question about how many apps we need to measure for building a deep learning based model?

Building API recommendation systems for energy-aware developers

Previous research [170], [246] focused on building API recommender systems for developers. Developers can select an API based on their requirements when multiple options are available. Different metrics can be used while ranking different APIs: documentation, performance, usability, number of users, and number of reported bugs. These recommender systems do not consider energy efficiency of APIs, which might be crucial for the energy-aware developers. Researcher can use *GreenScaler* to build an API recommender system that includes energy efficiency as one of the performance metrics. Results from Section 5.7.5 clearly show that *GreenScaler* is adept for such studies.

5.10 Dataset

For future researchers, we share our dataset publicly [47]. This dataset contains all the selected tests for the 472 AndroZoo apps with the CPU-utilization based heuristic and *GreenOracle* based heuristic—total 944 tests. These tests can be run on *GreenMiner* in order to explore more research questions and to reproduce our results.

The dataset also contains resource usage and energy consumption for all the apps used for *GreenScaler* and *GreenOracle* models. As we mentioned earlier, each test was run multiple times: 10 times for energy measurements, 10 times for capturing system calls, 10 times for CPU and related measurements, and 5 times for capturing screen shots. The dataset, however, only contains the mean values of each measurements. We have four types of data for the Androzoo apps: 1) AndroZoo apps with resource usage and energy consumption for tests selected with CPU-utilization heuristic that includes all the captured system calls (472 data points), 2) AndroZoo apps with resource usage and energy consumption for tests selected with CPU-utilization heuristic that groups similar system calls together (472 data points). Similarly, there are two more sets of measurements from tests with model-based heuristic (in total, $472 \cdot 2 = 944$ data points from model-based heuristics). We also shared the *GreenOracle* dataset with and without grouping system calls. This dataset can be used not only to reproduce our results, but also to investigate other machine learning and feature selection techniques for building better models.

Data points: A single data point (as used for model building) represents resource usage and energy consumption of an APK (for an app or for a version). So a single data point is:
Number of CPU jiffies, number of context switches,.., test duration, Red, Green, Blue, number of sento syscalls,.., number of dup system calls,.., Energy consumption.
Here, energy consumption is the dependent variable and all others are the independent variables.

5.11 Threats to validity

This section describes the threats to validity of our experiments and results: conclusion validity, construct validity, internal validity, and external validity.

5.11.1 Conclusion validity

Some of our conclusions might not be accurate due to the statistical tests we used. Although the tests we used do not assume anything about the distribution of the data, some of them still have their own set of assumptions. For example, the Kendall's τ test assume that there exists no tied rank in the data. Similarly, the Wilcoxon p assumes that the two distributions under test observe similar variance. The results would be inaccurate in case these assumptions are wrong. We mitigated this threat by comparing the actual error distributions with cumulative distribution functions (CDF). Like previous studies [92], [194], [269], we also evaluated our model's accuracy by calculating the percentage of error relative to the ground truths. This method, however, is asymmetric—the error limit in case of under-estimation is 100% whereas for over-estimation it does not have a boundary [225]. This threat was mitigated by showing the actual ground truths and estimated joules when we evaluated *GreenScaler*'s accuracy in detecting energy regression (section 5.7.4).

5.11.2 Construct validity

Modeling software energy consumption is difficult [159]. For example, a CPU can operate at different frequencies and use different power in these states. Consequently, two different apps in spite of utilizing the CPU for the same duration, might consume different amount of energy based on the triggered CPU frequency level. In our model building process, we relied on the number of CPU jiffies (the time between two clock ticks that can vary) instead of the CPU time directly. Our assumption is that a CPU in higher frequency state would have a different number of CPU jiffies than a CPU in lower frequency state. We do not have direct empirical evidence for the accuracy of such assumption. Instead, we relied on the accuracy of the model built on such assumption.

Similarly, our approach for capturing resource usage by tracing system calls can be criticized. Although capturing different system calls usually indicate the types and amount of resources accessed by an app during a test run, there can be some exceptions—direct memory access (DMA) for example.

We also relied on the claim that system call based models do not suffer from tail energy phenomenon [5], [48], [194]. Although the good accuracy of *GreenScaler* across a large number of apps suggest that such a claim might be true, we do not have direct empirical evidence for that. We do not know if the tests we exercised (including *GreenOracle* dataset) were able to observe any tail energy leaks. For that matter, we do not have evidence that any of our subject apps has an execution path that can produce tail energy leaks. Finding if an app actually has tail energy leak requires manually investigating the app's source code to see if it is not doing batch processing (e.g., not sending a number of packets in a single batch) although batch processing option was available.

5.11.3 Internal validity

The resource usage we collected are mostly related to a process, such as the number of CPU jiffies used by an app, and system call traces. However, we also used global resource usage, such as the number of context switches, the number of global CPU jiffies. These global resources can be affected by other background processes than the process we are interested in. We mitigated this threat by uninstalling all other optional apps that can impact the global resources. *GreenMiner* also uninstalls an app immediately after running it, even when the next run is scheduled for the same app. This ensures that the current run does not use any stored data from a previous run.

The 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 we would hope. There was no direct control over the laboratory temperature that might be harmful for measuring accurate energy consumption. However, INA219's specification [113] suggests that measurements would not be significantly different over the expected laboratory temperature range. We mitigated these two threats by running each scenario 10 times.

The best feature set for modeling Android apps' energy consumption is obtained from a recursive elimination process. There are other feature selection algorithms [130] that might produce a different set of features. However, the selected features with the followed procedure complements earlier findings that CPU, screen, test duration, file operations and network transmissions are the main sources of energy consumption [95], [138], [141], [142], [194]. Similarly, energy model from the CPU-utilization based test generation is compared against the model built on tests based on the *GreenOracle* model. Other energy models might produce better tests.

5.11.4 External validity

External validity can be criticized for using a single version of Android phone and OS. Architecture independent energy models, however, still remain as open research problem, but there is preliminary work on converting energy models between platforms [265].

The three apps—Storyhoard, Klaxxon, and Password Hash—we used for evaluating code coverage are comparatively older than the AndroZoo apps, and might not cover the latest Android coding features and style. The *GreenScaler* model is trained and tested on 472 AndroZoo apps, with the leave-one-out approach. Although it is possible that this model might fail to estimate the energy consumption of a new app, the chance is low. We also mitigated this threat by testing *GreenScaler* model on the *GreenOracle* dataset. However, from our subject apps, we do not know if any of them was using GPS. Even if we had such an app, our phones were immobile, thus would not reflect the actual usage scenario of apps with GPS usage. Also, we only experimented with Wi-Fi, and do not know how a system call based model would perform for other technologies like 3G, and 4G.

Our model building test generation tool created test cases with random events. Such a tool would fail to exercise app's components where human intervention is required—such as providing correct *id* and *password*. Also, the generated tests might not be meaningful—the tests might drive an app in a very different way than a human user. Our objective, however, was not to develop a tool that can exercise every functionalities of every Android apps. We also did not target to produce meaningful tests. Rather, we investigated if an energy model built on randomly generated test cases can accurately estimate energy consumption of human written meaningful tests. By achieving high accuracy on human written tests (i.e., *GreenOracle* dataset), we believe that our objective is indeed achieved.

5.12 Related work

We divide the related previous studies into three areas: modeling software energy consumption, techniques to optimize software energy consumption, and studies related to software energy testing.

5.12.1 Modeling energy consumption

Instruction-based modeling is estimating energy consumption using program instruction cost [92], [219]. The basic problem of these approaches is their rigidness to one particular programming language. Energy estimation for apps without source code is not possible with such approaches. *GreenScaler* applies black-box testing and does not require source code. Instruction-based modeling might also require per-instruction power profile, which is not available for all devices [7]. In contrast, *GreenScaler* relies on features that can be accessed from any Linux-based systems.

The most commonly used approach for modeling software energy consumption is the utilization-based approach [40], [62], [71], [90], [227], [269]. The basic philosophy is that capturing the usage time of a component with its energy consumption allows modeling its energy profile. Such approaches, however, cannot model tail energy leaks [5], [48], [194]—energy consumed by a component even after completing its task before becoming inactive (transition time energy consumption). In our models, however, we did not model energy consumption using active usage period of hardware components. Instead, the cumulative counts of different system calls, CPU jiffy, and other OS-level statistics were used. This automatically alleviated the intricacy of separately modeling tail energy for every hardware components. As a result, in contrast to up to 200% error in estimating *joules* with utilization-based approach [192], our count-based model exhibits only ≈15% error in extreme cases.

Pathak *et al.* [194] proposed a complex Finite State Machine (FSM) based model using system call traces. Aggarwal *et al.* [4], [5] 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 consider screen colour and may profile other components inaccurately. The number of apps used for learning and validation was also very small compared to our dataset.

Nucci *et al.* [184] proposed PETrA, an energy estimation tool that leverages Android tools such as `dmtracedump`. As PETrA is the state-of-the-art for estimating energy consumption of Android systems, we wanted to compare *GreenScaler*'s accuracy with PETrA. Unfortunately, PETrA relies on measurements that are not supported by all Android devices. For example, the `batterystats` program to collect which components were active during an app run, is not supported by the version of Android running on the *GreenMiner*'s Galaxy Nexus phones. We found that the same file could be accessed by using the `batteryinfo` program, but again the provided data was very different than what PETrA expects. We had the same issue with Galaxy Nexus hardware and OS while trying to run other components of PETrA—e.g., `dmtracedump`. We also tried to run PETrA with LG Nexus 5, a close relative of LG Nexus 4 used by PETrA, but failed to produce any results. Again, it was because of the different `batterystats` file. We contacted one of the PETrA authors, and came to know that in order to run PETrA on a different device than LG Nexus 4, we need to re-implement PETrA. The authors are also thinking to make PETrA open source so that such implementation is possible. In contrast to PETrA, *GreenScaler* is already open source and relies on information that are available on any Linux-based systems. Moreover, PETrA heavily relies on the built-in *power_profile.xml* file for getting the current draw for components like CPU, which is not the same as the current from the battery where the voltage is measured. *GreenScaler*, on the other hand, is built on real energy measurements that does not involve any battery information and does not need a battery to run or estimate energy consumption. In worst cases, PE-

TrA's estimation error is more than 50%, especially for apps with high network usage, which is much higher than *GreenScaler*. PETrA also requires manual app instrumentation, which makes it hard to work with hundreds of apps for research purpose. App instrumentation also makes it hard to adopt a tool in a continuous integration system. Also, in contrast to *GreenScaler*, PETrA's performance on detecting energy regression is unknown.

5.12.2 Energy optimization

A process of app recommendation based on energy usage is proposed by Saborido *et al.* [209]. A user can select an energy efficient app when multiple apps with the same functionalities are available. With the availability of such recommendation systems, developers would be forced to develop energy efficient apps. In order to help developers optimize their apps' energy consumption, a significant number of research was dedicated on energy optimization techniques and guidelines.

Wake locks are frequently used by Android developers to continue operations even when a device goes to sleep status [149]. Unfortunately, programmers may write code to acquire wake lock that never releases the lock [10]. Pathak *et al.* [193] observed that 70% of energy bugs are related to wake locks. Much research [10], [28], [149], [192], [195], [250] has been conducted to characterize, detect, and minimize wake lock bugs.

In a previous work [50], we observed that employing `HTTP/2` server can help significantly in reducing clients' energy consumption. For energy efficient logging, we showed in a separate study that developers can combine small log messages and write them together to save energy [46].

As screen colour is very sensitive for OLED screen's energy consumption, tools for automatic colour transformation have been developed [142], [145]. In case of video streaming, pre-fetching has been found helpful to save energy [74]. Job off-loading to a server to save energy was also studied [169], [188]. The overhead associated to data off-loading can be so expensive that it might even worsen energy consumption [169].

For reducing tail energy, bundling I/O operations can be effective [50], [141], [194]. Ad-blockers help reducing energy consumption [204], as advertisements are source of significant energy drains [89]. Some studies concentrated on writing energy efficient code during the development phase [138]. For example, energy profiles of the frequently used Java collection framework were studied [95], [196]. Manotas *et al.* [160] developed a framework for automated selection of energy efficient Java collections.

5.12.3 Energy testing

Research on software energy testing focused on reporting well-known energy-hungry APIs, and locating energy bugs in a system.

Linares-Vásquez *et al.* [144] reported a list of energy-greedy APIs by studying 55 Android apps. The authors concluded that careful selection and application of these selected APIs can lead to more energy efficient apps. This list of energy-greedy APIs are, however, obtained only from 55 apps and might not be complete in listing all energy-hungry APIs. Moreover, energy efficiency is not only effected by the energy greedy APIs. There are other factors (e.g., tail energy [194], code obfuscation [213], code refactoring [212]), that can impact energy consumption. Our *GreenScaler* model does not estimate energy consumption based on counting energy hungry APIs, and thus do not have such limitations.

Jabbarvand *et al.* [117] proposed a test suite minimization approach for energy testing. The authors hypothesized that tests that covers energy-greedy APIs (using the API list from Linares-Vásquez *et al.*) should be enough to locate energy bugs. In our case, however, we needed to generate test cases from the scratch with no existing test suite. Moreover, our objective was to generate test cases for building energy models, not to locate energy bugs. Finally, in contrast to merely stating a hypothesis, we provided empirical evidence that code coverage is not a good heuristic for generating energy model building tests.

5.13 Conclusion and future work

In this paper, we proposed and showed the value of continuous software energy consumption model building through automatic test generation. This process built *GreenScaler*, an ever improving software energy model. The success of random test generation for building energy models is encouraging. More software energy research can be conducted with our simplistic approach. Our model building approach uses measurements of resource usages that are accessible from any Android systems, and is reproducible for other Android devices.

We demonstrated code coverage's irrelevance to power usage. In fact, code coverage correlates more with test run-time than with power usage. Instead of code coverage, we built energy models using automatic test generation by two resource-utilization heuristics: CPU-utilization and E-heuristic (software energy model estimation). We found that simple CPU-utilization heuristic exhibits similar performance to a more complex model based heuristic in generating tests to produce energy models.

There is a clear relationship between the number of apps measured and the upper error-bound of count-based software energy consumption models. By automating formerly manual-labour intensive testing work, we can continuously produce ever more accurate models that can be used by developers with no hardware-based instrumentation. We also demonstrated that these models work well in the relative case whereby version to version the model successfully predicts changes in energy consumption of an app undergoing modification. We shared our *GreenScaler* tool so that developers can have direct feedback on energy consumption without dealing with hardware instrumentation [47].

Future work includes scaling up this app measurement approach even further, so that approaches like deep learning and domain specific energy modelling can be studied. We hope that the idea of energy consumption test heuristics excites other researchers as well, as there is a need for more investigation into test generation heuristics that are good for energy modelling—perhaps other energy models serve as better heuristics than CPU-time heuristics. We used random search, other forms of search such as genetic algorithms might prove fruitful. We do not yet know the bounds of this model, perhaps there is a true saturation point. Questions left unanswered include: "what is the effect of more tests per app on a model", and "what are other features we should be measuring?"

Part II

Guidelines

Chapter 6

Energy Efficiency of HTTP/2

This chapter starts the contributions related to the second objective of this thesis (explained in 1.3.2)—enhancing energy optimization guidelines for the energy-aware developers.

This chapter was published as:

- Shaiful Alam Chowdhury, Varun Sapra, Abram Hindle, "Client-side Energy Efficiency of HTTP/2 for Web and Mobile App Developers", In 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), pages 529-540. March 14-18, 2016. Osaka, Japan [50].

Many of the modern smartphone apps use the Internet for supporting sophisticated features demanded by the users (e.g., multiplayer on-line games). Accessing the Internet (i.e., network operations) is extremely energy expensive and can drain smartphones battery much faster than usual. `HTTP/2` is now the standard Internet protocol, replacing the older `HTTP/1.1`. Yet the energy consumption of the new `HTTP/2` protocol was unknown. *Should the energy-aware app developers be concerned about `HTTP/2`'s energy consumption? Is it better to stay with the older `HTTP/1.1` protocol for saving clients' (e.g., smartphones) energy consumption?* This chapter investigates these questions after experimenting with synthetic and real-world websites.

The takeaways from this chapter include:

- Software energy measurement can be tricky and difficult. For some scenarios, injecting sleep time before the final energy measurement is crucial. If this step is ignored, researchers can come up with a completely wrong conclusion—stating something is more energy efficient when it is not.

- The Transport Layer Security (TLS) makes `HTTPS` more energy expensive than plain `HTTP`. This is because of the handshaking, encryption/decryption, and authentication phases involved with the TLS.

- `HTTP/2` is a free lunch. It never consumes more energy than `HTTP/1.1` with TLS. On the contrary, it always performs better than `HTTP/1.1` when the round trip time is high.

- Energy-aware app developers should definitely switch to `HTTP/2`.

My role in the ***`HTTP/2`*** *study*:

I, with the help of my supervisor, made plans for the methodologies, data collection, experimentation, and evaluations. My co-author Varun Sapra helped me with some of the data collection. I also wrote the `HTTP/2` paper [50] with the guidance of my supervisor.

Impact: According to the Google Scholar website, this paper was cited 35 times (including the preprint). Most of the papers used this paper for discussing the performance issues of `HTTP/2`, while `HTTP/2`'s energy consumption was positively mentioned because of our findings (e.g., [121], [148]). In this paper, one of our auxiliary contributions was to show how energy measurement can go wrong, and how injecting sleep time before measuring energy can alleviate this problem. Injecting sleep time for accurate energy measurement was directly used by Santos *et al.* [108].

Abstract

Recent technological advancements have enabled mobile devices to provide mobile users with substantial capability and accessibility. Energy is evidently one of the most critical resources for such devices; in spite of the substantial gain in popularity of mobile devices, such as smartphones, their utility is severely constrained by the bounded battery capacity. Mobile users are very interested in accessing the Internet although it is one of the most expensive operations in terms of energy and cost.

HTTP/2 has been proposed and accepted as the new standard for supporting the World Wide Web. HTTP/2 is expected to offer better performance, such as reduced page load time. Consequently, from the mobile users point of view, the question arises: does HTTP/2 offer improved energy consumption performance achieving longer battery life?

In this paper, we compare the energy consumption of HTTP/2 with its predecessor (i.e., HTTP/1.1) using a variety of real world and synthetic test scenarios. We also investigate how Transport Layer Security (TLS) impacts the energy consumption of the mobile devices. Our study suggests that Round Trip Time (RTT) is one of the biggest factors in deciding how advantageous HTTP/2 is compared to HTTP/1.1. We conclude that for networks with higher RTTs, HTTP/2 has better energy consumption performance than HTTP/1.1.

6.1 Introduction

In recent years, the popularity of mobile devices (e.g., smartphones, and tablets) has dramatically increased. As of 2014, more than 1.4 billion smartphones were used globally [28], which induced a 70% increase in worldwide mobile data traffic [54]. With the recent technological advancements, there has been an exponential improvement in memory capacity and processing capability of mobile devices. Moreover, these devices come with a wide range of sensors and different I/O components, including digital camera, Wi-Fi, GPS, etc.—thus inspiring the development of more sophisticated mobile applications. These new opportunities, however, come with new challenges: the availability of these devices is severely constrained by their bounded battery capacity. A survey [253] has indicated that a longer battery life is one of the most desired features among smartphone users. Unfortunately, the advancement in battery technology is minimal compared to the improvement in computing abilities, thus amplifying the increasing importance of energy efficient application development [28].

The energy consumption of servers has also become a subject of concern for large data centers—consuming at least one percent of the world's energy [38]. Data centers must cater to the continually increasing demand for storage, networking and computation capabilities. In 2010, 4.3 terawatt-years of energy was consumed within the US by LAN switches and routers [187]. Energy efficiency was reported as one of the pivotal issues even by Google, facing the scale of operations, as cooling becomes a very important operational factor [31]. Another very important aspect of energy consumption is the environment: energy consumption has a detrimental effect on climate change, as most of the electricity is produced by burning fossil fuels [77]. Reportedly, 1000 tonnes of CO_2 is produced every year by the computer energy consumption of mid-sized organizations [101].

With the increased penetration of the mobile devices, the Internet usage on these smartphones is also mounting. According to eMarketer [64], it is expected that Internet access from mobile devices will dominate substantially by 2017. Accessing the Internet, however, is undoubtedly one of the most energy expensive use cases for mobile users [137].

Loading Web pages has become more resource intensive than ever, and this poses challenges to the inefficient HTTP/1.1 protocol which has served the Web for more than 15 years. HTTP/1.1, with only one outstanding request per TCP connection, has become unacceptable for today's Web, as a single page might require around 100 objects to be transferred [238]. HTTP/2—mainly based on SPDY, a protocol proposed and developed by Google [250]—is the second major version of HTTP/1.1 and is expected to overcome the limitations of its predecessor in the contexts of end-user perceived latency, and resource usage [87]. The Internet Engineering Steering Group (IESG) has already approved the final specification of HTTP/2 as of February, 2015 [238]. It is no exaggeration to state that "the future of the Web is HTTP/2" [8].

While HTTP/2 is expected to reduce page load time, we ask if using HTTP/2 improves energy consumption over using HTTP/1.1? In other words, is HTTP/2 going to be more mobile-user-friendly by offering longer battery life? Subsequently, should mobile application developers switch to this new HTTP/2 protocol for developing applications with HTTP requests? A recent study claimed the positive impact on energy consumption through efficient HTTP requests [137]. HTTP/2 is based on the promise of making efficient HTTP requests but the more complicated operations might require more CPU usage, such as dealing with encryption—a requirement in HTTP/2. Will this extra computation harm its energy consumption?

In this paper, we study and compare the energy efficiency of HTTP/1.1 and HTTP/2 on mobile devices using a real hardware based energy measurement system: the *GreenMiner* [102]. Our observations/contributions can be summarized as:

1. Using Transport Layer Security (TLS) incurs more energy consumption than HTTP/1.1 alone.

2. HTTP/2 performs similarly to HTTP/1.1 for very low *round trip time* (RTT).

3. For a significantly higher RTT, HTTP/2 is more energy efficient than HTTP/1.1.

In addition, we show the perils related to software energy measurements. We observed that energy measurement of software can be very tricky and making an incorrect conclusion is very likely in the absence of enough domain knowledge or controls. In such a case, an energy-aware software developer, in spite of having all the required energy measurement equipment, might not be measuring what they intend to measure.

6.2 Background

In this section, we review the evolution of the HTTP protocol and the motivation for HTTP/2. We also define some of the terms that are frequently used in software energy consumption research.

6.2.1 Hyper Text Transfer Protocol (HTTP) and its limitations

Hypertext Transfer Protocol (HTTP) was proposed in 1989 and documented as HTTP v0.9 in 1991 by Tim Berners-Lee, laying out the foundation for modern World Wide Web [254]. In 1997, IETF published HTTP/1.1 [91] as the new improved official standard and more features and fixes were added afterwards: persistent connections, pipelining requests, improved caching mechanisms, chunked transfer encoding, byte serving etc. Users were not only able to request a hypertext resource from the servers but could also request images, Javascript, CSS and other types of resources.

According to HTTP Archive [110], as of April 2015, most Web applications are composed of HTML, images, scripts, CSS, Flash and other elements, making the size of an average page more than 1.9 MB. It can take more than 90 requests over 35 TCP connections to 16 different hosts to fetch all of the resources of a Web application [238]. Although new features were proposed in HTTP/1.1 to handle such Web applications, some of these features suffered from their own limitations. For example, pipelining was never accepted widely among browsers because of the FIFO request-response mechanism, which can potentially lead to the head of line blocking problem resulting in performance degradation [238]. To keep up the performance of Web applications, Web developers have come up with their own techniques like domain sharding—splitting resources across different domains; spriting—e.g., combining a number of images into a single image; in-lining—avoiding sending each image separately; and concatenation of resources—aggregating lots of smaller files (Javascript for example) into a bigger one. These techniques, however, come with their own inherent problems [238].

6.2.2 SPDY and HTTP/2

Google recognized the degrading performance of Web applications [80], and in mid-2009 they announced a new experimental protocol called SPDY [33]. While still retaining the semantics of HTTP/1.1, SPDY introduced a framing layer on top of TLS persistent TCP connections to achieve multiplexing and request prioritization. It allowed SPDY to achieve one of its major design goals to reduce page load time by up to 50% [233]. SPDY reduced the amount of data exchanged through header compression, and features such as server push also helped to reduce latency.

SPDY showed the need and possibility of a new protocol in place of HTTP/1.1 to improve Web performance. SPDY was the basis for the first draft of the HTTP/2 protocol [34]. HTTP/2 is a binary protocol that incorporates the benefits provided by SPDY and adds its own optimization techniques. It uses a new header compression format HPACK to limit its vulnerability to known attacks. HTTP/2 uses Application Layer Protocol Negotiation (ALPN) over a TLS connection as compared to Next Protocol Negotiation (NPN) used by SPDY. However, unlike SPDY, it does not make the use of TLS mandatory [238]. In early 2015, IESG allowed HTTP/2 to be published as the new proposed standard [183].

6.2.3 Power and energy

In this paper we focus on power use and energy consumption induced by a change in workload: switching from HTTP/1.1 to HTTP/2. Power is the rate of doing work or the rate of using energy; energy is defined as the capacity of doing work [5]. In our case, the amount of total energy used by a device within a period is the energy consumption, and energy consumption per second is the power usage. Power is measured in *watts* while energy is measured in *joules*. A task that uses 4 watts of power for 60 seconds, consumes 240 joules of energy. For tasks with the same length of time, mean-watt is often used to reduce noise in the measurement. This difference between power (rate) and energy (aggregate) is important to understand—improving one does not necessarily imply improving the other.

6.2.4 Tail energy

Some components including NIC (Network Interface Card), sdcard, and GPS on many smartphones suffer from tail energy—a component stays in a high power state for sometimes even after finishing its task [5], [192], [194]. This is inefficient as the application consumes energy without doing any useful work in this period. In 3G for example, approximately 60% of the total energy can be wasted only because of this tail energy phenomenon [27], which is a concern for mobile application developers.

6.3 Methodology

6.3.1 GreenMiner

In order to run and capture the energy consumption profiles for HTTP/2 and HTTP/1.1, the *GreenMiner* test bed [102] was used. *GreenMiner*—a continuous testing framework similar to a continuous integration framework but with a focus on energy consumption testing—consists of five basic components: a power supply for the phones (YiHua YH-305D); 4 Raspberry Pi model B computers for test monitoring; 4 Arduino Unos and 4 Adafruit INA219 breakout boards for capturing energy consumption; and 4 Galaxy Nexus phones as the systems under test.

A constant voltage of 4.1V, generated by the YiHua YH-305D power supply, is passed to the Adafruit INA219 breakout board and subsequently goes to the Android phones. The INA219 reports voltage and amperage measurements to the Arduino that aggregates and communicates it to the Raspberry Pi. The Raspberry Pi sets up and monitors tests by initiating the test cases on a phone through ADB shell, and it controls the USB communication power (by using the Arduino Uno). Finally, the collected data (i.e., total energy consumption for a test case) is uploaded to a centralized server.

In order to disable cellular radios and bluetooth, the airplane mode was enabled in each phone and then Wi-Fi was re-enabled so that the phones can access the Internet. The phones were connected to a WPA secured wireless N network located in the same room, and thus ensuring very low variability of Internet access in order to have reliable measurements for our test scripts. The *GreenMiner* is fully described in the prior literature [102], [204].

6.3.2 Writing a test script

In order to emulate a use case for the Android clients, a test script is required. For example, to emulate the use case where a user wants to load the Google home page to search for an item, we need a test script that can load a browser, write `www.google.com` in the address bar, and can press enter to load the webpage. This test can be automated by injecting various touch inputs into the input systems – these events can also be captured during actual use. A sequence of such actions (a test script) represents a specific use case for a user. The *GreenMiner* executes the test script on the actual devices to execute the user actions (e.g., tap, swipe, enter etc.).

6.3.3 Collecting Mozilla Firefox nightly versions

We have selected 10 versions of Mozilla Firefox Nightly (mobile US versions) to conduct our experiments [178]—using more than one Mozilla Firefox Nightly version improves generality and ensures that our results/observations are not contaminated by energy bugs that can be present in a specific version. Nightly versions—also known as Central in contrast to Aurora, Beta, and Release—are committed each day and are used to test the effectiveness of new features before including them in the actual releases [242]. We opted for the Nightly versions so that we could test a constantly changing codebase and avoid single version bugs while improving generality. The versions used in this paper, however, exhibit a stable energy consumption profile without any significant differences in terms of energy consumption.

Of the 10 Firefox versions, 9 versions were from January, 2015 to March, 2015 (three versions from each month with equal time intervals) and one version was from April, 2015. These versions had HTTP/2 support enabled by default while HTTP/1.1 can be enabled by disabling HTTP/2. The test scripts can enable or disable HTTP/2 within the Firefox browser: to test HTTP/1.1, HTTP/2 was disabled. We could not use Chromium in our tests as one cannot force newer Chromium versions to use HTTP/1.1 with TLS when HTTP/2 is enabled, regardless of disabling HTTP/2 in Chromium. *GreenMiner* removes and installs Mozilla Firefox Nightly for each separate test, thus ensuring no caching advantages for any of the runs.

6.3.4 Deploying a HTTP/2 server

Among several implementations of HTTP/2 servers [111], we decided to deploy and experiment with the H2O [185] webserver, located at University of Alberta, Canada. H2O supports both HTTP/1.1 and HTTP/2 thus enabling a fair comparison between the two technologies. Besides, the performance of H2O was found significantly better than other implementations like Nginx [185]. The final version of HTTP/2 specification is also supported including NPN, ALPN, Upgrade and direct negotiation methods; dependency and weight-based prioritization; and server push. For the Gopher Tiles tests (described later) and the Twitter and Google tests, we relied on third-party webservers and webservices. This helped us to measure real world performance; when the page load time varies depending on different network scenarios.

6.3.5 Workload

Our objective is to observe the performance of HTTP/2 compared to HTTP/1.1 with benchmarks that can represent real world scenarios. Recent observations for popular websites suggest that on average 2 MB of data needs to be downloaded in order to load a full page, and on average 100 objects must be downloaded [238]. Previous studies have found that the number of objects can play a key role in SPDY performance—the closest relative of HTTP/2 [250]. Although the evaluation criteria was different (page load time), this would be practical to do the similar for our analysis. Consequently, we experimented with the following benchmarks with varying number of objects and sizes. Table 6.1 shows the summary of our benchmarks.

Table 6.1: Description of the Workloads

	Number of Resources					**Size (KB)**					
	HTML	Image	CSS	JS	Other	HTML	Image	CSS	JS	Other	Total
World Flags	1	238	1	5	1	0.92	1261.87	4.61	117.73	27.47	1412.60
Gopher Tiles	1	180	0	0	1	17.14	165.80	0.00	0.00	0.76	183.70
Google	4	6	1	5	1	162.53	434.03	34.95	840.90	1.18	1473.62
Twitter	1	4	2	3	2	53.37	197.37	125.74	588.43	81.80	1046.73

World flags with fgallery

We installed fgallery [58], a static photo gallery generator, on our own H2O server [185] that shows thumbnails of a set of images installed on the server. For our experiments, images of the world flags were used; a similar benchmark was used by Wang et al. [250]. The fgallery loads all the given images as thumbnails along with the full view of the first flag. The users have the option to view the subsequent flags one after another. Instead of using 50 world flags, we used all the country flags to make the workload heavier. The H2O server does not support HTTP/2 without TLS, leading us to experiment with three different settings: 1) HTTP/1.1 without TLS,[1] 2) HTTP/1.1 with TLS[2] and 3) HTTP/2 with TLS.[3]

Gopher tiles

We also used another HTTP/2 server, developed by using the open-source *Go* programming language, which hosts a grid of 180 tiled images.[4] This demo server enables experiments with added artificial latencies. This is very important for our evaluation, as previous study observed significant performance variations with differing RTTs [189]. We captured the energy consumption of our Android devices for downloading the tiled images with different RTTs for both HTTP/1.1 and HTTP/2. The server, however, does not have TLS option for HTTP/1.1. On the contrary, its HTTP/2 implementation works only with TLS. As a result, we were able to evaluate the performance for only two settings: 1) HTTP/1.1 without TLS and 2) HTTP/2 with TLS.

Google and Twitter

In order to work with real websites, we have selected Google and Twitter for our evaluation because of their adoption of HTTP/2. This type of workload helps to investigate how HTTP/2 reacts for systems that are distributed; it is expected that for such highly accessed servers, Google and Twitter distribute different resources at different nodes, even if not totally at different domains. In contrast to the previous workloads, these two sites do not have access without TLS. This led us to experiment with two settings: 1) HTTP/1.1 with TLS and 2) HTTP/2 with TLS. For both the websites, the data collection period was from 2015-04-18 to 2015-04-19.

[1]`http://pizza.cs.ualberta.ca:1800/`
[2]`https://pizza.cs.ualberta.ca:1801/`
[3]Same as HTTPS but with different browser setting
[4]Gophertiles `https://http2.golang.org/gophertiles` (last accessed: 2015-APR-22)

For Google, all the requests from our Android devices were automatically redirected to *google.ca* and the resource statistics as reported in Table 6.1 are for HTTPS, as of writing Google does not support HTTP/1.1 without TLS.

Twitter requests, on the other hand, were redirected to *mobile.twitter.com*. Interestingly for Twitter, we observed that different resources (e.g., images) were downloaded for our mobile Mozilla Firefox Nightly versions than FireFox or Chrome in our Desktop computers.

6.3.6 Validation

Problems with energy measurement

Aggarwal et al. [5], using the *GreenMiner*, observed that a single measurement for a particular setting could be misleading, as there is variation in the measurements because of several factors unrelated to the application of interest. Consequently, taking the average from at least 10 runs produces more accurate results. In this paper, we repeated each test 20 times for world flags and 15 times for others (after several tests we found that distributions of 15 were indistinguishable from 20 repeats).

GreenMiner enables us to collect energy consumption measurements for different tasks (partitions) in our tests so that we can attribute energy consumption more accurately to a particular task. For example, in our world flags experiment, our script for capturing energy consumption for HTTP/1.1 with TLS has different tasks including app loading, disabling HTTP/2 (to enable HTTP/1.1), and page loading. We are, however, only interested in page load section so that we can compare it with the same section for HTTP/2. The challenge is that tasks, such as configuration, before the page load section for HTTP/1.1 with TLS is very different than HTTP/1.1 without TLS and HTTP/2 with TLS. Mozilla Firefox Nightly versions used in our experiments default to HTTP/2 support, hence forcing to HTTP/1.1 requires more configuration. As a result, for HTTP/2 with TLS experiments our tests do not have to change the browser's configuration: any encrypted request will automatically be a HTTP/2 request. Configuring the browser to use HTTP/1.1 with TLS requires many taps and clicks. These extra inputs can place the CPU into a different power state than if no configuration was done.[5] This is not required for HTTP/1.1 without TLS, as none of the servers used in our study support HTTP/2 without TLS. Consequently, any request without HTTPS will automatically be HTTP/1.1 (without TLS).

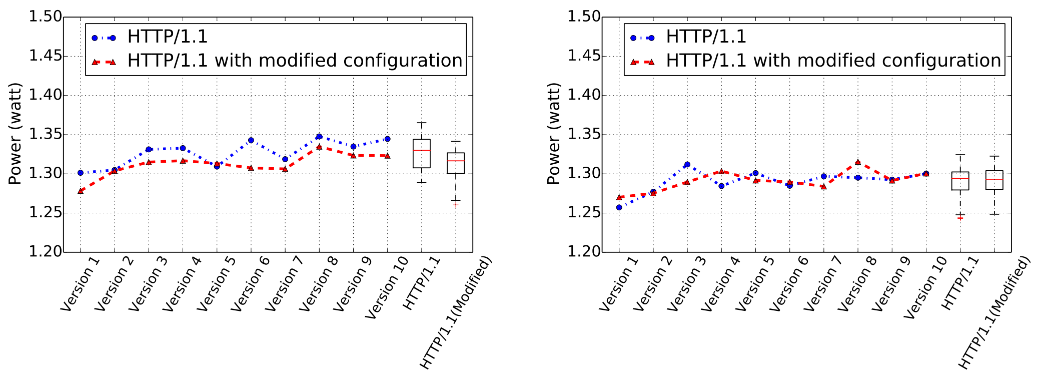
This different sequence of operations before the same page load section is a problem; modifying the about:config page might result in different power states for different components including CPU, screen, and NIC [28], [192], [194]. This could impact the subsequent operations' energy either positively (when components in high power states reduce the execution time significantly and nullify the effect of operating in high power states) or negatively (the reduction in execution time is not significant enough). In either case, our measurements for HTTP/1.1 with TLS would be affected by the previous task's energy consumption leading to an unfair evaluation. In order to verify this hypothesis—to measure how inaccurate the measurement is—we captured the page load energy consumption for the same protocol (HTTP/1.1 without TLS) twice:[6] once without changing the Mozilla Firefox Nightly config file (as we do not need to disable HTTP/2 for unencrypted HTTP/1.1) and another time by changing the config file (disable HTTP/2). These two settings should give us the same average measurement if the later one is not affected by either the different power states of the components or the tail energy.

Test runs were averaged and compared against each other using 2-sided paired t-tests paired by Mozilla Firefox Nightly version. Besides, we observed the effect size by calculating *Cohen's d*.[7] Unfortunately, the small P-value ($<< 0.05$) and the large *Cohen's d* (7.02) suggest that these two settings produce significantly different measurements, thus implying the CPU state

[5]In Mozilla Firefox Nightly about:config, we need to disable *network.http.sdpy.enabled* and *network-.http.sdpy.enabled.http2draft*

[6]one single value is actually the average value of 20 measurements

[7]In order to represent energy consumption by a Mozilla Firefox Nightly version to calculate *Cohen's d*, we took the average of 20 runs.



(a) Waiting time one minute (b) Waiting time two minutes

Figure 6.1: Comparing power usages for the same protocol with different settings

was not consistent.

A potential solution

The challenges we faced in measuring energy consumption led us to configure our test scripts differently: what if we apply a sleep period before accessing the main task, the page load? And how long is required to have accurate measurements? Our hypothesis is that this inactive/idle time would help our Android devices to come to the stable state, i.e., same CPU state.

We experimented with three different periods: 40 seconds, one minute and 2 minutes. The p-value for the 40 seconds period was still lower than 0.05, and slightly higher than 0.05 with 60 seconds of idle time. This P-value, however, becomes very high (0.86) for two minutes of waiting time with very low *Cohen's d* (0.08)—suggesting the very little difference between these two settings comes from randomness and led us to conduct our experiments for world flags with two minutes waiting time before loading the page.

The time-line graphs in Figure 6.1 show the average power usage over 20 runs for each Nightly version for both 1 minute and 2 minutes of waiting time. On the contrary, a box-plot in the Figure represents the distribution of power usages for a specific setting by all the versions across all the 20 runs. Encouragingly, the median value (from the box-plot) is very similar to the average values (from the time-line), implying the accuracy of *GreenMiner* in measuring energy consumption.

Results suggest that with one minute of waiting time the difference is obvious (one setting always consumed less energy than the other), whereas there is no such trend for two minutes of waiting time. Moreover, the variations over different runs for two minutes stable time are significantly lower than for one minute of stable time (box plot in Figure 6.1)—implying better accuracy can be achieved with longer waiting time before executing the actual operations. Interestingly for both settings, the power usage with 1 minute stable time is also higher than 2 minutes stable time. This is not surprising as the CPU is expected to operate in a low power state after having a significant amount of idle time.

We, however, imposed only 1 minute of waiting time for experiments other than world flags, as the previous operations before page load are exactly the same for all the settings. This energy measurement validation approach is crucial, revealing the need for controlling the states of components like the CPU.

6.4 Experiment and result analysis

6.4.1 World flags

Figure 6.2 shows the performance of three different settings for world flags with fgallery: HTTP/1.1 (without TLS), HTTP/1.1 with TLS and HTTP/2 with TLS. Interestingly, HTTP/1.1 and HTTP/2 exhibit very similar performance for our world flags workload when encryption is applied. The high P-value ($>> 0.05$) in Table 6.2 and low *Cohen's d* (< 0.3) between these two settings confirm that the observed small difference comes from randomness in data collection (i.e., the difference is not significant).

This observation is not surprising as previous studies [189] also found that HTTP and SPDY perform very similarly when the RTT is low. And for this experiment with world flags the RTT between our clients and the server was very low (close to 0 ms).

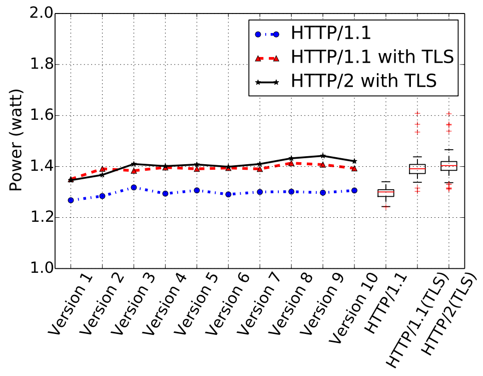


Figure 6.2: Power usage of different settings for world flags with fgallery

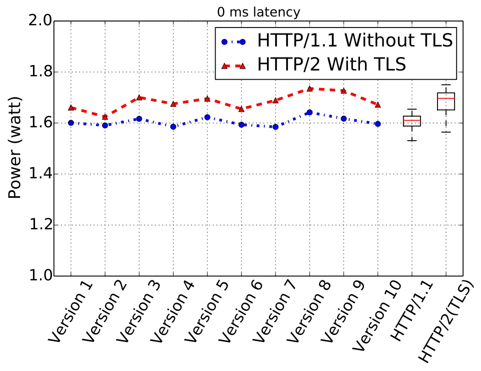
Table 6.2: P-Value for paired t-test among different settings for world flags with fgallery

	HTTP/1.1	**HTTP/1.1 with TLS**	**HTTP/2 with TLS**
HTTP/1.1	1	5e-11	3e-09
HTTP/1.1 with TLS	5e-11	1	0.244
HTTP/2 with TLS	3e-09	0.244	1

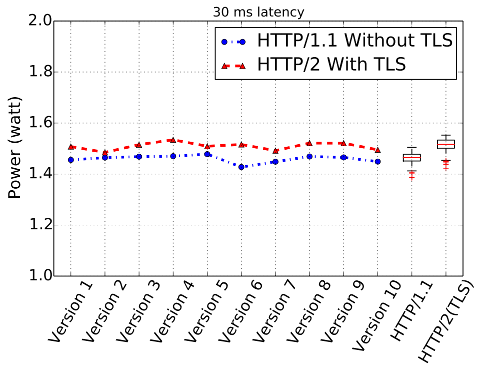
The performance of HTTP/1.1 without encryption is, however, very interesting as it clearly outperforms HTTP/1.1 with TLS although a previous study [78] found improved response time with encrypted messages compared to plain HTTP. Our observations, however, complement the assumptions made by Naylor *et al.* [181]: 1) The required handshaking mechanism for HTTPS consumes energy, which is not present in unencrypted communication; 2) As the browser also takes the responsibility for encryption/decryption, this may lead to more CPU usage and subsequently more energy consumption; 3) The browser verifies if the server, with HTTPS support, is authenticated by examining the server's certificate, which needs more work to be completed.

6.4.2 Gopher tiles

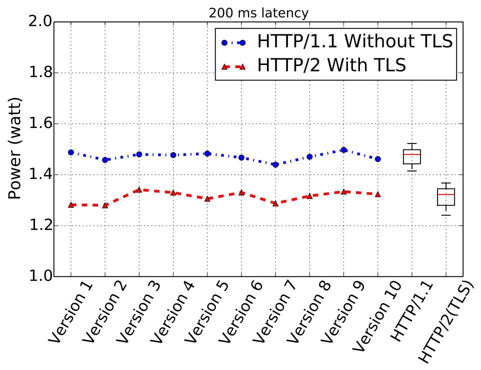
In order to compare the energy consumption of HTTP/2 with HTTP/1.1 for different network scenarios, experiments with Gopher tiles were conducted with different latencies: 0 ms, 30 ms, 200 ms and 1000 ms. The result is shown in Figure 6.3.



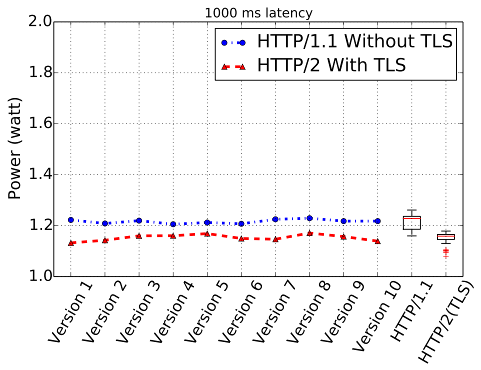
(a) RTT 0 ms



(b) RTT 30 ms



(c) RTT 200 ms



(d) RTT 1000 ms

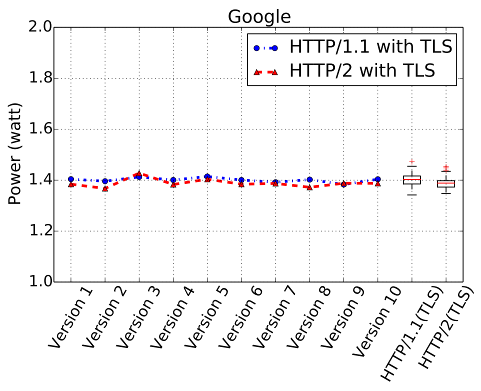
Figure 6.3: Power usage for Gopher tiles with different RTTs

The better performance of HTTP/1.1 than HTTP/2 for low RTT corroborates our findings for world flags: with no latency, HTTP/2 does not offer any improvement over HTTP/1.1, and secured encrypted transmission becomes an overhead which leads to more energy consumption. HTTP/1.1 loses its advantage over HTTP/2 once latency is 30 ms latency or larger. We suspect that HTTP/2 would have outperformed HTTP/1.1 if implemented without TLS. This overhead from TLS, however, becomes negligible for RTT 200 ms and 1000 ms; HTTP/2 significantly outperforms HTTP/1.1 with high RTT. For all the cases, the P-values were low ($<< 0.05$) and *Cohen's d* values—the effect size—were very high.

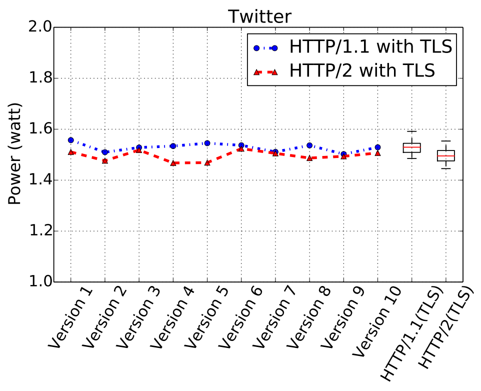
One interesting and useful observation is that although the total energy consumption for both protocols with 1000 ms latency is much higher than the energy consumption with 200 ms (the download time is much longer), the power usage for 200 ms latency is higher than for 1000 ms latency.[8] This could be because of the higher power states of components like CPU and NIC for 200 ms than for 1000 ms, as faster downloading/processing might push the hardware components to more aggressive energy consuming states. This complements previous findings that completion time is not necessarily proportional to a device's energy consumption—different hardware components can have different states of operation; a CPU, for example, can operate at different frequencies, and thus can have different energy profiles in different scenarios [198].

6.4.3 Google and Twitter

We also experimented with Google Search and Twitter home pages—two of the most accessed sites by the Internet users [12]. Instead of experimenting with different types of user interactions in these sites, we only considered the page load times similar to a previous study [250]. Figure 6.4 shows the performance of both protocols with TLS; as mentioned earlier, these two sites automatically redirect requests to HTTPS. Although the P-value (0.028) and



(a) Power usage for Google



(b) Power usage for Twitter

Figure 6.4: Power usage for Google and Twitter

[8] We calculate power by dividing the total energy consumption by the duration.

Cohen's d (0.88, large) suggest that the difference between the two settings for Google is statistically significant, the improvement in power usage reduction for HTTP/2 is very little for Google. In case of Twitter, however, the difference is not only statistically significant (P-value << 0.05 and large *Cohen's d* of 2.1), but the improvement in energy efficiency for HTTP/2 is large. We attribute this behaviour to the differing RTTs for these two sites from our client; the RTT for Google was around 20 ms and was around 80 ms for Twitter. This observation between Google and Twitter is very significant as it reveals how HTTP/2 will affect the mobile users in their everyday lives—if not better, HTTP/2 does not perform worse, at least for these two very top sites.

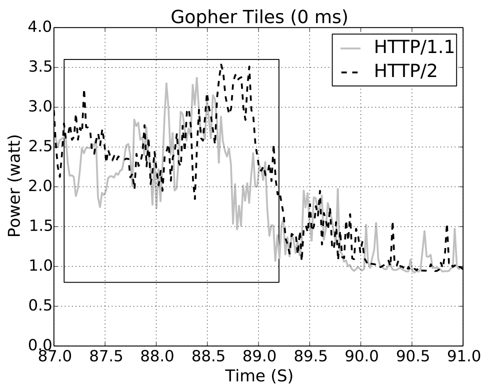
6.5 Discussion

When the RTT is 30ms or more, the difference in energy consumption between HTTP/1.1 and HTTP/2 is obvious. One might argue that the difference, although significant statistically, is not convincing enough to become a deciding factor. This statement is true when only considered for a single mobile device and for a browsing period of one second (as we compared in *watt)*. But when the effect of this difference is considered globally for billions of mobile devices and for average users' browsing time, the energy saving would be colossal. Moreover, in some of our experiments (e.g., Gopher Tiles), in contrast to HTTP/2, HTTP/1.1 was experimented without TLS, otherwise the difference could have been much more conspicuous.

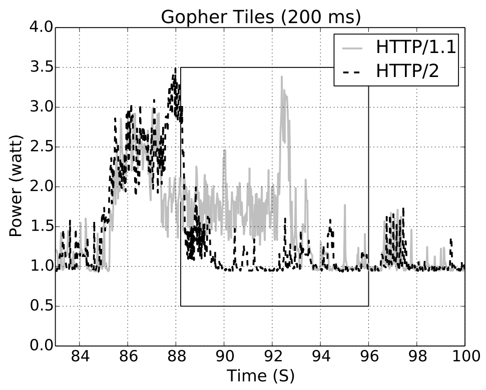
It is encouraging that although HTTP/2 design goals are mainly oriented towards faster page load time, in most cases it also offers better energy performance than its predecessor.

Why is HTTP/2 more energy efficient for larger RTTs?

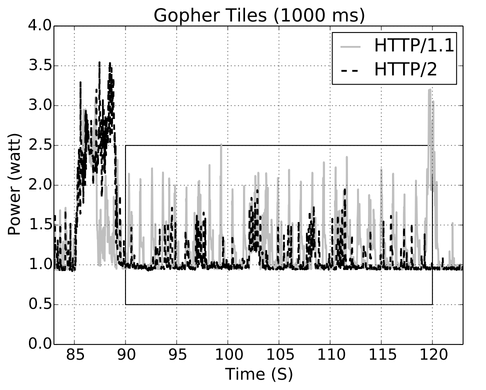
In HTTP/1.1, GET requests are processed in the exact order they are received. Moreover, multiple TCP connections may be established to achieve concurrency. In a high latency network, these factors result in longer waiting periods and also use additional computational resources for establishing TCP connections. The longer waiting times between subsequent network operations produce more tail energy leaks. HTTP/2 solved these issues through the multiplexing of GET requests over a single TCP connection per domain. Also, HTTP/2 eliminated the overhead of transferring redundant header fields and compresses the header metadata through the HPACK algorithm [216]. These factors reduce the network operation times between the client and the server supporting HTTP/2 which is vital in reducing energy consumption. The *prioritization* mechanism of HTTP/2 also helps in faster page loading, which can be a significant factor in reducing energy consumption.



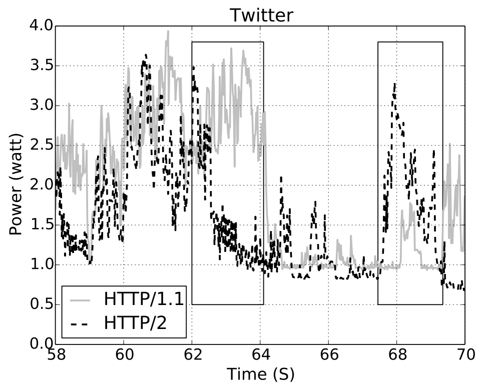
(a) Gopher with 0 ms latency



(b) Gopher with 200 ms latency



(c) Gopher with 1000 ms latency



(d) Twitter

Figure 6.5: Power usage over time

In case of a very low RTT (or no RTT), these factors (e.g., head of line blocking, and header overheads) do not play a significant role both in page loading and power usage, as we observed previously (Figure 6.3). In order to have more insight, we also captured power usage over-time for different settings. Figure 6.5 illustrates the power usage over time for selected configurations during the page load partitions. Figure 6.5 (a), using a single version of Firefox, shows the more power usage by HTTP/2 than HTTP/1.1 at the beginning (highlighted by the rectangle). We believe this is because of the encrypted transmission (HTTPS) employed by HTTP/2 but not by HTTP/1.1. This overhead, however, becomes negligible with the increase in RTT. Although the power usage of HTTP/2 for high latency networks are higher for the very first few seconds, Figure 6.5 (b) and (c), it becomes almost flat afterwards (with a few small spikes). It is easy to notice that (rectangles in b and c) page load time for HTTP/2 is much shorter in high latency networks. On the contrary, HTTP/1.1's network operations stay active for a much longer time and consume much more energy.

It is important to mention that *GreenMiner* does not know when a page load is completed so that it can stop the test immediately. This led us to experiment with fixed test durations for our different settings. For example, for the 1000 ms latency test with Gopher, our predefined test duration for the page load partition was 40 seconds. This is the tightest limit considering the slow page load time of HTTP/1.1. If the actual page load time could have been configured through *GreenMiner*, the difference would be even more significant. This conclusion can also be made from the Twitter experiment. Figure 6.5 (d) shows a very significant spike for HTTP/2 at the end (second rectangle), although the page load was completed much earlier (rectangle 1). During the time highlighted by the second rectangle, HTTP/2 was switching to a different home page image after loading the first one long before, in contrast to HTTP/1.1 which, at that time, was not completely done with loading the first image. This is a clear illustration of the energy efficiency of HTTP/2 over HTTP/1.1.

Web app developers should really consider adopting HTTP/2 as their transport protocol. If RTT is 30ms or more there should be noticeable gains in load time, usability, and energy consumption for their clients. Also, HTTP/2 mechanisms are great examples of making a trade-off between computation and network operations toward producing energy smart systems.

6.6 Threat to validity

We did not experiment with HTTP/2 server push and left it as future work; it would be interesting to see how this feature affects the overall performance. The workload we experimented with also affects our observation; although we backed our results, in most cases, with previous studies. The Mozilla Firefox Nightly versions—although unlikely—can have their own inherent energy bugs and can contaminate the results. The realism of our test-cases can be argued for and against as a balance was to be made between synthetic tests (gopher tiles), realistic tests (world flags) and real-world subjects (Twitter and Google). More websites and more browsers and servers could be tested. Generalization was harmed by using only Mozilla's HTTP/2 clients. Future work should investigate the gain for app developers making HTTP requests from smartphone apps using libraries like Apache HTTPLib.

6.7 Related work

We divide this section into two subsections: studies related to optimization in mobile device energy consumption and performance evaluation of Web protocols.

6.7.1 Mitigation of energy bugs/hotspots in applications

Banerjee *et al.* [28] observed that the main sources of energy consumption in smartphones are the I/O components. As I/O components are accessed by applications through system calls, capturing those system calls can help to find energy bugs or hotspots in a particular application [5], [194]. Pathak *et al.* [192] observed that I/O operations consume more energy partly because of the tail energy phenomenon. According to the authors, this tail energy leak can be mitigated by bundling I/O operations together. Li *et al.* [142] proposed a Color Transformation Scheme for Web applications to find the most energy efficient color scheme while maintaining the enticement and readability at the same time.

Othman *et al.* [188] claimed that up to 20% energy savings is achievable by uploading tasks from mobile devices to fixed servers. A similar study by Miettinen *et al.* [169] suggests that most of the mobile applications were found to be suitable for local processing. This could be the result of limited available resources with such devices, resulting in deficient number of computationally expensive mobile applications. Trestian *et al.* [245] examined the impact of different network related aspects on mobile device's energy consumption in case of video streaming. The authors addressed the impact of several factors on mobile energy efficiency: video quality, selection of TCP or UDP as the transport layer protocol, link quality, and network. A similar study by Gautam *et al.* [74] suggests that applying algorithmic prefetching can help in saving substantial energy of mobile devices. Rasmussen *et al.* [204] found that a system with Ad-blockers, in most cases, is more energy efficient than systems without Ad-blockers.

6.7.2 Performance of web protocols

SPDY Studies: A study [234] on the page load times of the top 100 websites suggests that SPDY can improve page load time by 27-60% over HTTP without SSL and 39-55% with SSL. In a different study by Google [233], SPDY over mobile networks on an Android device was found to improve the page load time by 23%. Contradicting those observations, Erman *et al.* [66] found that unlike wired connections, SPDY doesn't outperform HTTP over cellular networks. Latency in cellular networks can continuously vary due to radio resource connection state machines, and TCP doesn't account for such variability which results in unnecessary re-transmissions. This affects SPDY more due to its use of a single connection.

Wang *et al.* [250] found that multiplexing and longer RTTs help SPDY to achieve an improvement over HTTP. However, the improvement is significantly reduced due to Web page dependencies, browser computations or under high packet loss. Padhye *et al.* [189] compared SPDY and HTTP on a dummy Webpage simulating different network conditions.

HTTP/2 Studies: In HttpWatch [112], the performance of HTTP/2, SPDY and raw HTTPS (HTTP with TLS) protocols were compared using different parameters. Compared to SPDY, request and response header size were found to be smaller for HTTP/2—indicating compression achieved in HTTP/2 using HPACK is more efficient than the DEFLATE algorithm used by SPDY. However, SPDY's response message sizes were smaller as they compressed textual resources.

The compression used by HTTP/2 also allowed headers [228] and images to be smaller. In terms of number of connections, due to multiplexing over a single connection, HTTP/2 and SPDY performed better than raw HTTPS. For page load time, HTTP/2 was consistently found to be better than SPDY (HTTPS performed worst). Centminmod [67] community's administrator benchmarked HTTP/1.1, SPDY 3.1 and HTTP/2 performance on different servers—Nginx, H2O and OpenLiteSpeed depending on the protocols supported by them. For all three servers HTTP/2 and SPDY 3.1 performed better than HTTP/1.1. Between HTTP/2 and SPDY, the performance of HTTP/2 on H2O server was best followed by SPDY/3.1 on Nginx and HTTP/2 on OpenLiteSpeed. Similar results were observed under 3G mobile network. Other studies [8], [76] compared HTTP/2 performance with HTTP/1.1 under different latency conditions and showed the performance benefits of HTTP/2 over HTTP/1.1.

Cherif *et al.*[41] used HTTP/2's server push feature in a *Dynamic Adaptive Streaming* (DASH) session to reduce the initial load time of a video. Loading time under HTTP and HTTPS increased with the increase in RTT, and at an RTT of 300 ms loading time with HTTP/2 outperformed HTTP and HTTPS by 50%. This gain was attributed to the fast increase in TCP receiver window size due to server push in HTTP/2.

6.8 Conclusions and future work

Does HTTP/2 save energy? Yes, when round trip times are above 30ms and when TLS is being used, our tests indicate that HTTP/2 outperforms HTTP/-1.1 with TLS in most scenarios. The Mozilla Firefox Nightly implementation of HTTP/2 consumes less energy than the HTTP/1.1 implementation to do the same work regardless of the webserver used in the tests.

The advantage of HTTP/2 highly depends on the round trip time between the client and the server. HTTP/1.1 becomes expensive, for large number of TCP connections, with large number of objects. This becomes even worse when the RTT is higher, which is common in cellular data networks. HTTP/-2, on the other hand, does not have this problem as it deals with one single connection by incorporating a multiplexing technique. We also observed that accessing the Internet has become more energy expensive for mobile clients after adopting HTTPS as the standard to ensure a user's privacy and security.

We conclude that the web served over HTTP/2 is going to be more mobile user friendly especially on high-latency wireless and Wi-Fi networks, and that energy-aware application developers should adopt HTTP/2.

In this paper, we did not consider the energy consumption of HTTP/2 servers; we only evaluated the energy usage of HTTP/2 from the context of mobile clients. This would be an interesting avenue of research to investigate how energy efficient HTTP/2 is from the server side perspective.

Chapter 7

Energy Consumption of Logging in Android

This chapter is the second contribution related to the second objective of this thesis (explained in 1.3.2)—helping energy-aware developers by providing energy optimization guidelines.

This chapter was published as:

- Shaiful Alam Chowdhury, Silvia Di Nardo, Abram Hindle, Zhen Ming (Jack) Jiang, "An Exploratory Study on Assessing the Energy Impact of Logging on Android Applications", Empirical Software Engineering Journal, Springer, 2017 [46].

Developers frequently employ logging to observe the run-time behaviour of their software. This helps in software maintenance when developers try to fix the bugs they find from the log files. Unfortunately, the energy consumption of logging was unknown. Developers do not know how much they can log without significantly impacting a device's energy consumption. This information is more important for mobile app developers. If logging consumes significant energy and makes battery life shorter, developers would need to know efficient ways of logging. It is also important to know if logged data can be used for understanding apps' energy consumption and for building more accurate energy models. This chapter investigates these issues using the *GreenMiner* energy measurement system with four Samsung Galaxy Nexus phones.

The takeaways from this chapter include:

- Conservative logging, such as less than 10 log messages per second, does not significantly impact an app's energy consumption. Developers do not need to worry about energy consumption for such logging.

- In case of heavy logging (frequent logging, or large log messages), developers can apply bundled logging—combining multiple log messages into one—for reducing energy consumption.

- Some log events (e.g., events related to graphics) are correlated with energy consumption. Developers can look for those events for understanding their apps' energy consumption.

- Although logged events, on their own, are not sufficient to model software energy consumption, they have potential to be used as added features with other important features (e.g., CPU usage, context switches, and system calls) for building accurate energy models.

My role in the logging study:

I, with the help of my supervisor and Dr. Zhen Ming (Jack), made plans for the methodologies, data collection, experimentation, and evaluations. I also wrote a significant part of this logging paper [46] with the guidance of my supervisor and Dr. Zhen Ming (Jack). My co-author Silvia Nardo has been instrumental in this project; she actually developed the benchmark app that was used to answer research question 2 of this chapter.

Impact: As of writing, the published version of this chapter has been cited 10 times (with more than 500 downloads). Cruz *et al.* [57], by mining software repositories and existing energy research, reported 22 common energy bugs and their solutions. One of the reported problems was the logging impact on energy, where they summarized our findings as the potential solution (i.e., striving for infrequent logging). A recent study by Zeng *et al.* [264] on the characteristics of logging practices confirmed our findings: unnecessary and frequent logging should be avoided to save energy consumption.

Abstract

BACKGROUND: Execution logs are debug statements that developers insert into their code. Execution logs are used widely to monitor and diagnose the health of software applications. However, logging comes with costs, as it uses computing resources and can have an impact on an application's performance. Compared with desktop applications, one additional critical computing resource for mobile applications is battery power. Mobile application developers want to deploy energy efficient applications to end users while still maintaining the ability to monitor. Unfortunately, there is no previous work that study the energy impact of logging within mobile applications.

OBJECTIVE: This exploratory study investigates the energy cost of logging in Android applications using *GreenMiner*, an automated energy test-bed for mobile applications.

METHOD: Around 1,000 versions from 24 Android applications (e.g., CALCULATOR, FEEDEX, FIREFOX, and VLC) were tested with logging enabled and disabled. To further investigate the energy impacting factors for logging, controlled experiments on a synthetic application were performed. Each test was conducted multiple times to ensure rigorous measurement.

RESULTS: Our study found that although there is little to no energy impact when logging is enabled for most versions of the studied applications, about 79% (19/24) of the studied applications have at least one version that exhibit medium to large effect sizes in energy consumption when enabling and

disabling logging. To further assess the energy impact of logging, we have conducted a controlled experiment with a synthetic application. We found that the rate of logging and the number of disk flushes are significant factors of energy consumption attributable to logging. Finally, we have examined the relation between the generated OS level execution logs and mobile energy consumption. In addition to the common cross-application log events relevant to garbage collection and graphics systems, some mobile applications also have workload-specific log events that are highly correlated with energy consumption. The regression models built with common log events show mixed performance.

CONCLUSIONS: Mobile application developers do not need to worry about conservative logging (e.g., logs generated at rates of ≤ 10 message per second), as they are not likely to impact energy consumption. Logging has a negligible effect on energy consumption for most of the mobile applications tested. Although logs have been used effectively to diagnose and debug functional problems, it is still an open problem on how to leverage software instrumentation to debug energy problems.

7.1 Introduction

Execution logs are generated by output statements (e.g., `System.out.println` or `printf`) that developers insert into their source code. Execution logs record the run-time behaviour of the application ranging from scenario executions (e.g., “Browsing scenario purchase for user Tom”) to error messages (e.g., “Database deadlock encountered”) and resource utilization (e.g., “20 out 150 worker threads idle”). Software developers, testers and operators leverage logs extensively to monitor the health of their applications [256], to verify the correctness of their tests [123] and to debug execution failures [261], [263]. To cope with these tasks, there are many open source and commercial log analysis and monitoring frameworks available for large-scale server applications (e.g., Chukwa [52], Splunk [235], and logstash [151]).

Excessive logging could cause additional overhead inducing higher resource utilization or worse run-time performance [83]. For example, Google has shown that turning on the full logging would slow down their systems’ run-time by 16.7% [229]. Developers, testers, and system administrators are concerned about the impact of logging on their applications [69]. This is also the case for mobile application developers [119], [120], [126]. Compared with desktop applications, one of the additional critical computing resources for mobile applications is battery power. Mobile application developers (short for *app developers*) want to deploy energy efficient applications to end users while still maintaining the ability to monitor and debug their applications using logs. However, the energy impact of logging on mobile applications is not clear to the developers. When one app developer asked whether logging would drain the battery for Android phones, on the Stack Overflow forum [214], he received three conflicting responses: “yes”, “no”, and “it depends”. This lack of definitive response is similar to the disagreement between practitioners on energy consumption questions observed by Pinto *et al.* [198].

In this paper, we have studied the energy impact of logging on Android applications using the *GreenMiner* [102]. The *GreenMiner* is an automated test-bed for studying the energy consumption of mobile applications. It automatically tests the mobile applications while physically measuring the energy consumption of mobile devices (Android phones). The measurements are automatically reported back to developers and researchers. Using the *GreenMiner* the following three research questions are studied to assess the energy impact of logging on mobile applications:

- **RQ1: What is the difference in energy consumption for Android applications with and without logging?**

  This research question investigates whether the energy consumption of an Android application would be different when enabling and disabling logging. Around 1,000 versions from 24 real-world Android applications, including `Calculator`, `FeedEx`, `Firefox`, and `VLC`, were studied.

- **RQ2: What are the factors impacting the energy consumption of logging on Android applications?**

  This research question aims to identify the important factors in logging that impact software energy consumption. Controlled experiments were conducted to investigate two factors of logging energy consumption: log message rate and log message size. In addition, the relationship between energy consumption and the number of disk flushes was analyzed.

- **RQ3: Is there any relationship between the logging events and the energy consumption of mobile applications?**

  This research question explores the relationship between log events and software energy consumption to see if some log events are more correlated with energy consumption than others. Data analyses, like correlation and multiple linear regression, were carried out to study the relationship between events recorded in logs and mobile energy consumption.

The contributions of this paper are summarized as follows.

1. To the best of our knowledge, this is the first work that proposes a systematic approach to study the energy impact of logging on mobile applications.

2. The findings of this paper were based on an extensive set of measurements/experiments (approximately 70 days of testing time), which includes a wide variety of Android applications with logging enabled and disabled, and a controlled experiment with varying logging rates and message sizes. Each experiment was repeated multiple times to avoid measurement bias and errors.

3. We provide evidence for developers that they need not worry about impacting energy consumption of their mobile applications if they conservatively employ logging.

4. To encourage replication and further study on this important topic, we have disclosed our dataset and source code for our analysis in our replication package. We believe such data can be very useful for software engineering researchers and app developers [205].

The rest of this paper is organized as follows: Section 7.2 provides some background information on logging and the *GreenMiner*. Sections 7.3, 7.4, and 7.5 discuss the research questions RQ1, RQ2, and RQ3, respectively. Section 7.6 discusses the threats to validity. Section 7.7 explains the prior works in the area of mobile energy analysis and execution logs. Section 7.8 concludes this paper.

7.2 Background

This section provides the background information on software logging. It is broken down into three parts. First, we discuss the general approaches for software instrumentation. Then, we explain how logging is realized in Android applications. Finally, a brief description of our automated energy test-bed for mobile applications, *GreenMiner*, is provided.

7.2.1 General approaches for software instrumentation

Execution logs are generated by the instrumentation code that developers insert into the source code. Execution logs are widely available for software systems to support remote issue resolution and to cope with legal compliance [240]. There are three types of instrumentation approaches [252]:

- *Ad-hoc logging:* Developers insert logging in an ad-hoc manner using output methods such as print statements like `System.err.println`. Although this is the easiest approach to instrument, unexpected side effects might happen if one is not careful. For example, logs lines can be garbled if there are multiple threads trying to output log lines to the same file using I/O methods that are not thread-safe.

- *Systematic logging:* The general purpose instrumentation frameworks (e.g., Log4j [20]) address the limitations of the ad-hoc logging approach, as the frameworks support thread-safe logging. In addition, these frameworks provide better control of the types of information outputted. For example, the Log4J framework provides multiple verbosity levels: ERROR, WARN, INFO, DEBUG, and VERBOSE. Each of these verbosity levels can be used for different software development activities. For example, implementation level details can be logged using DEBUG or VERBOSE level, whereas critical errors should be logged under the ERROR level. When deploying a particular application, a verbosity level should be set. For example, if the verbosity level is set to be DEBUG, all the logs instrumented with DEBUG and higher (a.k.a., ERROR, WARN, INFO) are printed whereas lower level logs (a.k.a., VERBOSE) are discarded.

- *Specialized logging:* There are also instrumentation frameworks available to facilitate special purpose logging. For example, it is easier to instrument the system using the ARM (Application Response Measurement) framework [88] to gather performance information from the running application, than to manually instrument the system.

7.2.2 Android logging

Android handles application logs similar to how UNIX handles `syslog` logs. Calls to the Android logging API [20] write logs to a circular buffer in memory. This buffer can be ignored or dumped to disk. Periodic writing of logs can lead to log rotation where old logs are renamed and kept until too many logs are allocated. While running the Android applications, the circular buffer could be filled causing it to periodically dump the logging data to the disk, these dumps or writes are referred to as *disk flushes*. Different mobile phones can have different buffer sizes (e.g., 256 KB or 512 KB). To collect and filter logs for Android applications, there is a utility called `logcat` [150]. Similar to Log4j, the Android API for logging [20] provides multiple verbosity levels. Table 7.1 shows a set of sample log lines from the Android CALCULATOR application. At the beginning of each log line, there is a letter (e.g., "I/" or "D/"). These letters correspond to different verbosity levels. "I/" corresponds to the INFO level logs and "D/" to the DEBUG level logs. The words after the verbosity level show the components where the logs are generated. For example, the first log line is generated by the `ActivityManager` component from the calculator application. The second log line is generated by the `dalvikvm` component, which is the Java Virtual Machine used by the Android operating system.

When released, ERROR, WARN, and INFO level logs are printed for Android applications. Logging for Android applications can also be completely disabled. The following are mechanisms to enable and disable logging for Android applications:

- **Logging Enabled:** First, the log buffer is cleared with the command `logcat -c` [150]. Then, the following command is invoked to redirect the log output of a particular application to a log file (`logcat.txt`) on the SD card.

```
logcat -d | grep -e $PID -e \
net.fred.feedex > /sdcard/logcat.txt
```

Table 7.1: Sample log events from the CALCULATOR application

#	LOG LINES
1	I/ActivityManager(387): START u0 flg=0x10000000 cmp=com.android2.calculator3/.Calculator from pid 21740
2	D/dalvikvm(21750): GC CONCURRENT freed 177K, 3% free 8922K/9128K, paused 2ms+3ms, total 18ms
3	D/libEGL (21750): loaded /vendor/lib/egl/libEGL-POWERVR-SGX540-120.so
4	D/OpenGLRenderer(21750): Enabling debug mode 0
5	I/ActivityManager(387): Displayed com.android2.calculator3/.Calculator: +643ms
6	I/WindowState(387): WIN DEATH: Window(41e6d7c8 u0 com.android2.calculator3/.Calculator)
7	I/ActivityManager(387): Force stopping package com.android2.calculator3 appid=10062 user=0

- **Logging Disabled:** The configuration shown below was added to the `build.prop` file in the `/system` folder of the smartphones. The `/dev/log` folder was removed, along with all its contents.

```
logcat.live = disable
```

7.2.3 GreenMiner

To measure the energy consumption of the selected Android applications, we used the *GreenMiner* framework [102]. The *GreenMiner* has been widely used and accepted in the software energy research community. There are many published works on energy research that used the *GreenMiner* as their energy measurement tool [4], [5], [48], [50], [95], [102], [208].

The *GreenMiner* is a hardware-based energy measurement system that operates 4 Android Galaxy Nexus phones in parallel. Table 7.2 shows the detailed hardware and software specifications for these phones. These phones are used as the systems under test and are controlled by 4 different Raspberry Pi model B computers. Each Pi acts as a test manager for one single phone. It deploys and runs tests, collects energy measurements, and uploads the results to a central server. When a batch of tests are submitted to the *GreenMiner*, one of the four phones are selected randomly to execute a test. In this way four tests can be executed in parallel, enabling the expedient evaluation of experiments, reducing data collection time significantly. It is important to note that after completing a test for an app, the *GreenMiner* uninstalls the app and deletes app related data. This is to make sure that each test run is independent and is not affected by any of the previous test runs.

A constant voltage of 4.1 V, from a YiHua YH-305D power supply, was first passed through an Adafruit INA219 breakout board, and then to an attached phone. The pins, where the phone's battery is usually attached, were wired to receive energy from the power supply. This voltage and amperage were reported to an Arduino Uno by the INA219. The INA219 [113] relies on a shunt resistor to measure changes in amperage. The Arduino Uno then delivers the readings to a Pi through a serial USB connection. Figure 7.1 depicts the innards of the *GreenMiner* (one out of the four identical settings). For a more detailed *GreenMiner* methodology and architecture, please refer to [102], [204].

As energy measurements can vary slightly between different runs for the same tests, it has been a common practice in software energy research to run each test at least 10 times, to achieve acceptable statistical power, and to report average measurements [5], [48], [50], [95].

We measure energy consumption by the integration of power (watts) over time. This energy measurement is called joules (J). Joules are typically stored within a mobile device battery when it charges, and are expended for computation, communication, and peripherals while the device is in operation. 1 joule is 1 watt-second. The phones we use typically consume 0.7 J per second while idle with the screen on, and 1.5 J to 3 J per second when very busy with the screen on. Thus the difference of 10 J between 2 test runs could be due to 14 seconds of runtime or a few seconds of high CPU workload difference. All the measurements of energy consumption in this paper are in joules.

Table 7.2: Specs of the Samsung Galaxy Nexus phones used for the experiments.

Component	Specs
OS	Ice Cream Sandwich, 4.4.2
CPU	Dual-core 1.2 GHz Cortex-A9
GPU	PowerVR SGX540
Memory	16 GB, 1 GB RAM
Display	AMOLED, 4.65 inches
WLAN	Wi-Fi 802.11 a/b/g/n

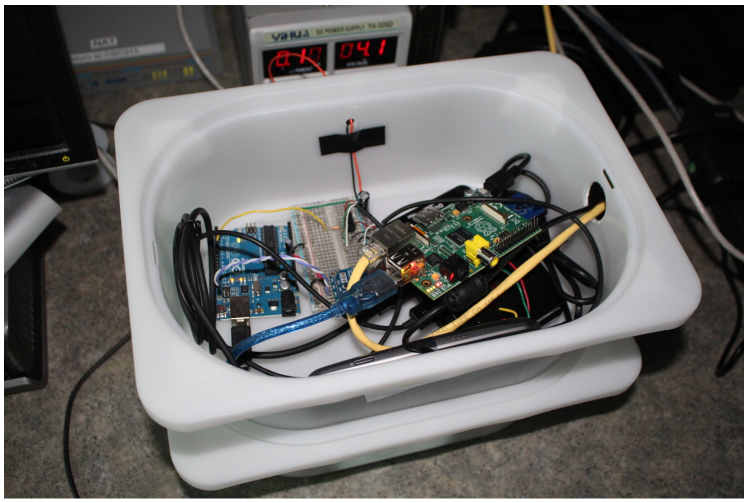


Figure 7.1: *GreenMiner* consists of an Arduino, a breadboard with INA219 chip, a Raspberry Pi, a USB hub, and a Galaxy Nexus phone connected to a Power Supply. Photo used with permission from the *GreenMiner* paper [102].

7.3 RQ1: What is the difference in energy consumption for Android applications with and without logging?

7.3.1 Motivation

On one hand, execution logs can bring insights about the run-time behaviour of mobile applications. On the other hand, should app developers be concerned about the potential energy overhead of logging on their applications? The energy impact of execution logs on 24 real-world Android applications is examined in this section.

7.3.2 Experiments

In order to draw a reliable conclusion on how logging impacts energy consumption of existing apps, we experimented with 24 Android applications from different domains (e.g., Games, Entertainment, Communication, News, and Utility). To capture the general behaviour of each studied application, multiple versions for each application are studied. One version in this paper refers to one binary compiled from one distinct commit from a source code repository, or one compiled binary released by the project. For example, we have studied 46 code commit versions (a.k.a., 46 versions) for the VLC app. The source code for the multiple versions of these 24 applications are part of the *GreenOracle* dataset collected by Chowdhury *et al.* [48].

Software changes over time. The logging changes of some versions of an app may consume much more energy than the other versions. Hence, it is worthwhile to study a number of versions for the same application. Another important aspect of studying the energy impact of logging is that writing test cases manually is difficult. But if we use multiple versions of the same app (versions with identical user interface) then writing a single test script is enough for all the versions. This can enlarge the size of our measurements and thus allows more reliable analysis. Out of the 24 Android applications in *GreenOracle* dataset, we had to exclude YELP from our analysis, as this particular application disables logging internally. We included one more application (FEEDEX with 35 versions) in our dataset as a compensation. Table 7.3 shows the details of the studied Android applications.

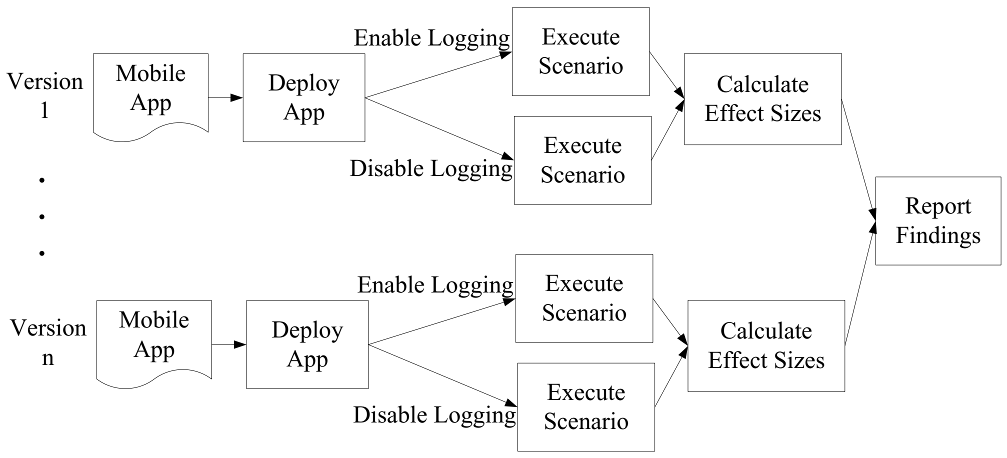


Figure 7.2: Process to Investigate the Energy Impact of Logging (RQ1)

Figure 7.2 illustrates the process for this study. For each version of the selected applications, we measured the energy consumption with logging enabled and disabled with one realistic test case per app. A test scenario for an application, which is automated by a test script written in Android `adb shell`, simulates how an average user would use the application. For example, the FEEDEX first adds RSS feeds from Google News. Then it emulates a normal user opening and reading the first two RSS feeds. The CALCULATOR application test converts miles to kilo-meters, calculates tax amounts, and solves an equation using the quadratic formula.

Table 7.4 shows the list of test scenarios for all the applications. The test scenarios and the test duration are the same across different versions of the same application. For example, a single FEEDEX test (with or without logging), lasts for 100 seconds. Table 7.4 also shows the average number of log lines per test run, the average test duration in seconds, the average logging rate (events per second), and the joules consumed with logcat enabled (logging enabled) and logcat disabled (logging disabled).

For both logging conditions (logging enabled and disabled), the experi-

Table 7.3: The applications under test, selected from the *GreenOracle* [48] dataset.

Categories	App Names	App Descriptions	# of Versions and Committed time	Repo
Games	2048	Puzzle game	44 (03/2014 - 08/2015)	GitHub
	24game	Arithmetic game	1 (01/2015 - 01/2015)	F-Droid
	Agram	Anagrams	3 (03/2015 - 10/2015)	F-Droid
	Blockinger	Tetris game	74 (04/2013 - 08/2013)	GitHub
	Bomber	Bombing game	79 (05/2012 - 11/2012)	GitHub
	Vector Pinball	Pinball game	54 (06/2011 - 03/2015)	GitHub
News	FeedEx	Reading news feeds	35 (05/2013 - 04/2014)	GitHub
	Exodus	Browse 8chan	3 (01/2010 - 04/2015)	GitHub
	Eye in the Sky	Weather app	1 (09/2015 - 09/2015)	Google Play
Entertainment	Acrylic Paint	Finger painting	40 (03/2012 - 09/2015)	GitHub
	Memopad	Free-hand Drawing	52 (10/2011 - 02/2012)	GitHub
	Paint Electric Sheep	Drawing app	1 (09/2015 - 09/2015)	Google Play
	VLC	Video player	46 (04/2014 - 06/2014)	GitHub
References	AndQuote	Reading quotes	21 (07/2012 - 06/2013)	GitHub
	Wikimedia	Wikipedia mobile	58 (08/2015 - 09/2015)	GitHub
Communication	ChromeShell	Web Browser	50 (03/2015 - 03/2015)	APK repository
	Face Slim	Connect to Facebook	1 (11/2015 - 11/2015)	F-Droid
	Firefox	Web browser	156 (08/2011 - 08/2013)	APK repository
Business	Budget	Manage income/expenses	59 (08/2013 - 08/2014)	GitHub
	Calculator	Calculations	97 (01/2013 - 05/2013)	GitHub
	GnuCash	Money Management	16 (05/2014 - 08/2015)	GitHub
	Temaki	To do list	66 (09/2013 - 07/2014)	GitHub
System Utilities	DalvikExplorer	System information	13 (06/2012 - 01/2014)	code.google
	Sensor Readout	Read sensor data	37 (03/2012 - 03/2014)	GitHub

ments were repeated 10 times (for each version) to address measurement error and random noise [75], [129]. For an approximate average test duration of five minutes (including uploading data to a server after each test), it took around 70 days to run and collect all the measurements from *GreenMiner*. All of these measurements were then used to compare the energy consumption between logging enabled and disabled.

7.3.3 Analysis

For the logging enabled tests, the log files have on average 142 log lines, ranging from 12 messages to 1080 messages per test run. The average test duration can last from 52 seconds (AndQuote) to 210 seconds (FireFox). We will perform a two-step analysis on the energy measurement data. First, we will perform a hypothesis testing to examine whether the energy consumption with and without logging for each version of the app is different. Then we will study the magnitude of the differences (a.k.a., effect sizes) to help quantify the size of the differences.

Comparing the Differences Between Two Groups: Some of the measured energy distributions are not always normally distributed, according to the Shapiro-Wilk normality test. Hence, we will use non-parametric tests throughout this paper. Different from parametric tests, non-parametric tests do not have any underlying assumptions of the distribution of the data being analyzed. For each version of the mobile applications, the Wilcoxon Rank Sum test is performed to check whether the differences in energy consumption between the cases of logging enabled and disabled are statistically significant. Table 7.5 shows the results of these tests.

Given $\alpha = 0.05$, $p \leq 0.05$ means that there is a statistical difference in the energy consumption between the logging enabled and logging disabled tests, whereas $p > 0.05$ means otherwise. We also correct for multiple comparisons/hypotheses using the Benjamini and Hochberg method [35] which attempts to control the false discovery rate. Most applications (e.g., 2048 and AndQuote) do not have statistical differences in terms of energy consump-

Table 7.4: Test scenarios and test results for the selected Android applications.

App Names	Test Scenarios	Avg # of log lines	Avg Test Duration	Avg Rate of Logging	Avg Energy(J) with logcat Enabled	Avg Energy(J) with logcat Disabled
2048	Makes some random moves	15.737	60.008	0.262	58.369	59.057
24Game	Randomly tries different numbers	110.000	80.014	1.375	85.816	84.407
Acrylic Paint	Draws a hexagon with legs	24.621	95.011	0.259	82.838	83.998
Agram	Generates anagrams (single and multiple)	46.447	77.006	0.603	75.299	74.985
AndQuote	Reads a series of famous quotes	24.265	52.003	0.467	44.671	44.473
Blockinger	Repositions/rotates blocks randomly	58.715	150.002	0.391	197.315	197.984
Bomber	Drops bombs at fixed intervals	194.091	130.008	1.493	170.483	170.826
Budget	Inserts and calculates expenses	101.684	125.010	0.813	113.017	113.007
Calculator	Converts units, calculates taxes, and solves equations	24.413	125.008	0.195	107.781	107.062
ChromeShell	Opens a webpage and scrolls	153.224	100.010	1.532	106.621	107.049
DalvikExplorer	Reads the system's information	20.799	80.004	0.260	65.750	65.591
Exodus	Reads threads from different topics	239.297	84.012	2.848	96.981	96.001
Eye in the Sky	Looks for the current temperature in Edmonton	182.583	130.008	1.404	116.768	119.310
Face Slim	Connects to Facebook homepage and access the help page	24.500	60.009	0.408	65.935	66.571
FeedEx	Adds and reads feeds from Google News	94.451	100.000	0.945	95.430	92.956
Firefox	Opens a webpage and scrolls	75.325	210.004	0.359	213.679	211.544
GnuCash	Creates an account and saves transactions	90.810	75.012	1.211	76.150	76.713
Memopad	Draws a hexagon with legs	17.966	95.011	0.189	79.649	79.908
Paint Electric Sheep	Draws a hexagon with legs	30.000	60.007	0.500	56.296	57.601
Sensor Readout	Shows graphs for different sensor reads	166.220	182.998	0.908	176.278	177.226
Temaki	Makes a TODO list, updates and deletes the list	12.244	75.010	0.163	72.373	72.189
Vector Pinball	Throws the ball several times and tires to hit the ball randomly	17.340	120.009	0.144	116.359	116.700
VLC	Plays a fireworks `.3gp` video	1,079.676	110.010	9.814	116.464	117.460
Wikimedia	Searches for the Bangladesh page and scrolls	169.783	120.011	1.415	160.015	160.171

Table 7.5: Wilcoxon Rank Sum Tests ($\alpha = 0.05$) comparing energy consumption between logging enabled versus disabled per version. $p \leq 0.05$ means that there is a statistically significant difference in the energy consumption between logging enabled and disabled, whereas $p > 0.05$ means otherwise. Cliff's δ magnitude across applications versions is from [103].

App Names	% versions with $p \leq 0.05$	Mean Cliff's δ	Effect Sizes % Negligible	% Small	% Medium	% Large
2048	0.000	−0.165	40.909	31.818	20.455	6.818
24game	0.000	−0.077	100.000	0.000	0.000	0.000
Acrylic Paint	7.500	−0.439	5.000	17.500	35.000	42.500
Agram	0.000	0.038	33.333	66.667	0.000	0.000
AndQuote	0.000	0.127	47.619	23.810	19.048	9.524
Blockinger	0.000	−0.085	41.892	37.838	14.865	5.405
Bomber	0.000	−0.120	34.177	46.835	13.924	5.063
Budget	0.000	−0.080	40.678	44.068	8.475	6.780
Calculator	3.093	0.352	21.649	22.680	22.680	32.990
ChromeShell	0.000	−0.159	32.000	42.000	22.000	4.000
DalvikExplorer	0.000	−0.073	53.846	38.462	0.000	7.692
Exodus	0.000	0.340	33.333	0.000	33.333	33.333
Eye in the Sky	0.000	−0.375	0.000	0.000	100.000	0.000
Face Slim	0.000	−0.319	0.000	100.000	0.000	0.000
FeedEx	54.286	0.612	8.571	5.714	14.286	71.429
Firefox	0.000	0.152	34.615	37.179	19.872	8.333
GnuCash	0.000	−0.147	37.500	18.750	37.500	6.250
Memopad	0.000	−0.187	34.615	36.538	17.308	11.538
Paint Electric Sheep	0.000	−0.597	0.000	0.000	0.000	100.000
Sensor Readout	0.000	−0.085	43.243	37.838	16.216	2.703
Temaki	0.000	−0.028	43.939	39.394	12.121	4.545
Vector Pinball	0.000	−0.047	40.741	35.185	20.370	3.704
VLC	4.348	−0.294	15.217	45.652	13.043	26.087
Wikimedia	0.000	−0.123	43.103	36.207	12.069	8.621
Overall (paired)	$p = 0.3748$		0.0139 (Negligible)			

tion in any of their versions between the logging enabled and disabled tests. However, for some other applications (e.g., FEEDEX and ACRYLIC PAINT), many of their versions exhibit statistical differences between the logging enabled and disabled tests. Figure 7.3 depicts the p-values per application per version comparing logging enabled (logcat enabled) and logging disabled tests. Only 4 out of 24 applications exhibit cases where their energy consumption in the logging enabled and disabled tests are statistically significantly different after correction for multiple hypotheses.

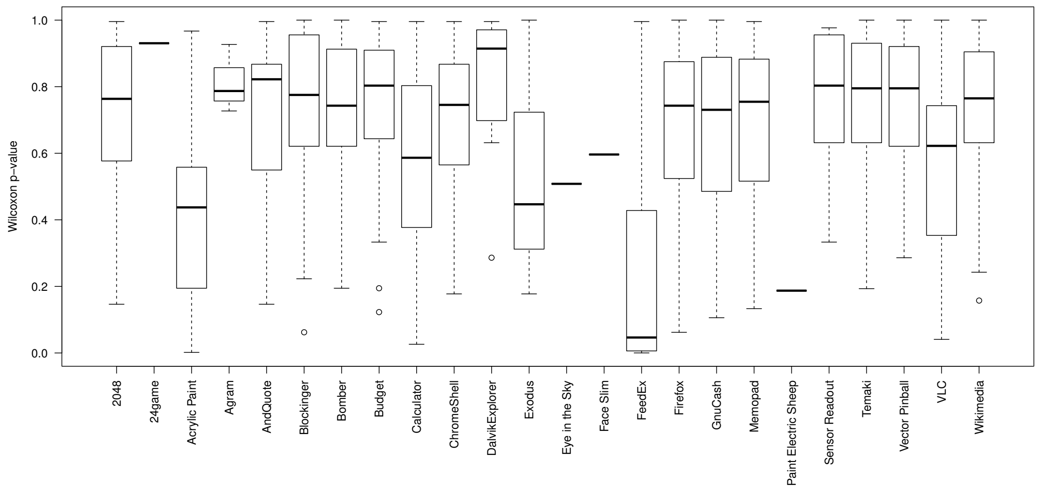


Figure 7.3: Wilcoxon Rank Sum *p*-values per application of energy consumed with logging and without logging. *p*-values less than 0.05 indicate that logging enabled and logging disabled consumed different amounts of energy. *p*-values were corrected for multiple hypotheses using Benjamini and Hochberg correction [35]

Effect Sizes: Although the Wilcoxon rank sum test can examine whether there is a statistical difference in terms of energy consumption between the logging enabled and disabled tests, it cannot quantify the magnitude of the differences. Hence, Cliff's δ (Cliff's delta) is used to calculate the differences of energy consumption for logging enabled and disabled tests. Cliff's δ is a non-parametric effect size measure that quantifies the proportional difference (or dominance) between two sets of data [206]. Cliff's δ has four categories: negligible effect, small effect, medium effect, and large effect. Effect sizes, which can be applied regardless of significance of a T-test or a Wilcoxon rank sum test, is used to characterized the observed differences of the effect in the measurements [244]. Reporting effect size is recommended in cases where there is not enough statistical power [182]. For instance given the number of repeated tests and given the number of hypotheses—how many times we repeated a statistical test—the critical value will be low. This means in order to be conservative enough to reduce false positive rates the p-value correction will make the multiple Wilcoxon rank sum tests quite conservative. This effectively turns the Wilcoxon rank sum test into a measure of sample size, but the effect still remains. Thus we report effect sizes to give the reader an idea of the differences between logging and not logging within the data, regardless of statistical significance.

Table 7.5 tabulates the effect size values, Cliff's δ, for different versions of the Android applications. For example, in Table 7.5, the values of Cliff's δ show that 21% of the versions of the CALCULATOR application have negligible effect, 23% of versions with small effect, another 23% versions with medium effect, and the remaining 33% of versions with large effect. In addition to the CALCULATOR application, there are five other applications that have more than half of their versions exhibit medium to large effect sizes.

Thus based on these observations, we want to statistically verify the effect of logging on applications. Between applications, aggregated by averaging joules across versions, we find that with the paired Wilcoxon signed rank test there is no statistically significant difference between enabling and disabling logcat across these applications ($p = 0.3748$ and $p > \alpha$). The effect size, over all applications, according to Cliff's δ is negligible (0.0139). The paired Wilcoxon signed rank test is used because the samples are related and paired (e.g., mean joules of FIREFOX with logging, and mean joules of FIREFOX without logging).

The results show that the differences in energy consumption are not statistically significant for most versions of the studied 24 applications. Furthermore, within the same applications we find that the effect of enabling or disabling logging is typically statistically insignificant and of negligible to small magnitude. However, 79% (19/24) of the studied applications have at least one version with medium to large effect sizes in terms of the differences of energy consumption when enabling and disabling logs.

In order to have more insight into the impact of logging on energy consumption, we select FeedEx, an application in our dataset, which shows not only statistically different energy measurements between logging enabled and disabled tests (in 54% versions), but also have 71% versions with large effect size. For instance, there is a big difference (≈ 10 joules) in terms of energy consumption between logging enabled and disabled tests for the version 1.6.0. Compared to the previous versions, there were 178 more Dalvikvm *WAIT_FOR-_CONCURRENT_GC* log lines and 224 more Dalvikvm *GC_CONCURRENT* log lines. These logs are related to the memory management of the applications. This version of the FeedEx app, seemed to suffer from memory bloat issues and produces a larger log file than its predecessor.

Figure 7.4 shows the energy consumption of FeedEx over time, for both logging enabled and disabled tests. It is clear that the later versions are more energy inefficient than their predecessors. Figure 7.5 shows the energy consumption for the FeedEx versions against the number of log lines. With few exceptions, we observe a monotonous increase in energy consumption with the increase in logging. This suggests that with more information in log files, one could investigate what types of log events can impact the energy consumption, and thus motivated us for RQ3. However, the energy differences between logging and no logging do not show any consistent pattern with the increase in log messages. With randomly selected real-world applications, there can be many factors that can significantly impact the energy consumption [48], [194], [249] of Android applications. Such uncontrolled tests can indicate if logging matters or not, but cannot offer an accurate estimation of the impact of logging on energy consumption.

These results indicate the need for a more controlled experiment—to show how much logging can be harmful in terms of energy consumption. We did not have control over the development of these applications and their use of

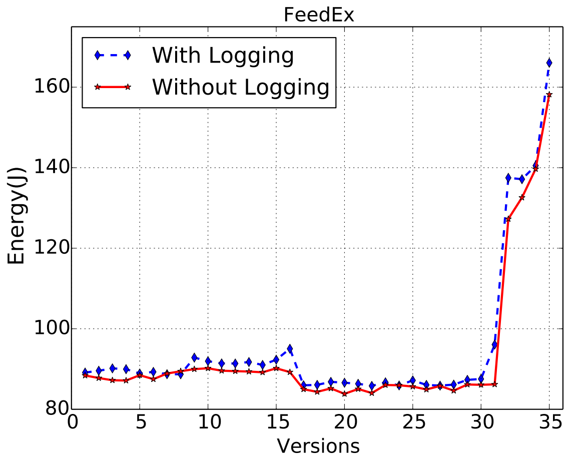


Figure 7.4: FeedEx Energy consumption over time. Versions 32 to 35 exhibit very different energy profiles compared to the previous versions.

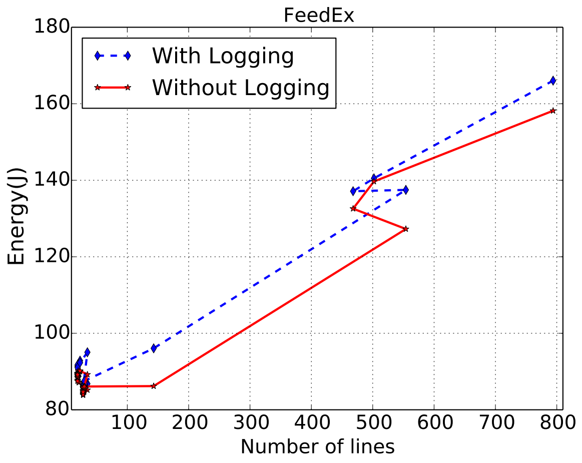


Figure 7.5: Energy consumption against the number of log lines across different FeedEx versions. The graph depicts 2 measurements and the lines connects between adjacent versions. The line depicts how the FeedEx versions move through the space of log length and energy consumption. Essentially consecutive FeedEx versions use more and more energy.

logging. Furthermore, high logging rates (a.k.a., consistent logging rates faster than 20 msg/sec) were not observed from these applications and tests. Hence, in the next RQ (Section 7.4), we will study the factors impacting the energy consumption of logging on Android applications using controlled experiments and with various logging rates.

7.3.4 Summary

Findings: The energy consumption between logging enabled and disabled tests are not statistically significant for most versions of the studied mobile applications. However, approximately 79% of the studied applications have at least one versions with effect sizes larger than or equal to medium. Internal factors such as memory management issues are the causes behind the energy increases correlated with logging.
Implications: Logging usually does not have a noticeable impact on the energy consumption of Android applications, although in some cases it can. Developers should be careful when adding additional instrumentation code, yet still leveraging this valuable debugging tool. Characterizing the best practices on making energy-efficient logging decisions in mobile applications is still an open research problem.

7.4 RQ2: What are the factors impacting the energy consumption of logging on Android applications?

7.4.1 Motivation

Currently, there are few guidelines regarding logging on mobile devices and logging's energy impact for mobile developers to follow. It is not clear to mobile developers how much they can log and how often. This section seeks to provide some insights into this aspect of logging and energy consumption.

7.4.2 Experiments

There are three orthogonal factors that can potentially impact the energy consumption of logging: (1) the rate of logging, (2) the size of log messages, and (3) the number of disk flushes. The rate of logging and the size of log messages can be controlled by the individual applications, but not the number of disk flushes. Depending on the volume of the logs and the buffer size, the number of disk flushes can vary. The volume of the logs (a.k.a., the size of the log file) depends both on the rate of logging and the size of log messages. Bigger log messages and more frequent logging lead to higher volumes of logs. The buffer size varies depending on the mobile phones. Our test-bed uses Android phones with 256 KB circular buffer sizes. The disk flush happens when the buffer gets filled up.

It is not easy to investigate the energy impacting logging factors with real-world applications. First, isolating the pure energy costs for logging is difficult; these applications interact with other components that also consume energy (e.g., radio and screen). Second, unstable logging rates and log size with real-world applications hinder controlled experiments.

Hence, we have built a test Android application to assess the energy-impacting factors for logging. Our test application, which consists of only the `MainActivity` and a JUnit test case, performs only one task: generating logs at different rates and with different message sizes. For each test, the operations are the same: the `MainActivity` is launched. Then the application starts to generate log messages of a specific size at a specific rate for 120 seconds and stops. The duration of 120 seconds is chosen to help stabilize measurements against unexpected CPU frequency switches. By the first 60 seconds of the test, the CPU frequency should be appropriately set for the logging workload. Table 7.6 lists the set of different message rates and message sizes that were run. For each of the specified message rates and sizes, the rationale is also included. For example, app developers might be interested in printing and storing stack traces or packet dumps in a log file. A typical stack trace is around 8,192 bytes (8 KB) and a typical Ethernet packet is 1,536 bytes (1.5 KB). If one logged UI level events, the UI events are usually generated at a rate of 1 to 5 events per second.

Much like RQ1, each experiment was repeated multiple times (40 times)

Table 7.6: Controlled experiments with varying logging rates and message sizes.

Logging Rate		Message Size	
Rate (msg/sec)	Rationale	Size (bytes)	Rationale
0.01	infrequent logging	64	a single line of text
0.10	browsing UI level logging	512	a medium sized line of text
1.00	UI event level logging	1024	a URL sized line of text
10.00	network traffic level logging	1536	maximum ethernet data frame size
100.00	printf debugging logging	2048	a large log message
1000.00	very frequent logging	8192	an exceptionally large log message

Table 7.7: Percentage growth rates of energy consumption (joules) for the log generating tests. All the calculations below used the energy consumption of the idle tests as the baseline.

msg/sec	64 bytes	512 bytes	1 KB	1.5 KB	2 KB	8 KB
0.01	0.20%	0.21%	0.18%	0.51%	0.28%	0.37%
0.10	0.27%	0.31%	0.38%	0.73%	0.30%	0.61%
1.00	0.65%	0.64%	0.70%	0.99%	0.64%	0.90%
10.00	1.38%	1.59%	1.71%	2.16%	1.91%	2.46%
100.00	8.26%	8.48%	9.23%	10.33%	10.55%	14.20%
1000.00	27.88%	30.66%	36.50%	45.14%	48.26%	75.47%

to avoid measurement errors and random noise [75], [129]. It took 70 hours in total to run these tests. The testing results are gathered for further analysis.

7.4.3 Analysis

The average energy consumption of each test is calculated. The data is grouped according to the rate of the logging (a.k.a, msg/sec). The average energy consumed for the idle tests (a.k.a., generating zero msg/sec) is used as the baseline in order to track the percentage increase in terms of energy consumption for the log generating tests. Table 7.7 shows the results. For example, there is a 0.2% energy increase under 0.01 msg/sec with 64 bytes as the message size.

In fact, for 10 msg/sec there is a very small energy increase ($\leq$ 2.46%), with the largest message size. However, the impact is significant with larger message rates. As the logging rate increases to 100 msg/sec, the increase in the energy consumption ranges from 8.26% to 14.20% for different message sizes. For 1000 msg/sec, the increase of the energy consumption can go up to 75.47%. Another interesting observation is that the energy increase for 10 msg/sec and 8 KB message size is much smaller than 100 msg/sec and 64 byte message size (2.46% vs. 8.26%), even though the unit volume of the generated logs are much higher (80 KB/sec vs. 6.25 KB/sec). *Evidently, logging rate is a more dominant factor than message size for energy consumption*

For further verification, we apply factor analysis to verify the importance of message size and log rate. Kruskal-Wallis (Kruskal-Wallis X^2) tests are performed to check whether the factors of logging rates and the message sizes have statistically significant impacts on the energy consumption. Kruskal-Wallis test is a non-parametric statistical test for checking whether the measurements of 3 or more groups, under different kinds of treatments, come from the same distribution. We test 2 factors independently: logging rates and logging message sizes. We correct for multiple/hypotheses with the Benjamini and Hochberg method [35]. Logging rate was a significant factor ($p < 2.2e - 16$) for energy consumption. Although the message size is also a statistically significant factor, the p-value ($p < 0.0471$) is very close to our α ($\alpha = 0.05$). In fact, when we test with the pairwise Wilcoxon rank sum test between the *message sizes*, corrected using Benjamini and Hochberg, we find no statistically significant differences in energy consumption between distributions of different message sizes ($p > \alpha$). Yet a pairwise Wilcoxon rank rum test shows there are statistically significant differences ($p \leq \alpha$) for all log rate comparisons except for 2 comparisons of log rates of 0.01 to 0.1 and log rates of 0.1 to 1.0. The similar distributions of energy consumption at low frequency log rates also helps to explain the mild inconsistency in trend observed in energy consumption with the message sizes.

In order to better understand the relationship between the message size and the energy consumption, we calculate the Pearson correlation coefficient between them—an indicator of a linear relationship between two random variables. The correlation coefficient is low (only 0.17 with $p \approx 0$) when the message rate is not fixed. However, with fixed and high message rate (e.g., 100 msg/sec), the correlation is high (0.72 with $p \approx 0$). This is also consistent when the message rate is fixed at 1000 msg/sec. These observations corroborate the results in RQ1 whereby most differences were not statistically significant as the message sizes and logging rates in RQ1 were often low.

We observe that with the increase in logging rate energy consumption also increases; but the same does not apply for message size. However, in case of heavy logging both the rate and the size become significant factors towards energy consumption. This also explains the observed inconsistencies in Table 7.7. One would expect that with the increase in message size, the energy consumption would also increase. However, no such trend is observed from Table 7.7 when the message rate is low. For instance, the energy increase in joules for 10 msg/sec with 1.5 KB message size is 2.16%, which is higher than 2 KB message size with the same rate (1.91% increase). This led us to evaluate if the differences in energy consumption with this two settings are really different, because Table 7.7 only shows the increase in percentage considering the average of the 40 measurements for each scenario.

To investigate the difference across both factors at once, we run a handful of tests to investigate trends depicted in Table 7.7. We apply the Wilcoxon rank sum test (a non-parametric test), and found that the energy consumption between the above mentioned two settings (10 msg/sec with 1.5 KB log message size versus 10 msg/sec with 2 KB log message size) are not statistically different (p-value > 0.05). The difference is not statistically significant either (p-value > 0.05) for 0.01 msg/sec with 1.5 KB log message size versus 0.01 msg/sec with 2 KB log message size. However, when the message rate is high, the difference in energy consumption with different message sizes are significant. For example, the energy consumption differences are statistically significant (p-value < 0.05) for 1000 msg/sec with 1.5 KB log message size vs. 1000 msg/sec with 2 KB log message size. This is another confirmation that message size only has a noticeable effect with high message rates.

We also build a linear regression model to estimate the energy consumption using the message sizes, the logging rates, and the number of disk flushes. This further clarifies the impact of these factors on energy consumption, as we executed the same application that does nothing than writing log messages. The logging rates and the message sizes are obtained from each test configuration (Table 7.7). The number of disk flushes can be calculated by dividing the estimated file size with the buffer size. For example, after 120 seconds of testing, the size of the log file from the 1000 msg/sec and 1536 bytes test would be 180,000 KB. Hence, with 256 KB buffer size, the estimated number of disk flushes would be 703. The resulting regression model is shown below as Equation 7.1 and has an adjusted R-squared value of 0.87.

$$\begin{aligned} \text{joules} =\,& 0.03370 \times \text{message rate} + \\ & 0.00006 \times \text{message size} + \\ & 0.01328 \times \text{number of disk flushes} + \\ & 112.10958 \end{aligned} \tag{7.1}$$

This model also confirms that message rate, and subsequently the number of disk flushes are more significant factors for energy consumption than message size. The rationale is that, as we have already shown, with very low message rate message size does not impact the number of disk flushes significantly.

In summary, our results suggest that mobile application developers do not need to prematurely optimize and trade-off energy consumption for logging. Infrequent logging has limited impact on the overall energy consumption. However, if there is a need to generate large amount of logging content, to conserve energy, it is preferable to log infrequently with larger message sizes rather than logging frequently with smaller message sizes. This is similar to the earlier findings [117], [141], [194] that bundling smaller packets together reduces significant energy consumption in data communication.

7.4.4 Summary

Findings: Small amounts of logging ($\leq$ 10 log messages per second) have little or no energy impact on the mobile applications. In fact, message size does not have any significant impact on energy when the logging rate is very low. On the other hand, both the message rate and size are significant factors towards draining energy under heavy logging. Under heavy logging, logging large amounts of data infrequently consumes much less energy than frequently logging smaller amounts of data.
Implications: To conserve energy, developers should strategically instrument their code. The preferred logging points can contain more contextual information but are less frequently executed (e.g., avoid logging within loops or commonly called library functions). When heavy logging is needed, developers should group small log messages and write them together to conserve energy.

7.5 RQ3: Is there any relationship between the logging events and the energy consumption of mobile applications?

7.5.1 Motivation

Are the causes of energy consumption, events correlated with energy consumption, apparent in the log? Measuring energy consumption directly often requires both hardware instrumentation and software instrumentation. It is a time-consuming process as hardware test-beds instrumented with a power monitor must run tests multiple times to get a statistically reliable estimate of power use [100], [208]. Moreover, such a test-bed might be expensive for many app developers.

The execution logs are debug statements that developers inserted into their code. These instrumentation locations are strategically selected to debug and monitor the functionalities of the applications. The key steps (e.g., displaying the hand-drawn objects or performing email reconciliations) during the executions are often logged and can provide us hints on the energy consumption patterns of the applications. It would be cheaper and faster for developers to diagnose their mobile application energy regression problems by analyzing their log files.

This RQ investigates the feasibility in terms of using the readily available execution logs to understand the energy consumption of mobile applications. In particular, we want to check if there is any relationship between the logging events and the energy consumption of the mobile applications. The question is what events, that get recorded in the log, induce energy consumption. Thus, we are not seeking to truly estimate energy but we seek to investigate the relationship between common events that get recorded in logs, and energy consumption.

7.5.2 Experiments

We do not perform additional performance testing in this RQ. Rather, we reuse the log files and the energy measurements from RQ1. In particular, we reuse the measurements of the log-enabled tests of the 24 Android applications, including all of the versions used.

7.5.3 Analysis

There are three steps involved in this analysis. First, the free-form log messages are abstracted into log events. Second, correlations are calculated between individual log events and energy consumption. Third, we look into the combination of variables using multiple regression—by exhaustive model building we hope to better understand what log events work together to consume energy.

Step 1 - log abstraction

Execution logs typically do not follow a strict format. Each log line contains a mixture of static and dynamic information. The static information is the descriptions of the execution events, whereas the dynamic values indicate the corresponding context of these events. For example, the last log line in Table 7.1 contains static information like "I/ActivityManager", "Force stopping package com.android2.calculator3", "appid" and "user". The numbers "387", "10062" and "0" are likely generated during run-time.

Such free-formed log messages need to be abstracted into events so that they can be used in automated statistical or data mining analysis. We apply the log abstraction technique proposed by Jiang *et al.* [122] to automatically map log messages to log events.

Since the same test scenario was executed for the same version of an application, the generated log events should be similar or even identical. Hence, test runs on the same version are combined into a single log file, by averaging the count of each log event obtained during the repeated test.

Table 7.8 summarizes the number of unique log events and log length per application across all of the different test runs of their multiple versions.

Table 7.8: Summary of unique log events per application across all the versions.

App Names	Total Versions	# of Unique Events total	minimum	median	average	maximum	Standard Deviations
2048	44	15	12	13.000	13.091	14	0.603
24game	1	53	53	53.000	53.000	53	
Acrylic Paint	40	19	11	14.000	13.800	17	1.324
Agram	3	15	15	15.000	15.000	15	0.000
AndQuote	21	11	11	11.000	11.000	11	0.000
Blockinger	74	168	23	27.000	27.392	35	1.560
Bomber	79	24	17	20.000	20.152	22	1.292
Budget	59	74	13	37.000	31.525	41	9.482
Calculator	97	53	9	11.000	14.526	23	5.354
ChromeShell	50	105	92	101.000	100.680	104	2.860
DalvikExplorer	13	17	16	16.000	16.308	17	0.480
Exodus	3	88	52	54.000	54.000	56	2.000
Eye in the Sky	1	58	58	58.000	58.000	58	
Face Slim	1	18	18	18.000	18.000	18	
FeedEx	35	39	14	17.000	17.486	28	4.231
Firefox	156	138	29	44.000	52.340	80	22.350
GnuCash	16	399	39	54.500	54.688	62	5.449
Memopad	52	13	12	13.000	12.712	13	0.457
Paint Electric Sheep	1	21	21	21.000	21.000	21	
Sensor Readout	37	12	11	11.000	11.189	12	0.397
Temaki	66	15	8	9.000	9.485	12	1.231
Vector Pinball	54	56	14	16.000	16.463	21	1.342
VLC	46	492	385	390.000	390.022	396	2.679
Wikimedia	58	214	71	96.000	94.000	116	8.840
Average	42	88	42	46.646	46.911	52	3.597

The number of unique log events for each application ranged from 11 unique log events for AndQuote to 492 unique log events for VLC. The mean number of unique log events per application is 88, while the median was 46.646.

Now we look into how the prevalence of unique log events varies within log files from different runs and versions. The total number of unique log events can change across different versions of an application. The average standard deviation of total unique log events per application across versions was 3.597 events with a minimum standard deviation of total log events was 0.000 unique log events (no change) for AndQuote and Agram and a maximum standard deviation of total unique log events 22.350 for Firefox. The statistics about each application are described in Table 7.8.

Step 2 - correlation between log events and energy consumption

Table 7.9 depicts the Spearman correlation coefficients between each log event and the energy measurements for all the 24 applications.

Table 7.9: Spearman's ρ correlation coefficient distribution between log event types and joules per application. Each column shows how many log events correlated with the correlation scale proposed by Hopkins *et al.* [105]

App Names	# of Unique Log Events	% Trivial $[0.0, 0.1)$	% Small $[0.1, 0.3)$	% Moderate $[0.3, 0.5)$	% Large $[0.5, 0.7)$	% Very Large $[0.7, 0.9)$	% Near Perfect $[0.9, 1.0]$
2048	15	80.000	13.333	6.667	0.000	0.000	0.000
24game	53	100.000	0.000	0.000	0.000	0.000	0.000
Acrylic Paint	19	36.842	15.789	26.316	21.053	0.000	0.000
Agram	15	80.000	0.000	0.000	6.667	0.000	13.333
AndQuote	11	72.727	27.273	0.000	0.000	0.000	0.000
Blockinger	168	62.500	36.905	0.595	0.000	0.000	0.000
Bomber	24	33.333	25.000	41.667	0.000	0.000	0.000
Budget	74	39.189	22.973	13.514	5.405	18.919	0.000
Calculator	53	18.868	47.170	24.528	9.434	0.000	0.000
ChromeShell	105	22.857	70.476	6.667	0.000	0.000	0.000
DalvikExplorer	17	94.118	0.000	0.000	5.882	0.000	0.000
Exodus	88	21.591	0.000	0.000	21.591	39.773	17.045
Eye in the Sky	58	100.000	0.000	0.000	0.000	0.000	0.000
Face Slim	18	100.000	0.000	0.000	0.000	0.000	0.000
FeedEx	39	38.462	17.949	0.000	35.897	7.692	0.000
Firefox	138	20.290	78.261	1.449	0.000	0.000	0.000
GnuCash	399	21.554	30.576	39.850	7.769	0.251	0.000
Memopad	13	69.231	0.000	0.000	7.692	23.077	0.000
Paint Electric Sheep	21	100.000	0.000	0.000	0.000	0.000	0.000
Sensor Readout	12	91.667	0.000	8.333	0.000	0.000	0.000
Temaki	15	46.667	26.667	26.667	0.000	0.000	0.000
Vector Pinball	56	16.071	23.214	26.786	3.571	30.357	0.000
VLC	492	41.260	13.415	2.236	1.423	41.667	0.000
Wikimedia	214	22.897	1.402	53.738	21.963	0.000	0.000

Spearman's ρ correlation is a non-parametric test that assesses the relationship between two variables. The characterization of the strength of the correlation (trivial, small, medium, large, very large and near perfect) is proposed by Hopkins *et al.* [105]. For example, 205 out of the 492 unique events in VLC exhibit very large correlations with the energy consumption; whereas all the log events in Face Slim have little or no correlation. The correlation values in the table show that 79% of the studied applications have at least one log events which exhibit medium to near perfect correlation values with the energy consumption of the mobile applications. If only large to near perfect correlations are considered, there are still 50% of the studied applications that have some log events strongly correlated with energy consumption.

Across most applications (e.g., Firefox, Agram, Acrylic Paint, and Memopad), the log events with the highest correlation with energy consumption are often related to the Dalvik Virtual Machine, DalvikVM. The DalvikVM is the Java Virtual Machine used by the Android operating system to run mobile applications. The lists of events that are highly correlated with energy consumption are shown below. Some of them are related to identified "energy greedy APIs" [144].

- `Dalvikvm: GC CONCURRENT (X1)`: this event is triggered when the heap starts to fill up;
- `Dalvikvm: GC FOR ALLOC (X5)`: this event occurs when there is not enough memory left on the heap to perform an allocation;
- `Dalvikvm: GROW HEAP (X6)`: in order to save memory, Android does not allocate maximum amount of requested memory to every application automatically. Instead, the OS waits until the application requests more memory. Then this event is triggered to give more heap space until the maximum amount of memory is reached.

For these 3 events, the median magnitude of Spearman's ρ correlation (absolute value) over all applications with more than 1 version is 0.333 for

`Dalvikvm: GC CONCURRENT`, 0.308 for `Dalvikvm: GC FOR ALLOC`, and 0.219 for `Dalvikvm: GROW HEAP`. This shows a small to medium relationship between Dalvikvm memory management and energy consumption.

Not all log events that are correlated with energy consumption are common across applications. Some of the highly correlated log events are workload specific for a particular application. For example, in GnuCash, workload-specific log events regarding the *onCreateView* method for the *DatePickerDialog* class, and a log event about replacing account entries in the database exhibited large positive ($\rho = 0.6873$) and very large negative correlations with energy consumption ($\rho = -0.7515$), respectively. For Vector Pinball, trying to load the JNI library for Box2D, a 2D physics library, (`DalvikVM trying to load lib.data.app.lib.com.dozingcatsoftware.bouncy libgdx.box2d.so`) has a very large negative correlation ($\rho = -0.7923$) with joules. VLC has very large positive correlation ($\rho = 0.8377$) with input controls (`VLC core input control stopping input`).

Step 3 - building energy consumption models using logs

In this step, the relationship between these log events and energy consumption is further studied through multiple regression analysis. If an independent variable (i.e., a log event) reoccurs in numerous models, then we argue that that variable demonstrates a strong relationship with software energy consumption for different mobile applications. The intent of this section is not necessarily to build reusable predictors, but to further study the relationship between energy consumption and common log events. We use multiple linear regression to study the relationship between different log events (as independent variables) and the energy consumption (as the dependent variable).

There were 122 log events that commonly occurred across 3 or more applications. When considering common log events shared by four or more applications, there are only a total of 17 log events. There are 10 log events commonly shared by 6 or more applications. Hence, in this step, we pick the common log events that are shared by at least four applications, as we want to derive more general prediction models in order to study the effect of common log events on software energy consumption. Table 7.10 shows the 17 selected log events. For the sake of brevity, a short log event name is shown instead of the fully abstracted log events.

Table 7.10: OS Level Log events shared by all the applications

#	# applications	Event Name
X1	25	dalvikvm GC_CONCURRENT
X2	24	dalvikvm WAIT_FOR_CONCURRENT_GC
X3	21	libEGL loaded vendor lib egl libGLESv2_POWERVR_SGX540_120
X4	20	OpenGLRenderer Enabling debug mode
X5	19	dalvikvm GC_FOR_ALLOC
X6	17	dalvikvm heap Grow
X7	14	dalvikvm Late enabling CheckJNI
X8	8	dalvikvm Turning on JNI app bug workarounds for target SDK version
X9	6	TilesManager Starting TG
X10	6	Choreographer Skipped frames The application may be doing too much work on its main thread
X11	4	dalvikvm Jit resizing JitTable
X12	4	webviewglue nativeDestroy view
X13	4	GLWebViewState Reinit transferQueue
X14	4	dalvikvm null clazz in OP_INSTANCE_OF single stepping
X15	4	InputMethodManagerService Focus gain on non focused
X16	4	dalvikvm VFY replacing opcode
X17	4	GLWebViewState Reinit shader

Many log event counts are highly correlated with other event counts. We have exhaustively tried all subsets of variables in the model and ignored models that included co-linearity whereby any two independent variables had a Pearson correlation greater than 0.3 or less than -0.3. The models are kept if all the independent variables are reported as statistically significant to the model ($p \leq 0.05$). Models that do not produce a significant F-statistic ($p \leq 0.05$) are not kept.

The final models are the ones with the largest number of significant events. Several regression models with more than two log events are found. These models are for both the individual applications and all the applications at once. Table 7.11 shows the models extracted and their adjusted R-squared values.

Some applications do not have enough versions, or their common log events do not correlate well enough to produce a significant linear model (e.g., 24Game, 2048, and Blockinger). Each model has the following form:

$$\text{joules} \backsim c_0 + c_1 \times \text{event}_1 + c_2 \times \text{event}_2 + ... + c_n \times \text{event}_n \qquad (7.2)$$

In general, different models from the same application share similar prediction performance. The three models from FeedEx show that the top common log events can predict the energy consumption of FeedEx very well (with Adj-$R^2 \geq 0.92$) . However, the models from the Calculator and the Firefox applications show moderate prediction performance with Adj-$R^2 \geq 0.39$ and Adj-$R^2 \geq 0.22$, respectively. The ChromeShell app is not modeled well by the common log events with an Adj-$R^2 \geq 0.11$, while the Vector Pinball app performs the best for predictability with models with an Adj-$R^2 \geq 0.94$.

Across all the studied version of Android applications, typically events X1, X5, and X6 are part of the successful models. These events are related to the Dalvik Virtual Machine. The listed events are related to memory management operations such as garbage collection and memory allocations. In the models utilizing all applications and versions, X17 and X4 were also quite significant, as well as X1 and X6. X4 and X17 are OpenGL and graphics relevant.

Table 7.11: Linear models of energy consumption based on log events across numerous Android applications. Top three models are shown only if they are significant ($p \leq 0.05$).

	Events#	Adj-R^2	p-value
All applications/Versions	X4 + X6 + X17	0.4755	1.1222e-140
	X1 + X4 + X6	0.4965	1.4197e-149
	X1 + X4 + X6 + X9	0.5233	2.3270e-160
	X1 + X4 + X6 + X13	0.5328	9.2390e-165
	X1 + X4 + X6 + X17	0.5328	9.2390e-165
Acrylic Paint	X3 + X5	0.4870	1.6392e-06
	X4 + X5	0.4870	1.6392e-06
	X2 + X5	0.5306	3.1700e-07
Bomber	X6 + X14	0.2375	3.1351e-06
	X4 + X6	0.2375	3.1351e-06
	X6 + X13	0.2375	3.1351e-06
Calculator	X3 + X5	0.3991	2.3918e-12
	X5 + X16	0.3991	2.3918e-12
	X5 + X17	0.3991	2.3918e-12
ChromeShell	X1 + X15 + X16	0.1076	2.5888e-02
	X1 + X13 + X16	0.1076	2.5888e-02
	X1 + X2 + X16	0.1112	3.8161e-02
Firefox	X2 + X3 + X11	0.2242	1.3689e-09
	X2 + X3 + X13	0.2242	1.3689e-09
	X2 + X3 + X10	0.2242	1.3689e-09
GnuCash	X1 + X5 + X6	0.5914	3.0130e-04
	X1 + X4 + X5	0.5914	3.0130e-04
	X1 + X3 + X4	0.5914	3.0130e-04
Memopad	X1 + X8 + X12	0.6377	5.9170e-12
	X1 + X8 + X13	0.6377	5.9170e-12
	X1 + X8 + X14	0.6377	5.9170e-12
Sensor Readout	X1 + X2	0.4276	2.8786e-05
	X1	0.4427	4.2210e-06
Budget	X2 + X15	0.8176	7.6227e-22
	X1 + X5	0.8262	1.9788e-22
	X1 + X6	0.8330	6.4611e-23
Vector Pinball	X1 + X5 + X7 + X8	0.9456	3.1336e-32
	X1 + X5 + X8 + X16	0.9456	3.1336e-32
	X1 + X8	0.9465	1.4174e-33
Temaki	X1 + X7 + X15	0.4848	1.2541e-09
	X1 + X3 + X7 + X15	0.4848	1.2541e-09
	X1 + X7 + X15 + X17	0.4848	1.2541e-09
VLC	X1 + X5 + X16	0.6134	5.0341e-10
	X1 + X3 + X5	0.6134	5.0341e-10
	X1 + X5 + X17	0.6134	5.0341e-10
Wikimedia	X3 + X5	0.6396	3.1125e-14
	X1 + X5	0.6666	2.8477e-14
	X2 + X5	0.7289	9.6649e-17
FeedEx	X17	0.9297	8.2694e-21
	X16	0.9297	8.2694e-21
	X13	0.9297	8.2694e-21

7.5.4 Summary

Findings: Around 80% of the applications have at least one log event whose correlation with the energy consumption are medium or stronger. Memory management and graphics-related (OpenGL) log events are the most correlated log events related to mobile software energy consumption. For some applications, there are also some workload-specific log events which exhibit high correlation with the energy consumption. Models trained on top common log events demonstrate a clear relationship between those log events and the energy consumption for some but not all mobile applications.
Implications: App developers should watch out for log events related to garbage collection and graphics if they are concerned with the energy consumption of their applications — especially changes in the number of these log events. Furthermore, although logs have been used effectively to debug and troubleshoot functional problems, there is still no clear relation between the logging contents and the energy consumption for some applications. Researchers should investigate into innovative logging approaches which can help debug both energy and other performance-related problems.

7.6 Threats to validity

In this section, we discuss the threats to validity.

7.6.1 Construct validity

Reliability of the energy measurement

It is important to ensure reliable performance measurement, as performance measurement is subject to measurement error and random noise [75], [129], [179]. In this paper, we have used two strategies to mitigate this threat: (1) around 1,000 versions from 24 Android applications were studied with both logging enabled and disabled; and (2) each version of the same application was repeatedly tested and measured to ensure measurement accuracy. We did not have clear control over laboratory temperature, but according to the INA219's specification [113], measurements do not deviate much over the range of temperatures expected while running the tests. Energy measures can suffer from sampling, but the INA219 does sample at a high rate and output aggregated measurements at a lower rate. This aggregation can induce error but given the high rate of sampling by the INA219 it is unlikely to have meaningful effect on test runs. Most importantly energy consumption is a physical process thus one must measure multiple times as we do in this paper.

7.6.2 Internal validity

Controlling confounding factors while assessing the energy impact of logging on real-world android applications

There are many Android applications that try to log data in a real-world setting. Hence, while executing our performance tests on real-world Android applications, we make sure all the applications under test are running in the same Android running environment. In addition, we also make sure the application under test is the only running user application during the tests. For each application under test, we have executed two types of performance tests: logging enabled and logging disabled. All the test configurations are the same for these two types of tests, except for enabling and disabling logs. We ran the experiments one after the other. Each experiment takes less than five minutes. Hence, for some applications that access outside resources (e.g., FIREFOX requesting data from Wikipedia), the chances of a resource changing during a test (e.g., content updates in the Wikipedia webpage) exists but would be very low.

Controlling various logging factors while investigating the energy consumption of logging on Android applications

There are many factors impacting the energy consumption of logging on Android applications. Factors such as the logging rate and the log message sizes cannot be easily controlled on real-world use cases and applications. In addition, real-world applications also perform other tasks (e.g., networking and video playing), which makes it difficult to isolate the energy impact of logging. Hence, to control the various confounding factors, we have developed a testing

Android application which is dedicated only to log messages at different rate and size. The values of the logging rates and message sizes were derived based on actual scenarios in practice (e.g., the size of a network packet and the size of a typical stack trace). Since the logging rates and the message sizes combined could have an impact on the size of the log files, an additional factor, the number of disk flushes, is introduced to assess the combined impact of logging rate and log message size. Yet the Android operating system is a complex piece of software, thus state will slowly change while the operating system running, for instance the file system state will change between runs. Future tests could replace the file system each and every time in order to control the non-determinism in the file system.

7.6.3 External validity

Generalizing the energy impact of logging on real-world Android applications

To ensure our findings on the energy impact of logging on real-world Android applications are generalizable, we have selected 24 Android applications from different application domains. In addition, many of the applications in our dataset have many versions. These versions correspond to a range of different software development activities (e.g., new features and bug fixes). Increasing both the number of applications and versions covered would provide better generality. However, our findings might not be able to generalize to other mobile application platforms (e.g., BlackBerry, iOS or Windows phones) and other Android phones.

In addition, although we have designed our test cases to closely mimic the realistic user usage of mobile applications, the resulting test cases may not cover all the possible uses for the studied applications.

7.7 Related work

In this section, we will discuss three areas of prior research that are related to this paper: (1) energy testing and modeling for mobile applications, (2) empirical studies on energy-efficient mobile development, and (3) execution logs.

7.7.1 Energy testing and modeling for mobile applications

Hindle *et al.* developed the *GreenMiner*, an automated test-bed to assess the energy consumption for each revision of a given mobile application [100], [102]. Since running performance tests on each revision is time consuming, Romansky *et al.* [208] proposed a search-based test approximation technique to reduce the testing efforts in *GreenMiner*. Li *et al.* [140] proposed a test minimization technique that prioritizes the test suites with higher energy consumption. This paper leverages the *GreenMiner* [100], [102] to perform energy testing on different versions of the mobile applications with and without logging enabled.

There have been many studies dedicated to modeling energy consumption for mobile applications. In general, there are three approaches, which use three different datasets, collected by different monitoring and profiling tools, to model mobile energy consumption: (1) hardware-based counters [40], [62], [71], [90], [227], [269]; (2) program instructions from the applications [92], [139], [219]; and (3) system calls [5], [48], [194]. Different mobile monitoring and profiling tools can bring different insights into the mobile applications' dynamic behavior. However, they all have some runtime overhead. Different from the above three approaches, this paper builds the energy consumption models to explore factors of energy consumption prevalent in execution logs.

7.7.2 Empirical studies on energy-efficient mobile development

We further divide the empirical studies on energy-efficient mobile development into the following three sub-areas:

- **App Developers:** Pinto *et al.* [198] investigated questions on StackOverflow that programmers had about energy. They found that programmers lacked the resources to answer questions about software energy consumption. Similar findings were confirmed by other studies that surveyed programmers about their understanding on software energy consumptions [159], [191]. Chowdhury *et al.* [44] compared energy-aware software projects with projects that did not consider energy-efficiency as one of the non-functional requirements. They found that energy-aware software projects are more popular in terms of number of forks, and contributors.

- **Code Obfuscation:** Sahin *et al.* studied the impact of code obfuscation [213] and refactoring [212] on energy consumption of several Android applications. They found that code obfuscation does impact energy consumption but the differences could be too small for users to notice, whereas the impact of code refactoring could be mixed (a.k.a., either increases or decreases in energy consumption).

- **Energy Greedy APIs, Frameworks, and Platforms:** Li *et al.* [138] leveraged their technique of estimating energy consumption for source lines in [139] and studied the API level energy consumption patterns of different mobile applications. They found that the networking component consumes the most energy and more than half of the energy consumption is spent on idle state. This observation indicates that reducing the number of idle states can optimize energy consumption for mobile applications. Linares *et al.* [144] identified energy greedy Android APIs that can be helpful for the developers to write energy efficient code.

Chowdhury *et al.* [50] found that employing HTTP/2 server can save energy for the mobile clients. Pathak *et al.* suggested that around 70% of mobile software energy bugs are the direct result of problems linked to wake locks [193]. Hence, many studies have focused on understanding and optimizing wake lock in mobile applications [10], [28], [149], [192], [195], [249]. Tail energy leaks, the energy cost of powering up and eventually powering down peripherals, have been studied as a source of energy consumption in mobile applications [50], [141], [194]. Tail energy leaks can be optimized by bundling I/O operations together [50], [141]. Hasan *et al.* [95] studied the energy profiles of frequently used Java collection classes and suggested that using the most energy efficient collection classes can save up to 300% software energy.

Logs are widely used in software development for various purposes like debugging, monitoring and user behavior tracking. However, there are no prior studies focused on the energy impact of logging for mobile applications. Hence, in this paper, we performed an empirical study on another aspect of energy-efficient mobile development: the energy consumption of software logging.

7.7.3 Execution logs

We will discuss two areas of related research on execution logs:

- **Empirical Studies on Execution Logs:** There have been a few empirical studies conducted to investigate the logging activities in practice. Shang *et al.* [222] analyzed how log events evolve over time by executing the same scenarios across different versions of the same applications. They found that logs related to domain level events (e.g., workload) are less likely to change compared to logs related to feature implementations (e.g., opening a database connection). Yuan *et al.* [262] analyzed the source code revision history for 4 C-based open source software systems. They found that log events are often added as "after-thoughts" (a.k.a., after failure happens). They also developed a verbosity-level checker to automatically detect anomalous log levels (e.g., DEBUG vs. FATAL) using clone analysis. Fu *et al.* [73] performed a similar log characteristic study but on the source code of two large industry systems at Microsoft. Shang *et al.* [224] studied the release history of two open source applications (Hadoop and JBoss) and found that files with many logging statements have higher post-release defect densities than those without. Unfortunately, all of the prior empirical studies on execution logs focused on desktop and server-based applications. This paper is the first research work focused on studying the execution logs on mobile applications.

- **Analyzing Execution Logs** Execution logs have been used extensively by developers, testers and system operators to monitor and diagnose problems for large-scale software systems [186], [261]. Execution logs have a loosely-defined structure and a large non-standardized vocabulary. Due to its sheer volume of size (hundred megabytes or even terabytes of data), it is usually not feasible to analyze the logs manually. Techniques have been proposed to automatically abstract the loosely structured execution log events into regularized log events [122], [257]. Then automated statistical and AI techniques can be applied on these regularized log events to analyze the results of load tests [123], and to monitor, detect and diagnose problems in big data applications [223], [256], [257]. In addition to leveraging the existing logs, Yuan *et al.* proposed a technique to automatically suggest logging points to aid the debugging activities using program analysis [263]. Finally, Ding *et al.* [60] proposed a cost-aware logging mechanism so that informative logs can be generated while still ensuring the performance overhead is within the specified budget. Their performance overhead is defined in terms of resource usage (e.g., CPU, memory and disk I/O) and their target applications are large-scale server applications. In this paper, we used the log abstraction technique proposed by Jiang *et al.* [122] to build our energy consumption model.

7.8 Conclusion

Software developers use execution logs to debug and monitor the health of mobile applications. This paper investigates the energy impact of execution logs on Android applications. Around 1,000 versions of 24 Android applications were tested and measured under logging enabled and disabled. In addition, a controlled experiment with varying rates of logging and sizes of the log messages was carried out.

Our experiments show that limited logging (e.g. ≤ 1 msg/sec) has little to no impact on the energy consumption of mobile applications. Although there is little to no impact on the energy consumption of logging for most of the versions, there are still many versions with medium to large effect sizes when comparing the energy consumption between when logging is enabled and logging is disabled. The rate of logging, the size of log messages, and the number of disk flushes are three statistically significant factors that impact the energy consumption of logging. Log events can be used in energy consumption debugging as some events common across applications, that are logged as log events, are highly correlated with energy consumption—especially those regarding garbage collection or graphics. Depending on the application, some workload-specific log messages are also correlated with energy consumption. However, building energy consumption models with log events yield mixed performance. It would be an interesting future work to leverage event logs, as a proxy to predict the energy consumption of applications.

In conclusion we have presented evidence that logging under relatively liberal conditions of less than 1 log message per second does not have a significant effect on energy performance. Furthermore we have shown with numerous existing Android applications that logging typically has a negligible effect on energy consumption. Although there are some log events recorded in logs which are highly correlated to the energy consumption of the mobile applications, it is still an open research question on how one can leverage software logging to debug energy problems.

Replication package

To aide replicability, we freely disclose and share our dataset and source code for our analysis in our replication package [205]. The GreenOracle tests that were run on the *GreenMiner* are located at https://github.com/shaifulcse/-GreenOracle-Data/tree/master/Tests.

Chapter 8

GreenBundle: Addressing energy efficiency from design time

This chapter presents the final contribution related to the second objective of this thesis (explained in 1.3.2)—providing guidelines for achieving energy efficiency. In particular, it shows how developers can address energy efficiency from the design time, and the trade-offs that energy-aware developers need to consider.

This chapter was published as:

- Shaiful Chowdhury, Abram Hindle, Rick Kazman, Takumi Shuto, Ken Matsui, and Yasutaka Kamei, "GreenBundle: An Empirical Study on the Energy Impact of Bundled Processing", In 41^{st} ACM/IEEE International Conference on Software Engineering (ICSE 2019, technical track), pages 1107-1118. May 25-31, 2019. Montreal, Canada [84].

Research has been done for providing energy efficient guidelines. For example, developers can select the most energy efficient Java collections based on the recommendations by Hasan *et al.* [95]. Developers can follow energy efficient database commit techniques as suggested by Lyu *et al.* [154]. However, these energy efficient techniques (i.e., applying little tweaking in source code) often do not have significant impact on real-world apps [211]. This implies that we require higher level changes for efficiently accessing energy hungry hardware components. Also, it is important that developers address energy efficiency from the design time, before coding starts [159]. Developers, however, need to consider different trade-offs before making a design decision. Would an energy efficient design decision lead to worse user experience and higher maintenance cost?

In this chapter, an energy efficient design pattern—a bundled Model-View-Presenter—is proposed that developers can follow from the design time. With this pattern, we show how we can make view updates significantly more energy efficient by delaying the updates (adding latency in view updates). In contrast to the traditional Model-View-Presenter (MVP) pattern, a bundled MVP does not update a view immediately. Rather, the presenter stores all the incoming view update requests (i.e., events) for a certain period of time (here comes the latency and the trade-offs with user experience), and then processes them all in a batch. This chapter also shows the impact on the software maintenance cost for applying the bundled MVP pattern.

The takeaways from this chapter include:

- For Android systems, we show an energy efficient design pattern (bundled MVP) that can reduce the energy consumption of view updates even in real-world apps.
- Even with imperceptible latency in view updates (0.1 second delay), we can save significant energy consumption with the proposed pattern.
- We show that the bundled MVP has insignificant impact on the app maintenance cost.

My role in the GreenBundle study:

I, with the guidance from my supervisor Abram Hindle and my collaborator professor Rick Kazman, made plans for the methodologies, data collection, experimentation, and evaluations. Professor Yasutaka Kamei and two of his students Takumi Shuto and Ken Matsui helped us collecting energy measurements from a different phone. In addition, professor Yasutaka Kamei helped me to improve some of the sections of the published paper.

Impact: This chapter was published just before writing this paragraph, but because of its potential, it was accepted at the most prestigious software engineering conference (ICSE 2019). We have already received emails from the software energy research community mentioning the impact this paper would have.

Abstract

Energy consumption is a concern in the data-center and at the edge, on mobile devices such as smartphones. Software that consumes too much energy threatens the utility of the end-user's mobile device. Energy consumption is fundamentally a systemic kind of performance and hence it should be addressed at design time via a software architecture that supports it, rather than after release, via some form of refactoring. Unfortunately developers often lack knowledge of what kinds of designs and architectures can help address software energy consumption. In this paper we show that some simple design choices can have significant effects on energy consumption. In particular we examine the Model-View-Controller architectural pattern and demonstrate how converting to Model-View-Presenter with bundling can improve the energy performance of both benchmark systems and real world applications. We show the relationship between energy consumption and bundled and delayed view updates: bundling events in the presenter can often reduce energy consumption by 30%.

8.1 Introduction

Energy consumption is an important quality requirement for mobile devices [253] and for mobile software such as apps [131], [251] that affects availability, battery life, and sales. Unfortunately, and often, app developers are addressing energy consumption when it becomes a problem [159], [251], rather than at design time before coding starts. There is evidence that developers simply are not trained enough in the topic of energy consumption at the application level to be able to address energy consumption effectively [159], [191]. Also, available optimization tips do not impact energy consumption in real-world apps, demanding higher-level changes for efficient accesses of energy hungry components [211]. Unfortunately, developers have little idea about what design choices are even available that will affect energy consumption, as well as the consequence and tradeoffs of those design choices. Yet interest exists as Manotoas et al. [159] show: "experienced practitioners are often willing to sacrifice other requirements for reduced energy usage". This paper discusses the kinds of design choices and tradeoffs that architects face, and seeks to illustrate how we can improve energy consumption of mobile applications (and, indeed, of any application) by relatively small changes in an architecture. Specifically, we show how a change to Model-View-Presenter (MVP) with "bundling" or "dropping" strategies can improve the energy performance of apps.

But why focus on mobile? By 2019, the number of global smart-phone users is expected to reach 2.7 billion[237]. Smartphones are essentially portable networked pocket computers powered by batteries [199]. Smartphone apps range from email apps to games, to notifications, prompts, reminders, stock tickers, etc. This wide variety of software and uses is ever present as the network bandwidth demands on mobile networks starts to eclipse PC network bandwidth [53]. This pressure on functionality and portable computing puts a huge strain on a mobile device's battery, which unfortunately has not seen much technological improvement [28]. If the device's battery energy is consumed, the device is typically unusable. The importance of energy consumption on mobile devices has immediate consequences: app developers quickly learn that their apps that use lots of energy suffer in ratings [251] as consumers highly value battery life for their mobile devices [253].

We seek to aid developers in addressing energy consumption at design time, *before* runtime. Our concern is that developers do not have good guidelines or evidence-based models of the costs and benefits of the design choices they make in the design of energy efficient apps. Developers lack knowledge of architectures, patterns, and tradeoffs that are potentially "green" (energy efficient), or the parameters that can make an architecture "green" at runtime. So in this work, we demonstrate how the Model-View-Presenter pattern (MVP) can be modified to reduce the event processing overhead of model objects notifying view objects. We discuss how to modify the presenter of MVP into a proxy that bundles requests or drops redundant requests by delaying notifications—thus avoiding frequent expensive intermediate notifications or context switches that update views. Our research questions and contributions include:

RQ 1: What is the impact of the number of event sources and event generation rates on software energy consumption?

RQ 2: Can bundling and dropping events help in saving energy while varying the numbers of sources and rates?

Contribution 1: We developed a benchmark Android app that follows the Model-View-Presenter (MVP) architecture to understand the impact of bundling and dropping on Model-View-Presenter architecture. We implemented the presenter in three different forms: no bundling, bundling, and dropping. The number of event sources and the rate of event generation (i.e., number of events/second) are determined at runtime with user input. Using the benchmark app, we answer RQ 1 and RQ 2.

RQ 3: What are the energy impacts of bundling and dropping on real-world applications?

RQ 4: Can bundling and dropping help address users' feedback without harming apps' energy consumption?

Contribution 2: We confirm the realism of the findings from the benchmarks with four different real-world Android apps to answer RQ 3 and RQ 4. Because benchmark apps, although good for conducting controlled tests, do not necessarily reflect real-world scenarios and performance [211].

RQ 5: Why do bundling and dropping save energy?

Contribution 3: We investigate the cause of performance changes by analyzing resource access patterns (e.g., CPU use) of apps with bundling and dropping to answer RQ 5.

RQ 6: What are the maintainability consequences of implementing bundling and dropping on Android apps?

Contribution 4: We analyze the difficulty of incorporating bundling/dropping in Android apps, and the consequences of these changes on maintainability, to address RQ 6. Decoupling Level (DL) metric [173] is used for analyzing the maintainability cost of bundling and dropping versions.

We show that a small change to an architecture like MVP allows for making energy consumption tradeoffs, allowing for energy-aware decision making during design and maintenance phases. In general, developers can save significant amount of energy by adopting the proposed bundling and dropping mechanisms—without harming user experience and without materially affecting the maintenance cost. To support reproducibility and extension, our energy measurements and the open-source benchmark app are shared publicly [85].

8.2 Background

8.2.1 Energy efficiency is difficult to achieve

Power (P) is the rate of work expressed in *watts*. Energy (E), expressed in *joules*, is the total amount of work in a given time (T): $E = P \times T$. Energy consumption is linearly proportional to the run-time of a component, but only when P is constant. A reduced time T can save energy, but what if the CPU switches to a higher power consuming state for a reduced time

T? Without actual energy measurements or estimates, this is hard to answer. Moreover, CPU access patterns are just one of the many considerations that affect energy consumption in modern devices [142], [269]. Studies have recommended energy efficient Java collections [95], [196], energy-efficient communication protocols [50], [141], locating and finding energy bugs [10], [149], [195], [249], and building models and tools for energy estimation [40], [45], [62], [71], [90], [92], [184], [227], [269]. Despite the increasing amount of energy efficiency research, it is often unclear how software design decisions impact energy consumption [210], and what tradeoffs developers should be aware of [159], [191].

8.2.2 Model-View-Presenter

Model-View-Presenter (MVP) is a form of Model-View-Controller (MVC) [72], [200]. MVC often uses a design pattern such as the observable pattern to ensure synchronization between data in the *model*, and visual or concrete representations in the *views*, while shielding the model from direct manipulation from view objects via a *controller*. MVC has many variants. Some have different purposes. One popular variant of MVC is called *active MVC* [152], that is typically implemented with a single process whereby the observer pattern is used to allow interactions between the *model* objects and the *view* objects. In Active MVC, model objects are observables that notify observers (views) when their representation or data is updated. This is done by keeping a list of observers and then notifying each via a method call that the observable they are watching has been updated. It is then up to the observer to query the model objects for the information they need. This can be quite cumbersome as every change can cause a cascade of observers to react and deal with each change, regardless of the granularity or usefulness of the change. Another problem with this pattern is that it puts the notification and listener logic into model objects.

Active MVC is cumbersome and requires many model objects to keep track of observers. *Model-View-Presenter* is a variant that uses the observer pattern, but it provides a *proxy* (presenter) between the model and the views. The model objects, when modified, updates the presenter. The presenter notifies views and provides them with the information they need to update. The views do not necessarily need the model objects as the presenter is in the way, thus isolating the model objects further from views, while removing the responsibility of model objects to notify views for updates. The controller part of MVP is often folded into the presenter object itself. Using this presenter as a *proxy* allows one to put delegation logic into the presenter and keep that logic out of the model objects. This means that a presenter could, for example, bundle updates or drop updates that were deemed irrelevant. Both of these choices could improve the runtime behaviour of an application. Because of its simplicity, the MVP pattern has been recommended in several developers' blogs and discussions [21], [36], [203]. Our approach, however, can also be adopted with other architectures besides MVP.

8.2.3 Events, bundling and dropping presenters

An event can be a database update request, a packet transmission request, a view update request and so on. *Bundling* is the act of storing and queuing incoming events such that they can be processed together, even periodically. *Dropping* is the act of keeping only the last incoming event to be processed—periodically, or on demand, it processes only the most recent event. A *bundling presenter* stores incoming events and send them later in a single batch for processing them together. On the other hand, a *dropping presenter* discards previous events and processes only the most recent one. Bundling is applicable when a delay in processing is acceptable, whereas dropping is relevant when, along with the delay tolerance, the most recent event nullifies the importance of previous events.

Bundling has been found energy efficient in earlier studies. Pathak *et al.* [192] proposed I/O bundling for reducing tail energy leaks in mobile apps. The authors found that some hardware components, such as the network interface card or SDCard, suffer from the tail energy phenomenon. Tail energy is the wasted energy by a component while transitioning from the active to

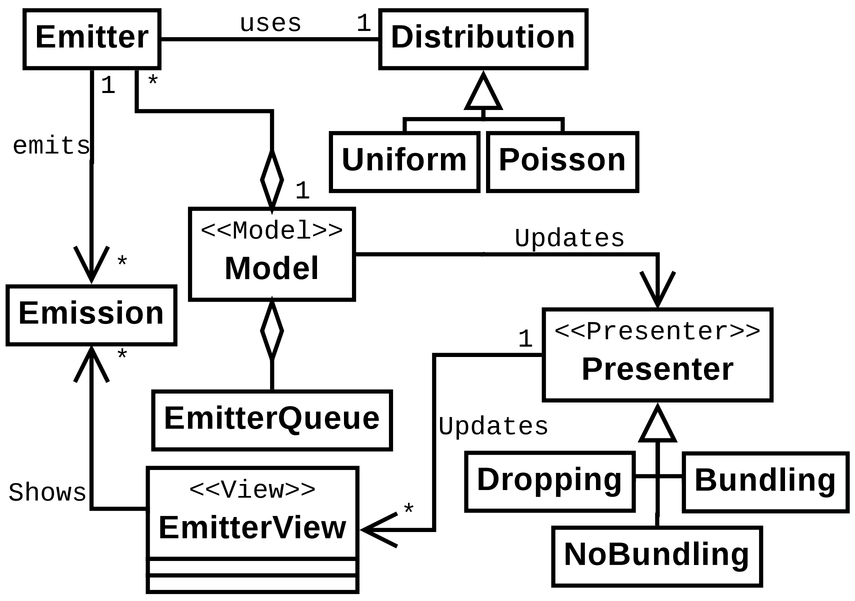


Figure 8.1: UML class diagram of the benchmark app.

the inactive state. Bundling operations that involve such hardware devices reduces the tail energy phases significantly, and thus save software energy consumption. For the same reason, Chowdhury *et al.* [46] found that writing log messages in batches can reduce energy consumption. Other research found that `HTTP/2` servers can reduce clients' energy consumption by enabling a form of bundling, compared to the `HTTP/1.1` servers [50]. Lyu *et al.* [154] has shown that energy efficiency can be significantly improved by grouping multiple database auto-commit transactions into a single transaction.

In this paper, however, we focus on modifying an existing architectural pattern (MVP) with dropping and bundling. We show that by adopting this modified architecture one can gain the maintainability and architectural benefits of MVP. Yet developers may still decide how to balance event-based energy consumption against other qualities such as latency.

8.3 Methodology

This section describes the benchmark app we developed for our experimentation, along with our energy measurement process and test scripts for driving the subject apps.

8.3.1 The benchmark app

This app has three major components including Model, View, and Presenter in compliance with the MVP pattern. Figure 8.1 illustrates the benchmark app with a class diagram.

The benchmark app, with an UI, allows a tester to choose a configuration of parameters to test. For example, the number of emitters (i.e., event sources), event generation rate, test run duration, and the version of the presenter—no bundling, bundling, or dropping—can be selected at run time. For bundling and dropping, a delay parameter is also provided, i.e., how long the app should wait for collecting the incoming events before processing all the saved events in a single batch? This UI, with a button click, can then spawn a new experiment running on a thread separate from the UI thread [202]. The new experiment will have emitters and views of emitters' emissions instantiated.

Model

The model is a collection of emitters objects (i.e., event sources) based on the user's input. The model is responsible for dealing with the emitters and forwarding their emissions to the presenter. The model has a *registerObserver* method which can add any number of presenters that can be interested in an update from this model. However, for simplicity, we used only one presenter in our experiments. The model is an observable from the Observer pattern. The model component uses four different sub-components: Emitter, Emission, Distribution, and EmitterQueue.

Emitter An emitter is an event source that emits events at a given rate (i.e., number of events/second). Each emitter creates emission objects that contain all the data of the next scheduled transmission from that emitter. The emitter is also responsible to notify the model about emissions, so that the model can notify the interested presenter. Emitters are meant to simulate event sources like stock prices, weather information, or sensor output.

Emission An emission is an event that contains some data, usually a message. These messages are randomly generated and timestamped.

Distribution Each emitter produces emissions following a probability distribution function (PDF) for scheduling the next emission. The benchmark app is designed to accommodate any PDF at run-time. For simplicity and low variability in our energy measurements, we used only the uniform distribution.

EmitterQueue The EmitterQueue uses a priority queue (Java's PriorityQueue) to schedule emitters for emitting and transmitting the next emissions. The priority queue enacts an efficient algorithm for scheduling the emitter. The EmitterQueue sorts all the emitters based on their next waiting time. The model then removes the first emitter from the priority queue, finishes its transmission, and then insert it again based on its next scheduled emission time. This process continues until the test run duration expires.

Presenter

The presenter is an observer of the model component. It is notified whenever one of the model's emitters transmits. The presenter maintains a mapping of emitters and views, which the presenter uses to notify the view of the corresponding emitter with the emission. There are 3 kinds of presenter used in both the benchmark App and the study: i) No bundling—forwards the update immediately; ii) Bundling: waits for the given bundling time, saves all the incoming updates, and forwards each of them all together; iii) Dropping: same as the bundling except the presenter forwards only the most recent update and discards all the previous updates.

The presenter runs in a separate thread than the UI thread. When the presenter receives an update, it decides which view to notify and passes off the necessary information to the view in the view's thread—such as the UI thread if the view has a UI. The bundling presenter, for example, sleeps inside a timer thread and stores the incoming events in parallel. It then forward all the stored events to the interested views once the sleeping time is over (i.e., the bundling time provided by the user). The dropping presenter is identical except it discards previous events and only forwards the most recent one. To help practitioners for implementing bundling/dropping presenters, we made the benchmark app public and open-source [85].

View

Views are meant to receive updates from the presenter. What they do with the update is up to them, but typically they only talk to the presenter and updates them with the emission objects they receive. They are observers of the presenter but might be associated with a particular object. For simplicity, the benchmark app maintains a one-to-one relationship between the emitters and view components—a single view, a *textfield*, is interested in a single emitter. A view in the benchmark app is thus responsible to display the received emission data from a emitter through the presenter.

8.3.2 Energy measurements and test scripts

We used two implementations, to verify the generalizability of our proposed approach, of the *GreenMiner* [102] software energy measurement platform to measure the energy consumption and resource usage of the apps used in this paper. The *GreenMiner's* tests and measurements can be accessed remotely, and *GreenMiner* has been used extensively in a variety of software engineering energy consumption research [4], [45], [48], [50], [95], [102], [164], [204], [207].

The system under test is typically an Android smartphone. Energy is measured using current sensor INA219 and INA159 chipsets that report to an Arduino Uno microcontroller. The microcontroller processes and aggregates measurements, sending to the test computer—a Raspberry Pi model B computer. The current sensors and the Pi are connected to the phones under test. The first *GreenMiner* is connected to 4 Galaxy Nexus phones (system-under-test) running Android 4.4.1 with an INA219, while the *GreenMiner-2* is connected to an ASUS ZenFone 2 running Android 5.0.2 with an INA159. For a given app and test, the Pi acts as the test-runner which pushes, runs, and collects measurements for a given test script. The first *GreenMiner* system has four identical settings with four Galaxy Nexus phones. Running different tests in parallel helps accelerate the measurement process. *GreeMiner-2* has only 1 ASUS Zenphone 2. *GreenMiner* test framework cleans any previously installed apps before running a new test. This is to ensure the same system state for each test; energy consumption of a particular test is therefore unaffected by previous tests. We ran the real-world app tests on the *GreenMiner* and *GreenMiner-2*; the benchmark app was tested solely on the *GreenMiner* to simplify comparison of parameters and energy consumption.

Previous *GreenMiner* based works [4], [45], [48], [50], [95] recommended running the same test multiple times, for the observed variability in energy consumption. We ran all versions of our subject apps (benchmark and real world apps) **10** times. To minimize outlier effects, we show the 99% confidence interval of our measurement distribution. In addition, whenever necessary, we used the Kruskal-Wallis test [104] to verify if two energy measurement distributions are statistically different. The advantage of Kruskal-Wallis test is that it does not assume any distribution of the data as it is non-parametric.

To measure the energy consumption of an app, we need to run the app multiple times, which is infeasible with manual testing. We used the Android's `adb shell` [19] script for writing the test cases. For the benchmark app, writing the test script was straightforward. The script selects the number of emitters, the presenter type, the bundling time in case of bundling and dropping, and provides the test duration and event generation rate before clicking the start button. For the real-world apps, however, the tests were written with two of the authors' consensus that these tests represent an average user's interaction with these apps.

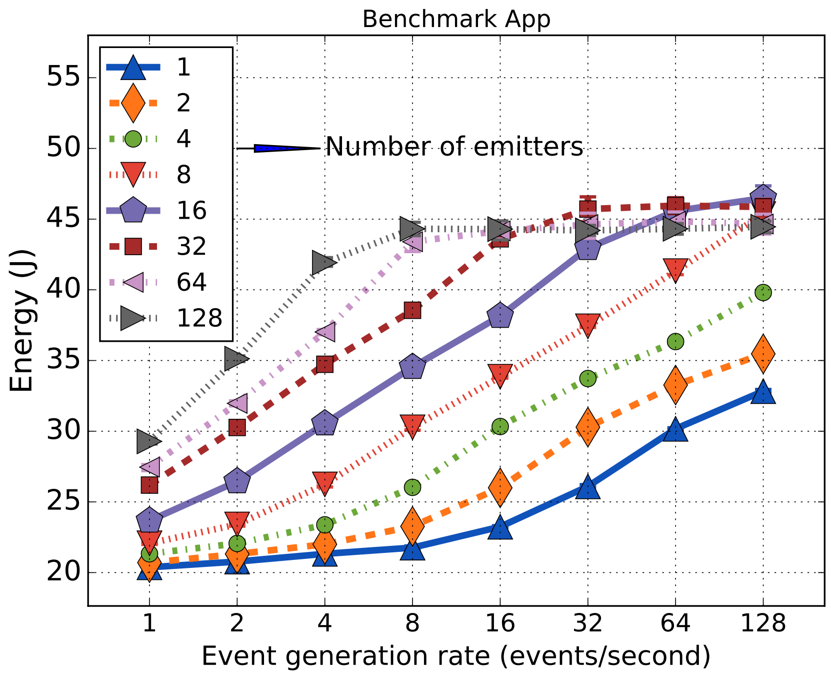


Figure 8.2: Energy consumption of the benchmark app with different numbers of emitters and event generation rates. Bars indicate the 99% confidence interval.

8.4 Results: benchmark app

In this section, we show the energy consumption of the different versions of the benchmark app with different settings. To select the number of emitters, and event generation rate, we use powers of 2: 1, 2, 4, 8, 16, 32, 64, and 128. This large range is able to show the big picture: the impact on energy consumption with the increase in the number of emitters and event generation rate. All versions (at all settings) of the benchmark app were run for a fixed period of 20 seconds, repeating 10 times each.

RQ 1: What is the impact of the number of event sources and event generation rates on software energy consumption? Figure 8.2 shows the energy consumption of the benchmark app with different numbers of emitters (i.e., event sources) and event generation rates. Clearly, the energy consumption goes up when we increase the number of emitters and/or the event rate. The Spearman [136] correlation coefficient is 0.66 ($p \approx 0$) between the number of emitters and energy consumption in *joules*. The coefficient is 0.69 ($p \approx 0$) between the event rates and energy consumption. The coefficient would have been higher if the phones were able to process high numbers of events and emitters. The variations in energy measurements among multiple runs for each setting are small; Figure 8.2 shows that the 99% confidence intervals are not noticeable until the performance is saturated. This is because we kept the benchmark app as simple and deterministic as possible, which is harder to control in real-world apps. We also observe that for high number of emitters (i.e., $\geq$32 emitters) the energy consumption does not change with the increase in the event rate after a threshold. This is because of the limited capacities of the phones we used for our measurements; these phones can process a certain number of events within the allotted 20 seconds test duration. Producing more events than this threshold does not impact the energy consumption, for the phone can not process the extra events within the allotted time. In fact, with the Kruskal-Wallis test, we found statistical differences between the energy measurements until the numbers of emitters and rates are high. For example, with 8 emitters, $\alpha = 0.05, p = 0.0001$ between 32 events/sec and 64 events/sec. However, for $\alpha = 0.05$, p is statistically insignificant (0.6242) between 64 events/sec and 128 events/sec when the number of emitters is fixed to 64.

Table 8.1 shows the percent increase in energy consumption in the number of emitters and event generation rates compared against the energy consumption of one emitter with one event per second. For example, even with a single emitter, the energy consumption can go up 38% when the event generation rate is high (128 events/second). And note that the percentage increase with high numbers of emitters and rates would have been much higher than the reported values if the phones were to able processing more events.

Findings: Many modern applications deal with large numbers of event sources with high numbers of incoming events [9], [135], [226]. Our results show that energy consumption is correlated with both the number of event producers and the rate of event production.

Table 8.1: Percent increase of energy consumption compared with the energy consumption of 1 emitter and 1 event/second. For readability, nearest integer values are presented.

Emitters	**Rates**							
	1	**2**	**4**	**8**	**16**	**32**	**64**	**128**
1	0	2	4	6	12	22	32	38
2	2	4	7	12	22	33	39	43
4	5	8	13	22	33	40	44	49
8	8	13	22	33	40	46	51	55
16	14	23	33	41	47	53	55	56
32	22	33	41	47	53	55	56	56
64	26	36	45	53	54	54	55	54
128	30	42	51	54	54	54	54	54

RQ 2: Can bundling and dropping events help in saving energy while varying the numbers of sources and rates? To answer this question, we considered three different waiting times for both bundling and dropping: 0.1 second—the corneal reflex time of human eyes; 0.5 second—half of user-acceptable latency; and 1 second—the broadly used acceptable latency target for interactive applications [168]. Unlike the real-world apps presented later, a wider range of waiting times were not considered for the benchmark app so the graphs are readable.

Figure 8.3 shows the energy savings of different bundling and dropping rates compared with no bundling or dropping (presented as Nobundling in the figures). Each graph shows the results for all the possible scenarios for a fixed number of emitters. Except for very high numbers of emitters or rates, the energy consumption goes up monotonically for the Nobundling version. While this trend is true for the bundling versions as well, for the dropping versions we do not see such clear trends. This is because the number of events processed by a dropping version (transferring events from the presenter to the views) is to some extent independent of the number of events generated. The small energy increase for the dropping versions with increased rates is due to the cost of producing more events by the model component in our benchmark app. Not surprisingly, dropping is more energy efficient than bundling;

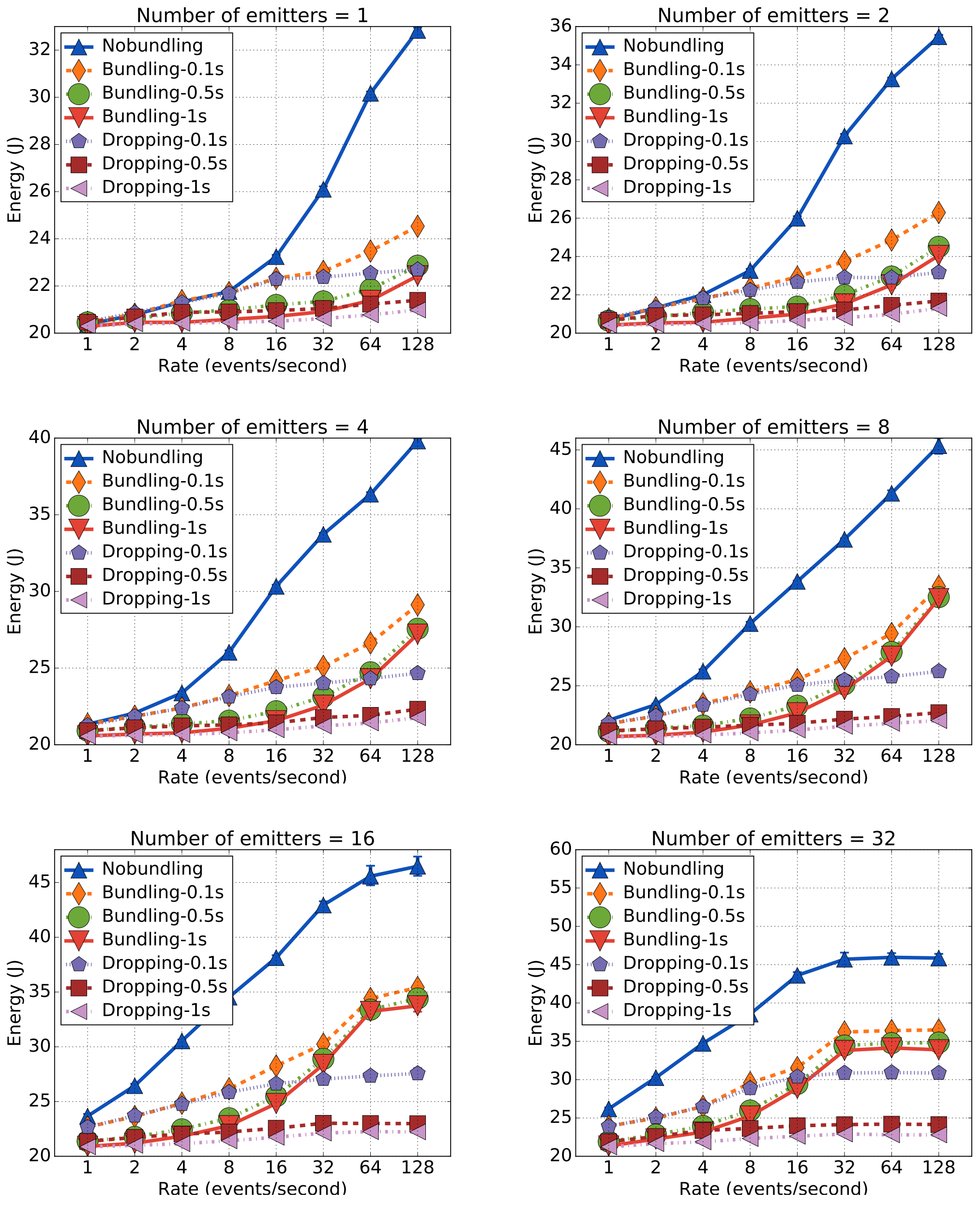


Figure 8.3: Energy consumption of bundling and dropping compared with the Nobundling versions for different number of emitters. Results for more than 32 emitters are not presented; due to resource limitations, the energy consumption is inconsistent for very high numbers of emitters and rates. Bars indicate the 99% confidence interval.

Table 8.2: Energy savings (in percent) by different bundlers and droppers when compared with no bundling or dropping. Results are presented for just one emitter.

	Rates							
Versions	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Bundling-0.1s	0	0	0	0	3	13	22	25
Dropping-0.1s	0	0	0	0	4	14	25	30
Bundling-0.5s	0	0	2	3	8	18	27	30
Dropping-0.5s	0	0	2	3	9	19	29	34
Bundling-1s	0	1	4	5	10	19	29	31
Dropping-1s	0	1	4	5	11	20	31	36

the dropping versions process fewer events than the bundling versions. The bundling versions, in spite of processing the same number of events as the no bundling version, can save significant energy.

It is encouraging that, with bundling, we can process and deal with the same amount of workload, and yet can make apps significantly more energy efficient. Table 8.2 shows the energy savings by different bundlers and droppers with fixed one emitter. This is to ensure that the energy consumption is not affected by resource limitations, thus enabling accurate comparison. It shows that with bundling (doing all the work without dropping anything) and maintaining a latency such that a user does not notice any change (0.1 second), we can still save up to 25% of the energy (Bundling-0.1s). With user-acceptable latency (Bundling-1s), bundling can save up to 31% in a simple app like our benchmark, with just one event source. We verified with the Kruskal-Wallis test that these differences are indeed statistically significant (with $\alpha = 0.05, p \leq 0.01$).

Findings: Dropping is the most energy efficient approach compared to bundling and no bundling. However, dropping might not be an acceptable alternative when accuracy is important. Application developers can consider bundling in such scenarios, which still saves significant energy over no bundling.

Table 8.3: Description of the selected four real-world Android apps from F-droid.

App	Type	# Classes	ULOC	Test scenario	Test duration (s)
Sensor Readout [218]	"Real-time graphs of sensor data"	56	6009	Measure the Gyroscope sensor	70
ColorPicker [56]	"Pick colors and display values"	12	908	Move the scroll bars for R,G,B	50
Angulo [22]	"Angle and Distance Measuring"	4	497	Start the measurements and wait	55
AcrylicPaint [2]	"Simple finger painting"	7	936	Draw a hexagon	27

8.5 Real world apps

Results from the benchmark app shows that bundling and dropping can save significant energy in Android apps. And this saving is larger with increased numbers of event sources or event generation rates. It is, however, not obvious how such optimization approaches would perform in real-world apps [211]. In this section, we evaluate bundling and dropping in four selected real-world Android apps.

8.5.1 Selection of applications

The apps we selected had to be open source so that we can implement bundling and dropping. We explored the F-Droid repository [68] to find suitable apps. F-Droid contains source code for all the posted Android apps and was used in earlier mining software repositories research [30], [134], [255]. Finding suitable apps with reasonably small code size (so that we could easily identify where to implement bundling and dropping) was challenging, which hindered us from analyzing more apps. Table 8.3 shows the characteristics of the four selected apps.

The different types and code sizes of these four apps enables reliable evaluation of bundling and dropping. The Sensor Readout app is also available on Google Play [81] and has been downloaded more than 50,000 times (as of writing). This app has received 540 reviews with an average rating of 4.3/5. This allows evaluating energy optimization techniques for apps that are already popular. AcrylicPaint, a finger painting app, represents apps where

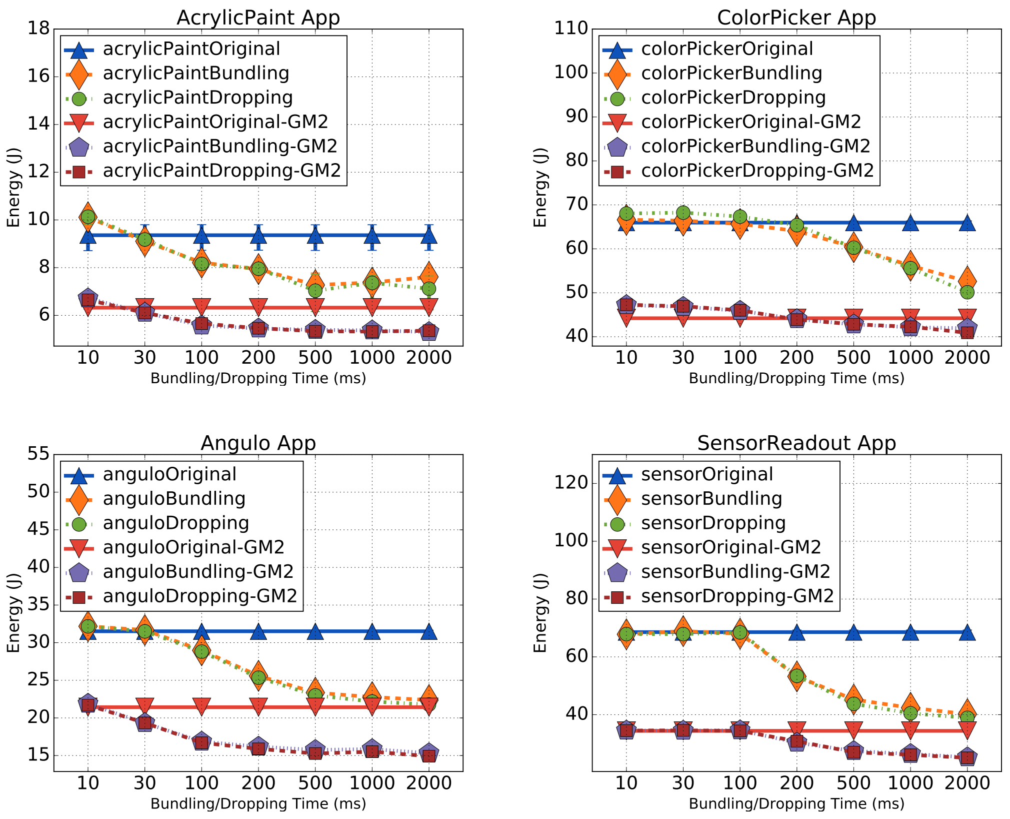


Figure 8.4: Energy consumption of bundling and dropping compared with the original versions of four real-world apps. Bars indicate the 99% confidence interval.

users might spend more continuous time, making energy optimization more crucial.

We have implemented the bundling and dropping versions of these apps, except for Sensor Readout, following the approach presented in section 8.3.1. The original Sensor Readout app, uses a timer function, and processes only 10 measurements per second, although the app samples measurements continuously. As a result, we did not have to implement our own timer for the bundling and dropping variants. The apps, however, did not follow a clear MVP pattern. Instead, their designs were closer to the MVC pattern. We have identified which classes contained the actual processing code, and refactored those classes to accommodate our bundling/dropping presenters. Our intention was to convert the existing design as close to the bundling MVP pattern as possible. For ensuring correctness, two of the authors were involved in refactoring and testing the apps afterwards.

8.5.2 RQ 3: What are the energy impacts of bundling and dropping on real-world applications?

The energy savings from bundling and dropping are of course impacted by the bundling/dropping time—the time these two variants wait before processing a batch of events. We selected six different times: 0.01s—fastest human time perception; 0.03s—animation speed; 0.1s—the corneal reflex time of human eyes; 0.2s—double the corneal reflex time; 0.5s—half of user-acceptable latency; 1 second—acceptable user latency [168]; and 2 seconds—double user-acceptable latency. Figure 8.4 shows the energy consumption of bundling and dropping compared with the original versions of the four selected apps. Here, we also show the results for the *GreenMiner-2* (GM2) on the Asus Zenphone 2.

Except for the Sensor Readout app, we observe more energy consumption for the bundling and dropping versions for very low bundling and dropping times (e.g., 0.01s). This suggests that running a timer thread for bundling or dropping incurs energy consumption overhead. As we mentioned before, Sensor Readout did not require a separate timer thread and does not have this overhead when bundling or dropping is used. For all the apps, however, the energy consumption of bundling and dropping improve significantly when the waiting time is reasonably higher. For example, even for 0.1s latency which is difficult for users to perceive, the bundling and dropping versions of Acrylic-Paint and Angulo can save 12% (12% with GM-2) and 9% (8% with GM-2) energy consumption respectively, when compared with their original versions. The energy savings become significant for larger latency. For example, we can save 37% (24% with GM-2) energy consumption for the Sensor Readout app with a 1s latency in drawing the measurements graphs, and that without losing any measurements (i.e., with bundling).

GreenMiner-2 (with the ASUS ZenFone 2 phone) consumes less energy than the *GreenMiner* (with the Galaxy Nexus phone) for all apps, and thus the energy savings are generally lower. The trends in percent of energy consumption reductions, however, are similar across the apps for both the GreenMiners.

8.5.3 RQ 4: Can bundling and dropping help address users' feedback without harming apps' energy consumption?

Answering this question might require analysis from multiple perspectives. However, with one case study, we show that there are scenarios where the developers can adopt our bundling approach to address user feedback that involves energy expensive modifications. For this study, we selected the Sensor Readout app—the only app available on Google Play with a significant number of reviews. For a selected sensor type, this app shows/updates 10 measurements per second.

In general, this app is praised by users. However, some reviews indicate user dissatisfaction. For example, one user desires to see colour changes with different measurements while updating graphs [218]. Another user is unsure why the sampling rates from the sensors are low—10 samples per second.

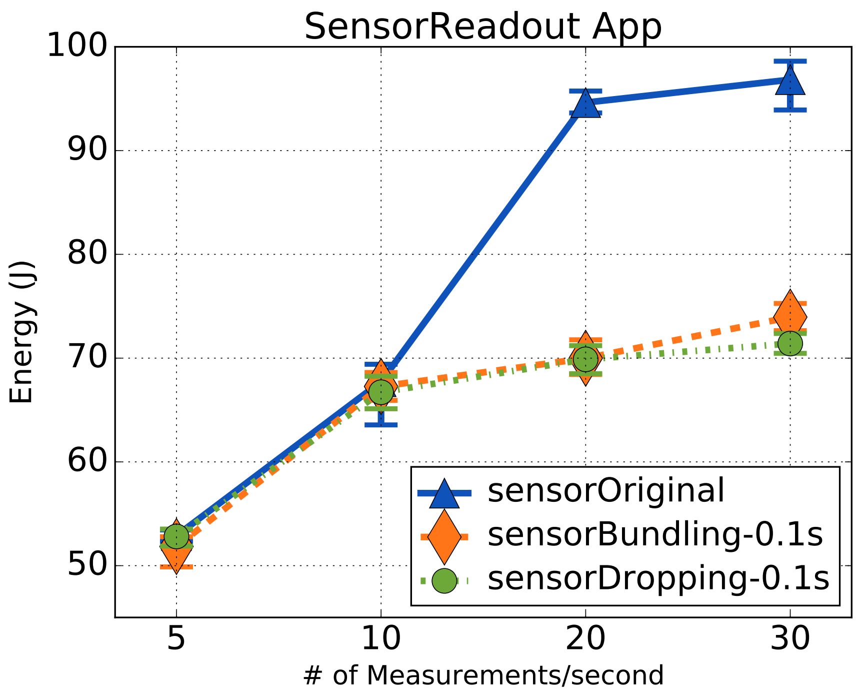


Figure 8.5: Energy savings of bundling/dropping for the Sensor app with higher sampling rates. Bundling/dropping time is fixed to 0.1 second. Bars indicate the 99% confidence interval.

Changing colors continuously and sampling at higher rates might increase the energy consumption of the app significantly. And yet, developers need to carefully address user feedback for their apps to stay popular [190]. In fact, there are reviews on the Sensor Readout app suggesting similar apps are supposedly better than Sensor Readout. We show that Sensor Readout can benefit by applying bundling—by sampling measurements faster for timeline graphs, but delaying graphical updates by only 0.1 second without harming latency (more measuresments per second but constant UI update rate).

Figure 8.5 shows the energy consumption of the original version compared with the bundling and dropping versions, with different sampling rates. The bundling/dropping time (i.e., the latency in updating the graphs) is fixed to 0.1s. The average energy consumption of the original version is high (≈94 joules) when the sampling rate is 20/second compared with the original 10/second (≈69 joules); a 36% increase in energy consumption. The difference is also statistically significant (Kruskal-Wallis test, $\alpha = 0.05, p < 0.01$). However, bundling with 20 samples/second consumes similar energy to the original version with just 10 samples/second. With sampling rate higher than 20, the energy consumption of the phones does not increase as expected. The Galaxy Nexus phones are unable to process more than a threshold number of samples.

Findings: Real-world Android apps can save significant energy with bundling and dropping. Sacrifices in latency correlate with the energy savings. Bundling with almost imperceptible latency (0.1s) can save energy without affecting user satisfaction. In some scenarios, bundling and dropping can help developers address users' concerns with no meaningful sacrifice in usability.

8.6 Understanding resource utilization patterns with bundling and dropping

It is unsurprising that dropping saves energy; in dropping the presenter only sends the most recent event for processing. This requires less CPU slots for the process (also known as CPU jiffies [45]). Bundling, however, does the exact same amount of work as on-time processing. Thus the question arises: *why does bundling save energy in spite of processing all the events?*

Assumption: A CPU jiffy is an assigned CPU time slot for a process in Linux [45], [48]. More CPU jiffies for a process causes more CPU jiffies for the kernel, because of more context switches between the user space and the kernel space. In on-time event processing system, each event requires at least one user CPU jiffy. This incurs at least one context switch and one kernel CPU jiffy. This effect, however, can be minimized with bundled processing. Batching of events minimizes the number of context switches and the number of CPU jiffies. If our assumption is correct, the energy efficiency of bundling is explainable. The number of CPU jiffies and context switches are almost linearly correlated with software energy consumption [45].

8.6.1 RQ 5: Why do bundling and dropping save energy?

To verify the above assumption, we have analyzed the AcrylicPaint app. Similar to Chowdhury *et al.* [45], we used the Linux proc file system. To capture the CPU jiffies used by an app, we used `/proc/pid/stat`. This also includes the kernel CPU jiffies used for that app. However, app (process) specific context switches can not be captured using such a file system. We captured the number of context switches from `/proc/stat` before and after running a test for an app. The difference is thus approximately the number of context switches for the app.

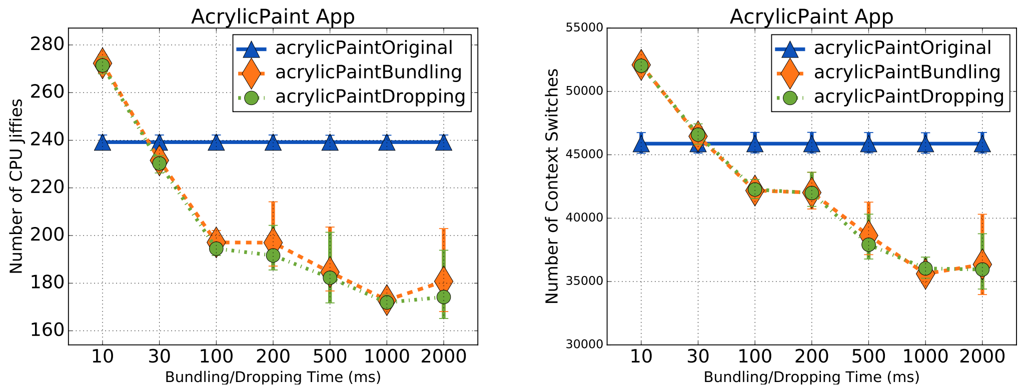


Figure 8.6: Numbers of CPU jiffies and context switches for bundling and dropping compared with the original AcrylicPaint app. Bars indicate the 99% confidence interval.

Figure 8.6 shows the result (10 measurements for each configuration). The number of CPU jiffies and context switches follow a similar pattern to the energy consumption of the AcrylicPaint's versions (Figure 8.4). This observation suggests that our assumption is true: bundling indeed reduces the number of CPU jiffies and context switches. This also indicates that bundling and dropping enable efficient resource usage, and thus can potentially provide similar energy savings for platforms other than Android. The mechanisms of context switches between the kernel and user space are similar across different platforms and architectures.

Findings: Bundling and dropping access resources in efficient ways—reducing the need for many context switches, leading to energy efficient software. This observation suggests that the energy efficiency of bundling and dropping is not restricted to Android systems, but also is applicable in other platforms.

8.7 Maintainability analysis

Developers and architects need to be concerned with the maintainability consequences of changes made to enhance any single aspect of a system's quality. Thus it is important to understand the consequences of the changes made to the apps to implement the bundling or dropping strategies. This motivates our sixth and final research question.

RQ 6: What are the maintainability consequences of implementing bundling and dropping on Android apps?

To assess these consequences we analyzed the before and after versions of the four selected real-world apps. We first reverse-engineered each of the apps using the *Understand* tool [217]. Using this tool we were able to collect code metrics on all versions of the apps. (For the purposes of brevity we only report on the original and bundling versions of the apps here. Results for the dropping versions were very similar.) In each case the modifications to the apps were slight in terms of effort, requiring fewer than 100 additional lines of code and at most three new classes (including one Handler class and one Thread class that runs the timer). But counting the lines of code and classes is just the measure of the required effort for converting a typical app version to a bundled version. Another important question is whether these changes affected the long-term maintainability of the system. If, for example, we added few lines of code but added many new dependencies between classes, this would increase coupling in the system, negatively affect the maintainability of the app going forward.

To determine whether this was the case we analyzed the coupling of the apps, in their before and after versions, using the Decoupling Level (DL) metric [173], a system-wide measure of coupling. The DL metric has been em-

Table 8.4: DL values for before/after versions of each app (bundling only).

App	**DL Score** Original	**DL Score** Bundling	**DL Δ**
Angulo	68%	69%	+1%
ColorPicker	17%	17%	+0%
AcrylicPaint	88%	82%	-6%
Sensor Readout	32%	30%	-2%

pirically validated [173] and shown to be more reliable than other coupling metrics such as Propagation-Cost [155] and Independence-Level [220] in predicting maintenance effort. DL scores range from 0 to 100, and the higher the number the better, as this indicates that the system's files are more highly decoupled and hence can be independently modified. The purpose of using this metric is to determine if the changes made to address energy efficiency significantly lowered the value of the DL metric. If so, this would mean that the maintainability of the system was negatively impacted by the energy-saving modifications.

The DL values of the before and after versions of four apps are shown in Table 8.4. While the values of the DL metric varied widely (indicating the inherent maintainability of the apps prior to our intervention) the changes for the apps due to the addition of bundling were small. The observed drops in DL scores were due to new relationships between classes that the bundling and dropping functionality required. But since the DL scores do not change dramatically (decreasing about 6% for AcrylicPaint, increasing 1% for Angulo, and staying the same for ColorPicker), this indicates that the tradeoffs made for energy efficiency were generally good ones—improving the energy efficiency of the apps while sacrificing little, if any, maintainability of the apps for the long term. In fact, in [173], it was noted that small variations in DL ($\leq 10\%$) are typically not meaningful.

Findings: Energy efficiency is largely ignored during software maintenance [159]. One reason could be the difficulty in fixing energy bugs [44]. Bundling and dropping, however, are easy to implement and maintain.

8.8 Threats to validity

External validity is hampered by the single version of the Android OS that we used on four Galaxy Nexus phones. Also, we do not know how many real-world apps can directly take advantage of the proposed bundling approach. The first threat is mitigated somewhat by using the *GreenMiner-2* with a different phone (Zenphone 2). To mitigate the second threat, we tried to select apps from different domains, and with the context-switching analysis we explained why bundling is energy efficient. This might help predicting what other types of apps can adopt GreenBundling.

Internal validity can be criticized for the way we calculated the number of context switches. Unlike the CPU jiffies, process-specific context switches are inaccessible using the `procfs` file system. The difference between after and before when running a process can be affected by other processes (e.g., garbage collection).

The Kruskal-Wallis test, although it does not assume any normality distribution about the data, still assumes that data in each group has similar skewness [163]. These threats are minimized by measuring each configuration 10 times and then showing the means, and confidence intervals. Construct and conclusion validity may also be questioned based on the tests scripts that we created for the real-world apps. It is not guaranteed that typical users of these apps would interact similar to the way our test scripts do. However, our test scripts exercise the main functionality of these apps: e.g., drawing measurement graphs and objects with the Sensor Readout app and the AcrylicPaint app respectively.

8.9 Related work

In recent years, developers have expressed more concerns about software energy consumption [158]. The software research community has been investigating several areas of this issue. Hasan *et al.* [95], Pereira *et al.* [196], and

Manotas *et al.* [160] have presented recommendations for selecting energy efficient Java collections. Energy efficient color transformation in Android apps was proposed by Li *et al.* [142] and Agolli *et al.* [6]. Off-loading jobs [169], prefetching content [74], and enabling ad-blockers [204] have been found to save energy in some cases. Chowdhury *et al.* [50] suggested that `HTTP/2` servers are more energy efficient, from the clients' perspective, than `HTTP/1.1` servers. Energy efficient logging techniques for Android systems [46] have also been studied.

Other research has shown correcting code smells helps to improve energy efficiency [39]. In a similar vein, the impact of code obfuscation and refactoring on software energy consumption was studied by Sahin *et al.* [212], [213]. The energy change from code obfuscation is too small to notice, whereas refactoring can impact both positively and negatively.

Developers need to measure or estimate their apps' energy consumption. Hao *et al.* [92] proposed an instruction-based energy estimation model. Machine learning based models were proposed by Aggarwal *et al.* (GreenAdvisor [4]), Chowdhury *et al.* (GreenOracle [48] and GreenScaler [45]), and Pathak *et al.* [194]. Nucci *et al.* proposed PETrA [184] to estimate Android apps' energy consumption leveraging various Android tools.

Locating software energy bugs and hotspots automatically is another important research area. Wakelock-related energy bugs have been frequently reported by earlier studies [10], [149], [195], [249]. Developers need to exploit tools and techniques to locate and solve such bugs [249]. Similar to the energy bugs, developers should also resolve energy hotspots [28]. Jabbarvand *et al.* [117] proposed a test-suite minimization approach focusing only on locating energy bugs. In their later work, the authors proposed an energy-specific mutation testing framework with high precision in detecting energy bugs [116].

This paper, however, focuses more on high-level design choices that can help developers writing energy efficient systems. To the best of our knowledge, this is the least explored area of software energy efficiency, and there is still a need for more research on this avenue. The closest to our work is the short study by Sahin *et al.* [210], where the authors investigated different existing design patterns and their energy consumption. In contrast to our work, that study lacks proper guidelines and cost analysis for making a design choice.

8.10 Conclusion & future work

In this work we showed that an architectural choice, such as choosing a bundled MVP architecture, can improve the sustainability and energy consumption of a system without negatively impacting system maintainability. The consequence of this research means that architects and developers can (and should) make design decisions to address energy consumption before they start coding.

We have demonstrated the value of a bundled presenter in MVP by first benchmarking a generic MVP architecture and then by demonstrating that the energy improvements demonstrated in the benchmark were in fact realized on real-world apps that were refactored into bundled MVP architectures from more classical MVC architectures. A significant reduction of energy consumption can in fact be achieved. Furthermore we showed that these modified apps did not seriously affect the user experience, nor did the refactored versions suffer in terms of their eventual maintainability. Thus, the energy-savings that we achieved were truly win-win.

Our final message is this: fundamental architectural choices, such as the ones we have investigated in this paper, can have substantial effects on energy consumption. Although we demonstrated our results on MVP-based architectures, it is our hope and belief that developers and researchers can use this study to motivate similar studies, allowing them to address questions of energy consumption, and their consequent tradeoffs, at design time. We do not need to wait until the app is built to make these important design choices. In our future work, we want to evaluate the proposed bundling architectures on other smartphones than Android, and with real end-users for evaluating actual usability. We also want to evaluate other architectural patterns and architectural choices so that architects can predictably translate sustainability requirements into designs and into working systems.

Part III

The Future

Chapter 9

Conclusions & Future Work

This chapter concludes this thesis by discussing the summary of the contributions, potential future works, and our concluding remarks.

9.1 Summary of the contributions

This thesis first categorizes the research areas in software energy efficiency into two broad categorizes: 1) models for estimating software energy consumption, and 2) enhancing energy optimization guidelines. The rationale for such categorization is that energy-aware developers, first and foremost, need to measure their apps' energy consumption. One obvious advantage of such measurements is, developers will be informed if a new version (before releasing it) consumes more energy than the previous versions. This would help them finding energy bugs in the new version, or making trade-offs between energy efficiency and other features. Actual energy measurement, however, is expensive and requires expertise, which led us to develop software-based energy estimation systems (machine learning based energy models) without any hardware-based instrumentations. Our energy modeling approach is reproducible because we have used common OS statistics that are accessible directly from any Linux-based systems. We also argue that, developers need recommendations for achieving energy efficiency. This thesis provided three different recommendations by empirically evaluating the energy consumption of the `HTTP/2` protocol, by analyzing the logging impact on energy consumption, and by providing an energy efficient design pattern.

9.1.1 A reproducible energy model

In order to build a reproducible energy model, we investigated if we can use common OS statistics and traces of system calls (as the independent variables in machine learning models) for building software energy models. The techniques we used for collecting the independent variables are reusable in other Linux-based systems other than Android. We have successfully developed a moderately accurate software energy model—*GreenOracle*.

Findings: A reproducible energy consumption estimation model can be built using common OS statistics and their counts. The accuracy of such models improve with more and more number of apps in the training set. However, adding more apps in training requires manually writing a test script for each of the app, because we need to automatically run each app for collecting the OS statistics and the energy consumption. This makes it difficult for making such models scalable.

9.1.2 Accurate energy models with automated test generation

We have shown that random test generation with test selection heuristics can help us build accurate software energy models. With such approaches we can add more and more apps in training and can continuously improve a model's accuracy. For test selection, however, the traditional code coverage heuristic does not perform well, rather we need heuristics that care more about exercising energy hungry source code. In the end, we have produced the *GreenScaler* model as a tool, which is publicly available [47]. Developers can download the *GreenScaler* tool for estimating their apps' energy consumption without any cost and without dealing with any hardware instrumentation.

Findings: Automatic random test generation with resource utilization-based heuristic such as CPU-utilization can help us to build accurate software energy models. With such automation, we can add more apps in training without manually writing laborious and time consuming test scripts. Code coverage does not perform well for selecting tests for building accurate software energy models.

9.1.3 Energy consumption of the `HTTP/2` protocol

The `HTTP/2` is the new standard Internet protocol. And yet, the energy efficiency of this protocol was unknown. We have measured the energy consumption of `HTTP/2` and compared it with the previous `HTTP/1.1`. We have provided recommendations for web app developers that they should employ `HTTP/2` without any concern for energy efficiency. In fact, `HTTP/2` is more energy efficient than the `HTTP/1.1`, especially for high round-trip times. We also have shown that the `HTTPS` for its encryption is more energy expensive than the plain `HTTP`.

Findings: `HTTP/2` is a free lunch for energy-aware web app developers. It never performs worse than the previous `HTTP/1.1` when energy efficiency is a concern. On the contrary, for high round-trip times (e.g., more than 30 ms) `HTTP/2` is always more energy efficient than the `HTTP/1.1`. Web app developers should switch to the `HTTP/2` protocol for making their web apps more energy efficient.

9.1.4 Energy consumption of logging

Logging is heavily used for tracking software bugs. Heavy logging, however, can negatively impact the energy efficiency of mobile apps, which is a concern for the devices' availability. We have measured the energy cost of logging on Android systems and found that logging can be energy efficient if done carefully. For example, for heavy logging, developers can bundle small log messages together so that logging becomes less frequent.

Findings: Small amount of logging (e.g., ≤ 10 messages per second) does not significantly impact energy consumption. Infrequent logging of large log messages is more energy efficient than frequent logging of small messages. Logging also has potential for understanding energy bugs.

9.1.5 Energy efficiency from the design time

Unlike benchmarking software, energy consumption of real-world software is often too high to benefit from small source code tweaks and API selections. Research should focus more on higher level changes (architectural and design levels) for efficient accesses of energy hungry hardware components. Anecdotally, there is a common belief in the community that a design good for energy efficiency is probably bad for software maintenance. In this thesis, we conducted the first study on finding energy efficient design patterns so that developers can consider energy efficiency from the design time. We also studied the possible trade-offs that developers need to be aware of while adopting a particular design choice.

We have provided the example of a bundled Model-View-Presenter where the presenter can bundle incoming view update requests from the Model and then can forward all the requests together in a batch. This approach can save significant energy consumption in real-world apps. In addition, we have shown that the proposed approach does not significantly impact the maintenance cost of the source code. It is, however, a trade-off between user experience and energy efficiency. We show that with bundle MVP for efficient view updates, developers can still save energy without significantly hurting user experience.

Findings: Energy efficiency can be addressed from the design time. For example, a bundled Model-View-Presenter can save significant energy consumption by delaying the view updates so that multiple view updates occur in a batch. This approach does not negatively affect source code maintenance, and offers ways to minimize the negative impact on user experience (i.e., updating views at every 0.1 second).

Summary: This thesis successfully demonstrated that accurate energy estimation models can be built for helping developers estimating their apps' energy consumption. Also, energy specific guidelines can be produced for guiding developers so they can write and deploy energy efficient systems. While studying the guidelines, we observed the following rules of thumb that can be followed by energy-aware software developers. 1) Developers should strive for I/O bundling. They need to bundle I/O operations so that multiple operations are completed with one I/O activation (this reduces the tail energy leaks) 2) Developers should try to reduce the number of packet transmissions. For example, compression reduces data size and can reduce the number of transmissions or packets. 3) Developers should also strive for reducing the number of context switches.

9.2 Future work

Software energy efficiency is a relatively new topic in the software research community. Although we have observed significant attention on this topic, especially in the last five years, there is still need for more research. In this section, we discuss some potential future works that we believe would be useful for the research community as well as for the energy-aware developers. In particular, two of the proposed future works are described with potential methodologies.

9.2.1 A generic energy estimation model

A common problem with software energy estimation is that the approaches are often specific for one particular device (consider PETrA [184] as an example which was described in chapter 5). Our proposed machine learning-based approach (described in chapter 4 and 5) with common OS statistics alleviates this problem, because this approach, with device specific measurements, is reproducible for a different device. Reproducing an energy estimation model for every new devices, however, might be difficult and time consuming. There is still a need for a generic energy estimation approach that can work on any devices.

While we believe developing a truly generic energy estimation approach is an open research problem, we offer a potential workaround. We postulate that an energy-aware developer should care about energy estimation for answering two particular questions: 1) how energy efficient is my app when compared with other similar apps, and 2) does the new version of my app consume more energy than the previous version?

We can conduct a new study to evaluate if a machine learning-based energy model (e.g., *GreenScaler*) can be used to answer those questions, especially when a developer do not have the same device that the model is built on. For answering question 1, we can collect different third party apps from different categories (multiple apps from each category). For apps under the same category, test cases will be written for exercising similar functionalities (e.g., add and delete items for to-do list apps). We can then measure the energy consumption of all the apps across multiple devices. Are the differences in energy consumption similar across different devices? If we run the *GreenScaler* model on those devices, does *GreenScaler* suggest similar results across different devices? We do not expect *GreenScaler* to estimate the actual energy consumption accurately for different devices, but if the differences in energy consumption are similar, that should meet the developers' needs.

Similarly, for answering question 2, we can collect energy consumption of different versions (energy efficient and energy inefficient) for multiple apps from a new device (other than the Galaxy Nexus phone that was used for building the *GreenScaler* model). On this new device, can the *GreenScaler* model still distinguish the energy inefficient versions from the efficient ones? If yes, then we can argue that developers can use machine learning based energy models (e.g., *GreenScaler*) on different devices to find out which of their app versions are energy (in)efficient.

9.2.2 Automatic energy regression testing

Software undergoes changes. Although a new version might come with new feature(s), this is not always the case. Developers might modify source code, with the same functionality, expecting better performance [153]. These changes, however, can induce harmful energy regressions [267]. Hence, before publishing an app with source code modification, a developer needs to test the modified code. This requires a test case that must execute the modified source code. Writing test cases manually, however, demands excessive manual labor [16]. Automatic test case generation relieves developers from writing labor-intensive manual test case generation. There exists tools to generate a whole test suite [172], or guided test cases to locate crashes/bugs [18], [156], [161] for Android apps. Unfortunately, there is no research on generating test cases to find energy regression between two versions of a given app. This motivates a more specialized automatic testing methodology that only executes the code difference between two versions of an app for locating the source of energy regression.

One of the possible ways to find energy regression is to generate a whole test suite—manually or with a tool like TrimDroid [172]. The whole test suite is then exercised with the two given versions. *GreenScaler* can be applied to find test cases that exhibit energy difference between the two versions. Generating a whole test suite and running it for two versions of an app, however, can be very time consuming. A recent study shows that excessive time demanding testing is not acceptable to the developers [132].

As a future work, we propose test generation only for modified source code. The hypothesis is that if two versions of an app do not differ for test cases that exercise only the modified source code, then their performance (energy consumption) should be identical considering the whole test suite. This also facilitates significantly less testing time compared to the whole test suite testing approach. By leveraging the previous works on automatic testing, we propose to develop eRED (**E**nergy **Re**gression **D**etector). Figure 9.1 depicts the proposed mechanism of eRED. Given the source code of two versions of an Android app, modified source code is identified (with `git diff` tool for example). eRED then applies program analysis techniques to understand the flow graph (e.g., how a modified method is called from other methods in the program), and an Interface model (how different GUIs are connected to each

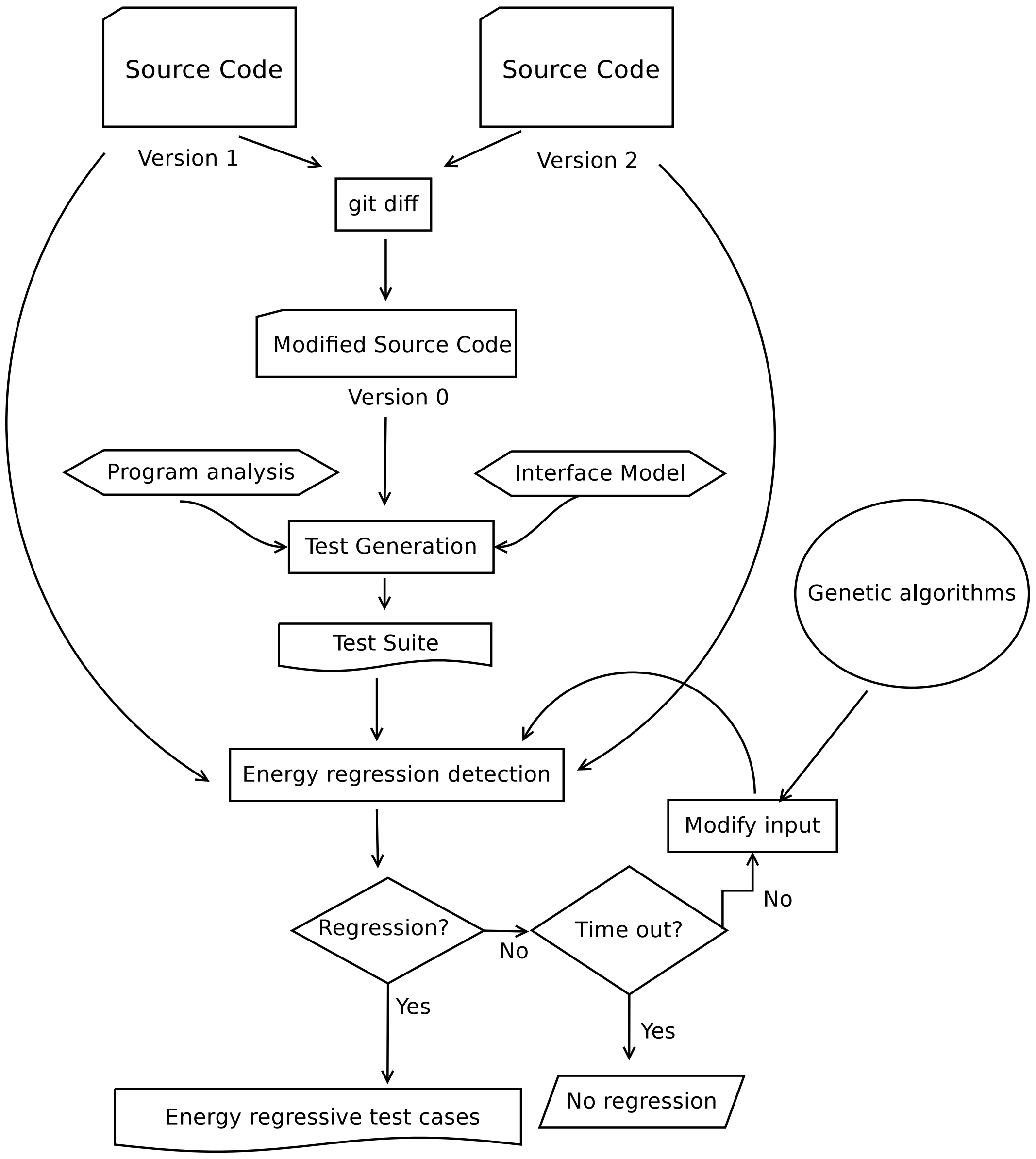


Figure 9.1: eRED: Automatic test generation to detect energy regression between two versions.

other and with the modified source code). eRED then generates different test cases that can execute the modified code. With the *GreenScaler*, eRED then runs each test cases on both the versions to check energy regression. It is, however, possible that energy regression exists only for certain types of input. For example, if code modification is performed inside a loop, the energy regression might depend on the length of the loop. Similarly, energy regression might depend on the type of input of a text field.

Genetic algorithms can be applied for modifying test input [157], [161]. In genetic algorithms, a set of candidate solutions are evaluated for a given fitness function. In case of eRED, the fitness function would be the energy consumption difference between two versions of an app after executing with a test case. From the best candidate solutions, tests with high energy difference, a new generation of candidates are formed: either by mutating over each candidate separately, or by applying cross over across different candidates. The newly formed generation is then evaluated for producing further generations until a demanded number of generations is reached.

9.2.3 Other potential research

Developers often follow some common patterns for optimizing the energy consumption of their software. For example, Cruz *et al.* [57] found 22 such common patterns by mining open source software repositories and by leveraging existing energy researches. What we do not know is how generalizable are these patterns are. Do they offer similar energy efficiency across different devices and across different usage scenarios? One really useful future work would be to empirically evaluate the energy efficiency of these common techniques across multiple devices (different smart-phones for example) and across different usage scenarios of the apps. For example, we can make two versions of an app based on an energy commit (a commit that indicates the intent of improved energy efficiency). The first version is built on source code that does not have the new code from the energy commit, but the second version does. Now after writing manual test scripts for different usage scenarios, we can run these two versions in different phones. *Do we see similar energy consumption differences across different devices and usage scenarios? Are there cases where an energy efficient pattern does not help at all? Can we build better recommendation systems (e.g., device type and usage scenario-based recommendation) for these commonly used energy optimization techniques?*

9.3 Concluding remarks

Software energy efficiency is difficult to achieve. There are just too many uncertainties. For example, should we always strive for an idle CPU? We do not know. The answer might depend on the subject app and usage scenario. An idle CPU induces less CPU energy, but it might increase the execution time. Likewise, reducing the run-time of a system does not necessarily reduce its energy consumption—less run-time might put hardware components to higher power consuming states. There are also multiple components that need to be utilized efficiently (e.g., CPU, Wi-FI, disk, and memory). It is even possible that just by optimizing the energy consumption of one component (e.g., CPU), one might increase the energy consumption of another component (e.g., memory). This is why, measuring or estimating the actual energy consumption is so important: until we know what is the actual energy consumption of a software, we can not tell if the applied optimization techniques are helping or hurting.

Developers are trying to understand this rather foreign topic of energy efficiency [159], but often do not receive enough guidelines from their peers (e.g., inadequate answers from programming questions & answer sites like StackOverflow [198]). We thus argue that academics should think about accommodating *software energy efficiency* as a topic in computing science curriculum. The research community can still do a lot for helping energy-aware developers: by providing new guidelines, and by evaluating existing guidelines on different scenarios.

We also believe that future software energy research should focus more on generalizability: when we build a new energy estimation or optimization technique, we need to evaluate its performance across multiple devices, because energy consumption can be very different on different hardware configurations and architectures.

References

[1] */proc/stat explained*, `http://www.linuxhowtos.org/System/procstat.htm`, (last accessed: 2014-May-22). 41, 53

[2] *Acrylicpaint*, `https://f-droid.org/en/packages/anupam.acrylic/`, (last accessed: 2018-Jun-02). 224

[3] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the ISSTA 2015*, ser. ISSTA 2015, Baltimore, MD, USA, 2015, pp. 83–93. 83

[4] K. Aggarwal, A. Hindle, and E. Stroulia, "Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption," in *2015 IEEE ICSME*, Bremen, Germany, Sep. 2015, pp. 311–320. 14, 18, 25, 44, 69, 79, 81

[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*, Markham, Ontario, Canada, 2014. 13–16, 25, 45, 46, 48, 52

[6] T. Agolli, L. Pollock, and J. Clause, "Investigating decreasing energy usage in mobile apps via indistinguishable color changes," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2017, pp. 30–34. 234

[7] R. W. Ahmad, A. Gani, S. H. A. Hamid, F. Xia, and M. Shiraz, "A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues," *Journal of Network and Computer Applications*, vol. 58, pp. 42–59, 2015. 4, 6, 126

[8] Akamai, *HTTP/2 is the future of the Web, and it is already here!* `http://http2.akamai.com/demo/`, (last accessed: 2015-APR-22), Akamai. 137, 157

[9] T. Akidau, *The world beyond batch: Streaming 101*, `https://www.oreilly.com/people/09f01-tyler-akidau`, (last accessed: 2018-AUG-05). 220

[10] 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)*, Mar. 2014, pp. 1–4. 4, 32, 128, 201, 212, 234

[11] 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*, ser. NOLISP'11, Las Palmas de Gran Canaria, Spain, 2011, pp. 246–253, ISBN: 978-3-642-25019-4. 54

[12] Alexa, *The top 500 sites on the web*, `http://www.alexa.com/topsites`, (last accessed: 2015-APR-22), Alexa. 151

[13] 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*, ser. MSR '16, Austin, Texas, 2016, pp. 468–471, ISBN: 978-1-4503-4186-8. 92

[14] E. Alpaydin, "Combining multiple learners," in *Introduction to Machine Learning (Second Edition)*, MIT Press. 57

[15] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, Essen, Germany, 2012, pp. 258–261. 36, 82, 83

[16] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013. 36, 82, 84, 244

[17] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 367–381. 36, 84

[18] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, Cary, North Carolina, 2012, 59:1–59:11. 83, 244

[19] Android, *Android Debug Bridge*, `https://developer.android.com/studio/command-line/adb`, (last accessed: 2018-Jul-22). 218

[20] Android Open Source Project, *Andriod API for Logging*, `http://developer.android.com/reference/android/util/Log.html`, Last accessed 02/10/2015. 166, 167

[21] android-architecture, *Android architecture blueprints*, `https://github.com/googlesamples/android-architecture`, (last accessed: 2018-Aug-02). 213

[22] *Angulo*, `https://f-droid.org/en/packages/eu.domob.angulo/`, (last accessed: 2018-Jun-02). 224

[23] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 259–269, ISBN: 978-1-4503-2784-8. DOI: `10.1145/2594291.2594299`. [Online]. Available: `http://doi.acm.org/10.1145/2594291.2594299`. 118

[24] I. Ayala, M. Amor, L. Fuentes, and D. Muñoz, "An empirical study of power consumption of web-based communications in mobile phones," in *15th Intl Conf on Pervasive Intelligence and Computing*, Nov. 2017, pp. 861–866. 72

[25] I. Ayala, M. Amor, and L. Fuentes, "An energy efficiency study of web-based communication in android phones," *Scientific Programming*, vol. 2019, Jul. 2019. 72

[26] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA '13, Indianapolis, Indiana, USA, 2013, pp. 641–660. 83

[27] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '09, Chicago, Illinois, USA, Nov. 2009, pp. 280–293, ISBN: 978-1-60558-771-4. 140

[28] Banerjee, Abhijeet and Chong, Lee Kee and Chattopadhyay, Sudipta and Roychoudhury, Abhik, "Detecting Energy Bugs and Hotspots in Mobile Apps," in *FSE 2014*, Hong Kong, China, Nov. 2014, pp. 588–598, ISBN: 978-1-4503-3056-5. 4, 31, 49, 128, 136, 146,

[29] A. Banerjee and A. Roychoudhury, "Energy-aware Design Patterns for Mobile Application Development (Invited Talk)," in *Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile*, ser. DeMobile 2014, Hong Kong, China, Nov. 2014, pp. 15–16, ISBN: 978-1-4503-3225-5. 33

[30] L. Bao, D. Lo, X. Xia, X. Wang, and C. Tian, "How android app developers manage power consumption?: An empirical study by mining power management commits," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, Austin, Texas, 2016, pp. 37–48, ISBN: 978-1-4503-4186-8. 224

[31] L. A. Barroso, J. Dean, and U. Hölzle, “Web Search for a Planet: The Google Cluster Architecture,” *IEEE Micro*, vol. 23, no. 2, pp. 22–28, Mar. 2003, ISSN: 0272-1732. 3, 136

[32] D. F. Bauer, “Constructing confidence sets using rank statistics,” *Journal of the American Statistical Association*, vol. 67, no. 339, pp. 687–690, 1972. 102

[33] M. Belshe, *A 2x Faster Web*, `http://googleresearch.blogspot.ca/2009/11/2x-faster-web.html`, (last accessed: 2015-APR-22), Google. 139

[34] M. Belshe and R. Peon, *SPDY Protocol*, `http://tools.ietf.org/html/draft-ietf-httpbis-http2-00`, (last accessed: 2015-APR-22), HTTPbis Working Group. 139

[35] Y. Benjamini and Y. Hochberg, “Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995. 174, 177, 184

[36] Bohdan Samusko, *Model-view-presenter: Our choice of architecture for your android app*, `https://steelkiwi.com/blog/model-view-presenter-our-choice-of-android-app/`, (last accessed: 2018-Aug-02). 213

[37] P. Boonstoppel, C. Cadar, and D. Engler, “Rwset: Attacking path explosion in constraint-based test generation,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 351–366. 36, 84

[38] H. Brotherton, “Data center energy efficiency,” English, PhD thesis, Purdue University, May 2014, ISBN: 9781321439083. 136

[39] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, “Investigating the energy impact of android smells,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2017, pp. 115–126. 234

[40] A. Carroll and G. Heiser, “An Analysis of Power Consumption in a Smartphone,” in *Proceedings of the USENIXATC’10*, 2010. 23, 26, 47, 81, 126, 199,

[41] W. Cherif, Y. Fablet, E. Nassor, J. Taquet, and Y. Fujimori, “DASH Fast Start Using HTTP/2,” in *Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, ser. NOSSDAV ’15, Portland, Oregon, USA, Mar. 2015, pp. 25–30, ISBN: 978-1-4503-3352-8. 157

[42] W. Choi, G. Necula, and K. Sen, “Guided gui testing of android apps with minimal restart and approximate learning,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA ’13, Indianapolis, Indiana, USA, 2013, pp. 623–640. 83

[43] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15, Washington, DC, USA, 2015, pp. 429–440, ISBN: 978-1-5090-0025-8. 82–84, 118

[44] 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*, ser. MSR '16, Austin, Texas, 2016, pp. 508–511. 75, 200, 232

[45] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, "Greenscaler: Training software energy models with automatic test generation," *Empirical Software Engineering*, Jul. 2018. 1, 4, 7, 14, 24, 25, 34, 36

[46] S. Chowdhury, S. Di Nardo, A. Hindle, and Z. M. (Jiang, "An exploratory study on assessing the energy impact of logging on android applications," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1422–1456, Jun. 2018. 8, 41, 80, 108, 128, 159,

[47] S. Chowdhury, S. Gil, S. Romansky, and A. Hindle, *Greenscaler-tools-and-data*, `https://github.com/shaifulcse/GreenScaler-Tools-and-Data`, 2017. 7, 71, 77, 113, 118, 121,

[48] S. Chowdhury and A. Hindle, "Greenoracle: Estimating software energy consumption with energy measurement corpora," in *Proceedings of the 13th International Conference on Mining Software Repositories*, Austin, Texas, 2016, pp. 49–60. 6, 14, 15, 18, 40, 41, 75,

[49] 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, Dec. 2015. 13, 45, 48, 52, 53, 78, 10

[50] 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, Mar. 2016. 7, 13, 35, 46, 78, 128, 12

[51] *Chromeshell apks*, `http://commondatastorage.googleapis.com/chromium-browser-continuous/index.html?prefix=Android/`, (last accessed: 2015-May-22). 51

[52] *Chukwa - hadoop wiki*, `http://wiki.apache.org/hadoop/Chukwa`, Last accessed 04/18/2015. 163

[53] Cisco, *Cisco visual networking index: Forecast and methodology, 2016-2021*, `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html`, (last accessed: 201-Aug-06). 209

[54] "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014—2019," Cisco, Technical Report, Feb. 2015. 44, 136

[55] N. Cliff, *Ordinal methods for behavioral data analysis.* Psychology Press, 2014. 102

[56] *Colorpicker*, `https://f-droid.org/en/packages/com.enrico.sample/`, (last accessed: 2018-Jun-02). 224

[57] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, Mar. 2019. 34, 35, 160, 246

[58] Y. D'Elia, *Fgallery: A modern, minimalist javascript photo gallery*, `http://www.thregr.org/~wavexx/software/fgallery/`, (last accessed: 2015-APR-22), Wavexx. 143

[59] *Dalvikexplorer apks*, `https://code.google.com/archive/p/enh/downloads`, (last accessed: 2015-Aug-22). 51

[60] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *2015 USENIX Annual Technical Conference (USENIX ATC)*, 2015. 202

[61] 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*, ser. DAC '09, San Francisco, California, 2009, pp. 652–657, ISBN: 978-1-60558-497-3. 97, 109

[62] M. Dong and L. Zhong, "Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems," in *Proceedings of the MobiSys '11*, Jun. 2011, pp. 335–348, ISBN: 978-1-4503-0643-0. 24, 47, 126, 199, 212

[63] 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), eMarketer. 44

[64] ——, *Smartphone users worldwide will total 1.75 billion in 2014*, `http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536`, (last accessed: 2015-APR-22). 136

[65] Emma, *EMMA: a free Java code coverage tool*, `http://emma.sourceforge.net/`, (last accessed: 2016-JUL-22), 2006. 87

[66] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan, "Towards a SPDY'Ier mobile web?" In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13, Santa Barbara, California, USA, Dec. 2013, pp. 303–314, ISBN: 978-1-4503-2101-3. 156

[67] eva2000, *HTTP/2 - h2o vs OpenLiteSpeed vs Nginx SPDY/3.1*, `http://community.centminmod.com/threads/http-2-h2o-vs-openlitespeed-vs-nginx-spdy-3-1.2564/`, (last accessed: 2015-APR-22), Centmin Mod. 157

[68] *F-droid: Free and open source android app repository*, `https://f-droid.org/`, (last accessed: 2018-May-22). 50, 224

[69] A. Fedotyev, *The Real Cost of Logging*, `http://blog.appdynamics.com/net/the-real-cost-of-logging/`, Last accessed 04/18/2015, Apr. 2014. 163

[70] *Firefox apks*, `https://ftp.mozilla.org/pub/mobile/nightly/`, (last accessed: 2015-May-22). 51

[71] J. Flinn and M. Satyanarayanan, "PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications," in *WMCSA '99*, New Orleans, Louisiana, USA, Feb. 1999, pp. 2–10, ISBN: 0-7695-0025-0. 24, 47, 126, 199, 212

[72] M. Fowler, "Gui architectures. 2006," *URL http://www. martinfowler. com/eaaDev/uiArchs. html*, 2007. 212

[73] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, 2014. 202

[74] 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*, ser. MoVid '13, Oslo, Norway, Feb. 2013, pp. 7–12, ISBN: 978-1-4503-1893-8. 28, 49, 128, 156, 234

[75] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented Programming Systems and Applications (OOPSLA)*, 2007. 174, 183, 196

[76] Go Language, *Go + HTTP/2*, `http://http2.golang.org/gophertiles/`, (last accessed: 2015-APR-22), Golang. 157

[77] Í. Goiri, M. E. Haque, K. Le, R. Beauchea, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "Matching Renewable Energy Supply and Demand in Green Datacenters," *Ad Hoc Networks*, vol. 25, no. 0, pp. 520–534, Feb. 2015. 136

[78] A. Goldberg, R. Buff, and A. Schmitt, *A comparison of http and https performance*, `http://www.cs.nyu.edu/artg/research/comparison/comparison.html`, (last accessed: 2015-APR-22), New York University. 149

[79] I. Goodfellow, Y. Bengio, and A. Courville, *Regularization for Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`. 121

[80] Google, *Make the Web Faster*, `https://developers.google.com/speed/?csw=1`, (last accessed: 2015-APR-22), Google. 139

[81] Google Play, *Sensor readout*, `https://play.google.com/store/apps/details?id=de.onyxbits.sensorreadout`, (last accessed: 2018-Jun-02). 224

[82] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India, 2014, pp. 72–82, ISBN: 978-1-4503-2756-5. 36, 82, 84

[83] A. Grabner, *Top Performance Mistakes when moving from Test to Production: Excessive Logging*, `http://tinyurl.com/q9odfsm`, Last accessed 04/18/2015, Aug. 2012. 163

[84] "Greenbundle: An empirical study on the energy impact of bundled processing," in *Proc. of the International Conference on Software Engineering (ICSE-2019)*, 2019, p. 12. 8, 34, 205

[85] *Greenbundle: Replication and extension*, `https://github.com/shaifulcse/GreenBundle-Data-Code`. 211, 217

[86] *Greenoracle dataset*, `https://github.com/shaifulcse/GreenOracle-Data`, (created on: 2016-Jan-29). 53

[87] I. H. W. Group, *Http/2*, `https://http2.github.io/`, (last accessed: 2015-APR-22), IETF HTTP Working Group. 137

[88] T. O. Group, *Application Response Measurement - ARM*, `https://collaboration.opengroup.org/tech/management/arm/`, visited 2014-11-24. 166

[89] 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*, ser. GREENS '16, Austin, Texas, 2016, pp. 1–7. 129

[90] 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*, ser. HPCA '02, 2002, pp. 141–150. 23, 47, 81, 126, 199, 212

[91] E. Guttman, C. Perkins, J. Veizades, and M. Day, "Service location protocol, version 2," RFC Editor, RFC 2608, Jun. 1999, `http://www.rfc-editor.org/rfc/rfc2608.txt`. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc2608.txt`. 138

[92] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption Using Program Analysis," in *ICSE '13*, 2013, pp. 92–101. 2, 4, 13, 22, 44, 47, 60, 7

[93] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14, Bretton Woods, New Hampshire, USA, 2014, pp. 204–217. 83

[94] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2015, pp. 1–12. 82

[95] 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*, ser. ICSE '16, Austin, Texas, 2016, pp. 225–236, ISBN: 978-1-4503-3900-1. 2, 4, 14, 15, 30, 75, 79, 8

[96] 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. 56, 57, 98

[97] ——, "Model assessment and selection," in *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics. 102

[98] ——, "Support vector machines and flexible discriminants," in *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics. 57

[99] 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), 2015. 75

[100] A. Hindle, "Green mining: A methodology of relating software change and configuration to power consumption," *Empirical Software Engineering*, vol. 20, no. 2, pp. 374–409, Apr. 2015. 187, 199

[101] A. Hindle, "Green Mining: Investigating Power Consumption Across Versions," in *ICSE '12*, Jun. 2012, pp. 1301–1304, ISBN: 978-1-4673-1067-3. 1, 14, 44, 45, 63, 79, 136

[102] 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*, Hyderabad, India, May 2014, pp. 12–21, ISBN: 978-1-4503-2863-0. 14, 15, 51, 52, 79, 80, 13

[103] M. Hlavac, *Stargazer: Well-formatted regression and summary statistics tables*, R package version 5.2, Harvard University, Cambridge, USA, 2015. [Online]. Available: `http://CRAN.R-project.org/package=stargazer`. 176

[104] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods.* John Wiley & Sons, 2013. 102, 218

[105] W. G. Hopkins, *A new view of statistics*, [Online accessed 2017-01-15] `http://www.sportsci.org/resource/stats/index.html`, 2016. 190, 191

[106] M. A. Hoque, M. Siekkinen, and J. K. Nurminen, "Using Crowd-sourced Viewing Statistics to Save Energy in Wireless Video Streaming," in *Proceedings of the 19th Annual International Conference on Mobile Computing*, ser. MobiCom '13, Miami, Florida, USA, Oct. 2013, pp. 377–388. 28

[107] D. Horn, "Electrons and electricity," in *Basic Electronics Theory (Fourth Edition)*, McGraw-Hill. 12

[108] "How does docker affect energy consumption? evaluating workloads in and out of docker containers," *Journal of Software Systems*, pp. 1–14, May 2018. 134

[109] C.-w. Hsu, C.-c. Chang, and C.-j. Lin, *A practical guide to support vector classification*, 2010. 100

[110] HTTP Archive, *HTTP Trends*, `http://httparchive.org/trends.php?s=All&minlabel=Oct+22+2010&maxlabel=Apr+15+2015`, (last accessed: 2015-APR-22), HTTP Archive. 138

[111] *HTTP/2 Implementations*, `https://github.com/http2/http2-spec/wiki/Implementations`, (last accessed: 2015-APR-22), Github. 142

[112] HttpWatch, *A Simple Performance Comparison of HTTPS, SPDY and HTTP/2*, `http://blog.httpwatch.com/2015/01/16/a-simple-performance-comparison-of-https-spdy-and-http2/`, (last accessed: 2015-APR-22), HttpWatch. 157

[113] *Ina219 zerø-drift, bidirectional current/power monitor with i2c interface*, `http://www.ti.com/lit/ds/symlink/ina219.pdf`, Texas Instruments, Dallas, USA, Dec. 2015. 124, 169, 196

[114] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India, 2014, pp. 435–445, ISBN: 978-1-4503-2756-5. 84–86

[115] R. Jabbarvand, J.-W. Lin, and S. Malek, "Search-based energy testing of android," in *Proceedings of the 41st International Conference on Software Engineering*, Montreal, Quebec, Canada, 2019, pp. 1119–1130. 37

[116] R. Jabbarvand and S. Malek, "Mudroid: An energy-aware mutation testing framework for android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany, 2017, pp. 208–219. 234

[117] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, Saarbr&#252;cken, Germany, 2016, pp. 425–436. 35–37, 129, 186, 234

[118] E. Jagroep, J. Broekman, J. M. E. M. van der Werf, S. Brinkkemper, P. Lago, L. Blom, and R. van Vliet, “Awakening awareness on energy consumption in software engineering,” in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Society Track*, Buenos Aires, Argentina, 2017, pp. 76–85. 41

[119] A. Jay, *Do unused logging statements affect performance in Android apps?* Stack Overflow `http://tinyurl.com/ncf2nl9`, Last accessed 04/18/2015. 163

[120] ——, *Log.d and impact on performance*, Stack Overflow `http://stackoverflow.com/questions/3773252/log-d-and-impact-on-performance`, Last accessed 04/18/2015. 163

[121] M. Jiang, X. Luo, T. Miu, S. Hu, and W. Rao, “Are http/2 servers ready yet?” In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2017, pp. 1661–1671. 134

[122] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, “An automated approach for abstracting execution logs to execution events,” *Journal Software Maintenance Evolution*, vol. 20, pp. 249–267, 4 Jul. 2008. 189, 202

[123] Z. Jiang, A. Hassan, G. Hamann, and P. Flora, “Automatic Identification of Load Testing Problems,” in *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*, 2008. 5, 163, 202

[124] Jiffy, *Linux Man Page*, `http://man7.org/linux/man-pages/man7/time.7.html`, (last accessed: 2016-Jan-10), 2016. 79

[125] T. Joachims, “Making large-scale support vector machine learning practical,” in *Advances in Kernel Methods*, B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds., 1999, pp. 169–184, ISBN: 0-262-19416-3. 57

[126] JoJo, *Does logging slow down a production Android app?* Stack Overflow `http://stackoverflow.com/questions/6445153/does-logging-slow-down-a-production-android-app`, Last accessed 04/18/2015. 163

[127] 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), 2016. 1, 75

[128] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan, “Logging library migrations: A case study for the apache software foundation projects,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 154–164. 5

[129] T. Kalibera and R. Jones, “Rigorous Benchmarking in Reasonable Time,” in *Proceedings of the 2013 International Symposium on Memory Management (ISMM)*, 2013. 174, 183, 196

[130] 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). 55, 124

[131] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, May 2015. 209

[132] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, Saarbr&#252;cken, Germany, 2016, pp. 165–176. 82, 244

[133] S. Kornik, *University of alberta research could prevent app updates from draining smartphone batteries*, `https://globalnews.ca/news/4643134/`, (last accessed: 2019-May-22). 72

[134] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith, "A dataset of open-source android applications," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, Florence, Italy, 2015, pp. 522–525. 224

[135] J. Krystynak, *How to serve billions of web requests per day – without breaking a sweat*, `https://www.infoworld.com/article/2868513/database/how-to-serve-billion-web-requests-per-day.html`, (last accessed: 2018-AUG-22). 220

[136] Laerd, *Spearman's Rank-Order Correlation*, `https://statistics.laerd.com/statistical-guides/spearmans-rank-order-correlation-statistical-guide.php`, (last accessed: 2018-May-11). 219

[137] 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*, ser. DeMobile 2015, Bergamo, Italy, 2015, pp. 25–28, ISBN: 978-1-4503-3815-8. 2, 75, 137

[138] 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 ICSME*, Victoria, BC, Canada, Sep. 2014, pp. 121–130, ISBN: 978-1-4799-6146-7. 34, 117, 124, 129, 200

[139] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating Source Line Level Energy Information for Android Applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, 2013. 199, 200

[140] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond, "Integrated Energy-directed Test Suite Optimization," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, 2014. 199

[141] 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*, ser. ICSE '16, Austin, Texas, 2016, pp. 249–260, ISBN: 978-1-4503-3900-1. 4, 36, 89, 124, 129, 186,

[142] D. Li, A. H. Tran, and W. G. J. Halfond, "Making Web Applications More Energy Efficient for OLED Smartphones," in *ICSE 2014*, Hyderabad, India, Jun. 2014, pp. 527–538, ISBN: 978-1-4503-2756-5. 4, 26, 34, 49, 97, 113, 12

[143] Y. D. Lin, J. F. Rojas, E. T. H. Chu, and Y. C. Lai, "On the accuracy, efficiency, and reusability of automated test oracles for android devices," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 957–970, Oct. 2014, ISSN: 0098-5589. 83

[144] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, Hyderabad, India, 2014, pp. 2–11. 33, 34, 37, 117, 129, 191

[145] 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*, ser. ESEC/FSE 2015, Bergamo, Italy, 2015, pp. 143–154, ISBN: 978-1-4503-3675-8. 128

[146] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, Florence, Italy, 2015, pp. 111–122. 83

[147] Linux man-pages project, *Intro linux man page*, `http://linux.die.net/man/2/intro`, 2016. 53, 99

[148] Y. Liu, Y. Ma, X. Liu, and G. Huang, "Can http/2 really help web performance on smartphones?" In *2016 IEEE International Conference on Services Computing (SCC)*, Jun. 2016, pp. 219–226. 134

[149] 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. 4, 32, 128, 201, 212, 234

[150] *Logcat*, `http://developer.android.com/tools/help/logcat.html`, Last accessed 02/10/2015. 167

[151] *logstash - open source log management*, `http://logstash.net/`, Last accessed 04/18/2015. 163

[152] C. V. Lopes, *Exercises in programming style*. Chapman and Hall/CRC, 2016. 212

[153] Q. Luo, D. Poshyvanyk, and M. Grechanik, “Mining performance regression inducing code changes in evolving software,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, Austin, Texas, 2016, pp. 25–36. 243

[154] Y. Lyu, D. Li, and W. G. J. Halfond, “Remove rats from your code: Automated optimization of resource inefficient database writes for mobile applications,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands, 2018, pp. 310–321. 16, 34, 36, 205, 214

[155] A. MacCormack, J. Rusnak, and C. Y. Baldwin, “Exploring the structure of complex software designs: An empirical study of open source and proprietary code,” *Management Science*, vol. 52, no. 7, pp. 1015–1030, Jul. 2006. 232

[156] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: An input generation system for android apps,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia, 2013, pp. 224–234. 36, 82–84, 244

[157] R. Mahmood, N. Mirzaei, and S. Malek, “Evodroid: Segmented evolutionary testing of android apps,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Hong Kong, China, 2014, pp. 599–609. 36, 82, 83, 246

[158] H. Malik, P. Zhao, and M. Godfrey, “Going green: An exploratory analysis of energy-related questions,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR ’15, Florence, Italy, 2015, pp. 418–421. 75, 233

[159] 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*, ser. ICSE ’16, Austin, Texas, 2016, pp. 237–248, ISBN: 978-1-4503-3900-1. 2–4, 21, 75, 78, 79, 118,

[160] 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*, ser. ICSE 2014, Hyderabad, India, 2014, pp. 503–514. 31, 129, 234

[161] K. Mao, M. Harman, and Y. Jia, “Sapienz: Multi-objective automated testing for android applications,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, Saarbr&#252;cken, Germany: ACM, 2016, pp. 94–105, ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931054. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931054. 36, 82–84, 244, 246

[162] T. Mastelic, A. Oleksiak, H. Claussen, I. Brandic, J.-M. Pierson, and A. V. Vasilakos, "Cloud Computing: Survey on Energy Efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, 33:1–33:36, Dec. 2014, ISSN: 0360-0300. 3

[163] J. McDonald, *Kruskal–wallis test: Handbook of biological statistics*, `http://www.biostathandbook.com/kruskalwallis.html`, (last accessed: 2018-AUG-07). 233

[164] A. McIntosh, S. Hassan, and A. Hindle, "What can android mobile app developers do about the energy consumption of machine learning?" *Empirical Software Engineering*, Jun. 2018. 217

[165] 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. 62

[166] A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03, Victoria, B.C., Canada, Nov. 2003, pp. 260–269, ISBN: 0-7695-2027-8. 31

[167] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using java pathfinder," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, Nov. 2012, ISSN: 0163-5948. 83

[168] *Microsoft*, `https://docs.microsoft.com/en-us/windows/desktop/uxguide/progress-bars`, (last accessed: 2018-Jun-02). 221, 226

[169] 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*, ser. HotCloud'10, Boston, MA, USA, Jun. 2010. 27, 49, 109, 128, 129, 15

[170] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining api popularity," in *Testing – Practice and Research Techniques*, 2010, pp. 173–180. 121

[171] A. Miranskyy, Z. Al-zanbouri, D. Godwin, and B. Bener, "Database engines: Evolution of greenness," *Journal of Software: Evolution and Process*, vol. 30, no. 4, 2018. 109

[172] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, Austin, Texas, 2016, pp. 559–570. 36, 82, 83, 244

[173] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: A new metric for architectural maintenance complexity," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, Austin, Texas, 2016, pp. 499–510. 211, 231, 232

[174] P. Mohan, S. Nath, and O. Riva, "Prefetching Mobile Ads: Can Advertising Systems Afford It?" In *EuroSys '13*, Prague, Czech Republic, Apr. 2013, pp. 267–280, ISBN: 978-1-4503-1994-2. 29

[175] Monkey, *UI/Application Exerciser Monkey*, `https://developer.android.com/studio/test/monkey.html`, (last accessed: 2016-May-11). 82–84

[176] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2016, pp. 33–44. 83

[177] I. Moura, G. Pinto, F. Ebert, and F. Castor, "Mining Energy-Aware Commits," in *MSR 2015*, Florence, Italy, May 2015, ISBN: 978-1-4503-2863-0. 44

[178] Mozilla, *Index of /pub/mozilla.org/mobile/nightly*, `http://ftp.mozilla.org/pub/mozilla.org/mobile/nightly/`, (last accessed: 2015-APR-22), Mozilla. 142

[179] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009. 196

[180] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the ISSTA '09*, Chicago, IL, USA, 2009, pp. 57–68, ISBN: 978-1-60558-338-9. 36, 84

[181] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, "The cost of the "s" in https," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14, Sydney, Australia: ACM, 2014, pp. 133–140, ISBN: 978-1-4503-3279-8. DOI: `10.1145/2674005.2674991`. [Online]. Available: `http://doi.acm.org/10.1145/2674005.2674991`. 149

[182] J. Neill, *Why use effect sizes instead of significance testing in program evaluation*, `http://www.wilderdom.com/research/effectsizes.html`, 2008. 178

[183] M. Nottingham, *HTTP/2 Approved*, `http://www.ietf.org/blog/2015/02/http2-approved/`, (last accessed: 2015-APR-22), HTTPbis Working Group. 140

[184] D. D. Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. D. Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?" In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2017, pp. 103–114. 4, 25, 97, 110, 127, 212,

[185] K. Oku, T. Kubo, D. Duarte, N. Desaulniers, M. Hörsken, M. Nagano, J. Marrison, and D. Maki, *H2o - an optimized http server with support for http/1.x and http/2*, `https://github.com/h2o/h2o`, (last accessed 2015-APR-22). 142, 143

[186] A. Oliner, A. Ganapathi, and W. Xu, "Advances and Challenges in Log Analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, Feb. 2012. 202

[187] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre, "A Survey on Techniques for Improving the Energy Efficiency of Large-scale Distributed Systems," *ACM Comput. Surv.*, vol. 46, no. 4, 47:1–47:31, Mar. 2014. 136

[188] M. Othman and S. Hailes, "Power Conservation Strategy for Mobile Computers Using Load Sharing," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 2, no. 1, pp. 44–51, Jan. 1998. 27, 49, 128, 156

[189] J. Padhye and H. F. Nielsen, "A comparison of SPDY and HTTP performance," Tech. Rep. MSR-TR-2012-102, Jul. 2012. [Online]. Available: `http://research.microsoft.com/apps/pubs/default.aspx?id=170059`. 144, 148, 157

[190] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *21st IEEE International Requirements Engineering Conference (RE)*, Jul. 2013, pp. 125–134. 228

[191] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, May 2016. 2, 20, 21, 44, 75, 78, 200

[192] A. Pathak, C. Hu, and M. Zhang, "Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof," in *EuroSys '12*, Bern, Switzerland, Apr. 2012, pp. 29–42, ISBN: 978-1-4503-1223-3. 4, 16, 17, 29, 34, 35, 49,

[193] 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*, ser. HotNets-X, Cambridge, Massachusetts, 2011, 5:1–5:6. 32, 128, 201

[194] 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*, Salzburg, Austria, Apr. 2011, pp. 153–168, ISBN: 978-1-4503-0634-8. 4, 16, 18, 24, 35, 45–49,

[195] 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*, Jun. 2015, pp. 883–887. 4, 128, 201, 212, 234

[196] R. Pereira, M. Couto, J. Saraiva, J. Cunha, and J. 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*, ser. GREENS '16, Austin, Texas, 2016, pp. 15–21. 4, 30, 31, 75, 129, 212, 2

[197] G. Pinto and F. Castor, "Energy efficiency: A new concern for application software developers," *Commun. ACM*, vol. 60, no. 12, pp. 68–75, Nov. 2017. 41

[198] G. Pinto, F. Castor, and Y. D. Liu, "Mining Questions About Software Energy Consumption," in *MSR 2014*, 2014, pp. 22–31, ISBN: 978-1-4503-2863-0. 2, 21, 44, 66, 75, 113, 15

[199] 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). 44, 209

[200] M. Potel, "Mvp: Model-view-presenter the taligent programming model for c++ and java," *Taligent Inc*, p. 20, 1996. 212

[201] M. Pradel, M. Huggler, and T. R. Gross, "Performance regression testing of concurrent classes," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA, 2014, pp. 13–25. 82

[202] *Processes and threads overview*, `https://developer.android.com/guide/components/processes-and-threads`, (last accessed: 2018-Jul-22). 215

[203] Pulkit Sethi, *Xamarin application architecture*, `https://blog.kloud.com.au/2018/01/17/xamarin-application-architecture/`, (last accessed: 2018-Aug-02). 213

[204] K. Rasmussen, A. Wilson, and A. Hindle, "Green Mining: Energy Consumption of Advertisement Blocking Methods," in *GREENS 2014*, Hyderabad, India, Jun. 2014, pp. 38–45, ISBN: 978-1-4503-2844-9. 14, 29, 49, 52, 79, 129, 1

[205] *Replication Package Android Logcat Energy Study*, `https://archive.org/details/ReplicationPackageAndroidLogcatEnergyStudy`, Last accessed 05/24/2017. 165, 204

[206] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?" In *Annual Meeting of the Florida Association of Institutional Research*, Feb. 2006. 177

[207] S. Romansky, N. C. Borle, S. Chowdhury, A. Hindle, and R. Greiner, "Deep green: Modelling time-series of software energy consumption," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai, China, Sep. 2017, pp. 273–283. 41, 108, 217

[208] S. Romansky and A. Hindle, "On Improving Green Mining For Energy-Aware Software Analysis," *Proceedings of the 2014 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2014. 168, 187, 199

[209] R. Saborido, G. Beltrame, F. Khomh, E. Alba, and G. Antoniol, "Optimizing user experience in choosing android applications," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, Mar. 2016, pp. 438–448. DOI: `10.1109/SANER.2016.64`. 26, 128

[210] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, Jun. 2012, pp. 55–61. 35, 212, 235

[211] C. Sahin, L. Pollock, and J. Clause, "From benchmarks to real apps: Exploring the energy impacts of performance-directed changes," *Journal of Systems and Software*, vol. 117, pp. 307–316, 2016. 2, 5, 205, 209, 211, 224

[212] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014. 30, 129, 200, 234

[213] C. Sahin, P. Tornquist, R. McKenna, Z. Pearson, and J. Clause, "How Does Code Obfuscation Impact Energy Usage?" In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014. 30, 129, 200, 234

[214] sandalone, *Android debugging -¿ battery drains*, Stack Overflow `http://stackoverflow.com/questions/4958543/android-debugging-battery-drains`, Last accessed 04/18/2015. 163

[215] R. Sasnauskas and J. Regehr, "Intent fuzzer: Crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, ser. WODA+PERTEA 2014, San Jose, CA, USA, 2014, pp. 1–5. 83

[216] H. de Saxce, I. Oprescu, and Y. Chen, "Is http/2 really faster than http/1.1?" In *Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on*, Apr. 2015, pp. 293–299. 153

[217] Scitools.com, *Understand*, `https://scitools.com/`, (last accessed: 2018-Aug-18). 231

[218] *Sensor readout*, `https://f-droid.org/en/packages/de.onyxbits.sensorreadout/`, (last accessed: 2018-Jun-02). 224, 227

[219] C. Seo, S. Malek, and N. Medvidovic, "Component-level energy consumption estimation for distributed java-based software systems," in *Lecture Notes in Computer Science*, ser. Lecture Notes in Computer Science, vol. 5282, Springer Berlin Heidelberg, 2008, pp. 97–113, ISBN: 978-3-540-87890-2. 48, 81, 126, 199

[220] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna, "From retrospect to prospect: Assessing modularity and stability from software architecture," in *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*, Sep. 2009, pp. 269–272. 232

[221] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 201–211. 82

[222] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An Exploratory Study of the Evolution of Communicated Information About the Execution of Large Software Systems," in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, 2011. 201

[223] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting Developers of Big Data Analytics Applications when Deploying on Hadoop Clouds," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013. 202

[224] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, vol. 20, no. 1, pp. 1–27, Feb. 2015. 202

[225] M. Shepperd, M. Cartwright, and G. Kadoda, "On building prediction systems for software engineers," *Empirical Software Engineering*, vol. 5, no. 3, pp. 175–182, Nov. 2000. 123

[226] J. Shore, *How many data sources in your apps? let me count the apis.* `https://searchcloudapplications.techtarget.com/blog/Head-in-the-Clouds-SaaS-PaaS-and-Cloud-Strategy/How-many-data-sources-Let-me-count-the-APIs`, (last accessed: 2018-AUG-22). 220

[227] 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*, New York, NY, USA, Dec. 2009, pp. 168–178, ISBN: 978-1-60558-798-1. 23, 47, 81, 126, 199, 212

[228] H. Si and A. Hada, *Introduction of HTTP/2 and Performance Comparison of HTTP/1 and HTTP/2*, `http://www.ixiacom.com/about-us/news-events/corporate-blog/introduction-http2-and-performance-comparison-http1-and-http`, (last accessed: 2015-APR-22), Ixia. 157

[229] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: `%5Curl%7Bhttp://research.google.com/archive/papers/dapper-2010-1.pdf%7D`. 163

[230] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th. Wiley Publishing, 2008. 18

[231] R. J. G. Simons and A. Pras, "The hidden energy cost of web advertising," Technical Report TR-CTIT-10-24, Jun. 2010. 29

[232] W. Song, X. Qian, and J. Huang, "Ehbdroid: Beyond gui testing for android applications," in *ASE 2017*, Urbana-Champaign, IL, USA, 2017, pp. 27–37. 117

[233] *SPDY Performance on Mobile Networks*, `http://developers.google.com/speed/articles/spdy-for-mobile`, (last accessed: 2015-APR-22), Google. 139, 156

[234] *SPDY: An experimental protocol for a faster web*, `http://www.chromium.org/spdy/spdy-whitepaper`, (last accessed: 2015-APR-22), Google. 156

[235] *Splunk*, `http://www.splunk.com/`, Last accessed 04/18/2015. 163

[236] StackOverflow, *Why are synchronize expensive in Java?* `http://stackoverflow.com/questions/1671089/why-are-synchronize-expensive-in-java`, (last accessed: 2016-Jul-22), 2009. 66, 113

[237] Statista, *Number of smartphone users worldwide from 2014 to 2020 (in billions)*, `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`, (last accessed: 201-Aug-06). 209

[238] D. Stenberg, "HTTP/2 Explained," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 120–128, Jul. 2014, ISSN: 0146-4833. 137, 139, 140, 143

[239] strace, *Linux Man Page*, https://linux.die.net/man/1/strace. 19

[240] *Summary of Sarbanes-Oxley Act of 2002*, `http://www.soxlaw.com/`. 166

[241] *THE /proc FILES YSTEM*, `https://www.kernel.org/doc/Documentation/filesystems/proc.txt`, (last accessed: 2014-May-22). 41, 53

[242] TheOldFox, *What is FireFox Nightly ?* `https://support.mozilla.org/en-US/questions/970739`, (last accessed: 2015-APR-22), Mozilla. 142

[243] H. C. Thode, *Testing for normality.* CRC press, 2002, vol. 164. 102

[244] B. Thompson, *A suggested revision to the forthcoming 5th edition of the apa publication manual*, `http://people.cehd.tamu.edu/~bthompson/apaeffec.htm`, 2000. 178

[245] R. Trestian, A.-N. Moldovan, O. Ormond, and G. Muntean, "Energy consumption analysis of video streaming to Android mobile devices," in *Proceedings of the 2012 IEEE Network Operations and Management Symposium*, ser. NOMS'12, Maui, HI, USA, Apr. 2012, pp. 444–452. 28, 156

[246] G. Uddin and F. Khomh, "Automatic summarization of api reviews," in *ASE 2017*, Urbana-Champaign, IL, USA, 2017, pp. 159–170. 121

[247] V. Vapnik, *The nature of statistical learning theory*. Springer, 2000. 57, 98, 99

[248] *VLC apks*, `http://nightlies.videolan.org/build/android-armv7/backup/`, (last accessed: 2015-Aug-22). 51

[249] X. Wang, X. Li, and W. Wen, "Wlcleaner: Reducing energy waste caused by wakelock bugs at runtime," in *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, Aug. 2014, pp. 429–434. 32, 179, 201, 212, 234

[250] 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*, ser. NSDI'14, Seattle, WA, USA, Apr. 2014, pp. 387–399. 4, 128, 137, 143, 151, 15

[251] C. Wilke, S. Richly, S. Götz, C. Piechnick, and U. Aßmann, "Energy consumption and efficiency in mobile applications: A user feedback study," in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, Aug. 2013, pp. 134–141. 209, 210

[252] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Proceedings of the Future of Software Engineering (FOSE) track, International Conference on Software Engineering (ICSE)*, 2007. 166

[253] V. Woollaston, *Customers really want better battery life*, `http://www.dailymail.co.uk/sciencetech/article-2715860/`, (last accessed: 2018-Jul-22), 2014. 1, 3, 75, 136, 209, 210

[254] World Wide Web Consortium (W3C), *The Original HTTP as defined in 1991*, `http://www.w3.org/Protocols/HTTP/AsImplemented.html`, (last accessed: 2015-APR-22), World Wide Web Consortium (W3C). 138

[255] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, 29:1–29:10. 224

[256] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online System Problem Detection by Mining Patterns of Console Logs," in *Proceedings of the Ninth IEEE International Conference on Data Mining (ICDM)*, 2009. 5, 163, 202

[257] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting Large-scale System Problems by Mining Console Logs," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009. 202

[258] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra, "Robust test automation using contextual clues," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA, 2014, pp. 304–314. 82

[259] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'13, Rome, Italy, 2013, pp. 250–265. 83

[260] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing &#38; Multimedia*, ser. MoMM '13, Vienna, Austria, 2013, 68:68–68:74. 83

[261] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: Error Diagnosis by Connecting Clues from Run-time Logs," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010. 5, 163, 202

[262] D. Yuan, S. Park, and Y. Zhou, "Characterising Logging Practices in Open-Source Software," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012. 201

[263] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011. 5, 163, 202

[264] Y. Zeng, J. Chen, W. Shang, and T.-H. (Chen, "Studying the characteristics of logging practices in mobile apps: A case study on f-droid," *Empirical Software Engineering*, Feb. 2019. 160

[265] C. Zhang, "The impact of user choice on energy consumption," *MSc. thesis, University of Alberta*, 2013. 125

[266] C. Zhang, A. Hindle, and D. German, "The impact of user choice on energy consumption," *Software, IEEE*, vol. 31, no. 3, pp. 69–75, May 2014. 62

[267] C. Zhang and A. Hindle, “A green miner’s dataset: Mining the impact of software change on energy consumption,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, Hyderabad, India, 2014, pp. 400–403. 82, 244

[268] J. Zhang, A. Musa, and W. Le, “A Comparison of Energy Bugs for Smartphone Platforms,” in *MOBS’13*, San Francisco, CA, USA, May 2013, pp. 25–30. 31

[269] 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, ISBN: 978-1-60558-905-3. 24, 47, 123, 126, 199, 21

[270] 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). 55