University of Alberta

THE IMPACT OF USER CHOICE AND SOFTWARE CHANGE ON ENERGY CONSUMPTION

by

Chenlei Zhang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2013
Edmonton, Alberta

Dedicated to my parents Zhixian Zhang and Dilan Chen, my aunt Huixian Zhang, my sister Beibei Zhang, and my dear one You Fu, for their love and support.

Abstract

Hardware and software engineers are instrumental in developing energy efficient mobile systems. Unfortunately the last mile of energy efficiency comes from the choices and requirements of the end-user. Imagine an end-user who has no power-outlet access and must remain productive on her laptop battery life. How does this user maximize the laptop's battery life, yet remain productive? What does the user have to give up to keep on working? In the first half of this thesis, we highlight the peril that users face and the ultimate responsibility users have for the battery life and energy consumption of their mobile devices; using multiple scenarios we show that executing a task can consume more or less energy depending on the requirements and software choices of users. We investigate multiple scenarios demonstrating that applications can consume energy differently for the same task thus illustrating the tradeoffs that end-users can make for the sake of energy consumption.

Furthermore, as the builders and more frequently the maintainers of applications, software developers are responsible for updating and shipping energy efficient applications for end-users. Yet, the impact of software changes on energy consumption is still a mystery. Thus in the second half of this thesis we relate software changes to energy consumption by tracing the system calls that act as the interface between user applications and the OS kernel. We show the energy consumption evolution of multiple `gedit` versions under two test scenarios. We also present the potential of modeling software energy consumption via system call invocations.

Acknowledgements

I would like to thank my supervisor Dr. Abram Hindle for his great supervision and helpful advice. This thesis cannot be possible without his patient guidance. I appreciate that he has led me into the wonderful research area, Mining Software Repositories, and enriched me with all the skills I need for conducting those interesting projects. My gratitude also goes to Dr. Daniel M. German for his valuable suggestions in the first part of my thesis. In addition, I would like to thank the financial support from NSERC Discovery Grant during my master study.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Despite of the prominent popularity of various mobile computing platforms, such as smartphones, tablets, and laptops, the battery life of these devices is always one of the most important factors that affect the user experience. With a large number of applications developed for each mobile platform, as shown in Table 1.1, end-users have a wide range of applications to choose from. However, energy efficiency of these applications is unknown to end-users. If an application drains out a mobile device's battery very quickly, the end-user will not get service from her device until the next charging, which also gives mobile application developers the pressure of releasing energy efficient applications.

Table 1.1: The number of available applications in various application stores by 2013.

Application Store	The Number of Available Applications
Apple App Store [2]	900,000
Mac App Store [28]	15,300
Windows Phone Store [27]	160,000
Google Play [36]	1,000,000
Ubuntu Software Center [34]	45,000
BlackBerry World [18]	250,000

1.1 User Choice and Software Energy Consumption

Energy is a major concern in society. Energy efficiency is the concerned effort to reduce the amount of energy required to create and use products and services. Classically the energy consumption and battery life of mobile devices (from laptops to phones and other types of PDAs) has relied on the computer, electrical and software engineers who built and configured the system. The hardware components dictate how much power can be used and the software determines when and how the components are used.

In the realm of software, energy efficiency has been primarily the concern of operating systems (OS), in particular those used in mobile devices (including laptops). There is little research in the area of energy efficient end-user applications, and it has primarily focused on recommendations on how to increase energy efficiency; for example, Intel recommends the use of compilers and libraries designed to use power-saving features of their CPUs, and to use algorithms that minimize data movement and use cached memory efficiently [17].

From the point of view of energy efficiency, software can be seen as a service. Therefore, we define software efficiency of a software application as the amount of energy that it requires per "unit of service" it provides. In general, the purpose of measuring the energy efficiency of a device or service is to use it as a comparison: if we can normalize two devices to provide the same amount of service (say, two refrigerators of the same size, or televisions of the same size, two laptops with comparable displays playing the same movie), their energy efficiency ratings can be used to identify the device or service that is more energy efficient.

Within the software realm, a "unit of service" is a malleable unit that must be defined from the point of view of the user. This can be: typing an average page, listening to 1 hr of music, watching a 2 hrs movie, checking and downloading email every 5 minutes, running n transactions per minute, etc. A unit of service can be seen as a metric of quality-of-service, and in many cases, it does not make any difference if it can be executed any faster: the user will be satisfied as long as she can run the services she wants with the desired quality of service. For example,

when listening to audio it does not make sense to play it faster; the user might not want to be interrupted more frequently than every five minutes by new emails, and therefore, there is no need to check for them more frequently than that; or typing a page of text every 15 minutes because she cannot type any faster.

The user might have different applications to satisfy these needs. In general, users will be confronted with many software systems to choose from. For example, there is a very large number of music players, text editors and mail clients on the market. All of them are likely capable of delivering the quality of service that users require. The decision on which to use lies on more subjective requirements (both functional and non-functional) such as usability, features, cost, etc.

One non-functional requirement that is rarely considered is energy efficiency. Given two software applications that can provide the same service, at the expected quality, is one more energy efficient than the other? This is particularly important if the goal is to maximize battery life. For example, assume you are on a transoceanic flight without access to electricity, and you want to play music in your laptop as you type a letter. You can choose between multiple applications to do it. How much does the choice of application impact the battery life? How much battery life would you lose if you decide to play music with the most efficient player versus not playing any music at all?

In the first half of this thesis, we demonstrate that because users have a choice in terms of what software they use their choice plays an important role in the energy consumption and battery life of their mobile devices (and by extension in the energy requirements of society).

1.2 Software Change and Energy Consumption

Software developers are making changes to applications throughout the software development process. With bug fixed and new features added by the changes, possibly software energy consumption behaviour is changing too. A concrete body of research has been applied to build power models for applications on mobile devices. Zhang et al. [37] have implemented a power model for Android smartphones,

PowerTutor, which is based on the power modeling of each hardware component. Dong et al. [7] have applied a different approach to modeling application power consumption for Linux-based mobile systems based on the system statistics of each hardware component. Pathak et al. [26] have generated power consumption finite state machines for components on smartphones and developed energy profiler to estimate energy consumption of applications. The most recent study was done by Hao et al. [12]. They have created a power model, *eLens*, to model Android applications based on Java instructions. These studies could help both end-users and developers to do energy accounting for applications. However, none of them has investigated the impact of software change on application energy consumption. Hindle [15] has made the first step toward revealing the relationship between software change and power consumption. He has measured power consumption of multiple software versions and found potential correlation between software metrics and power consumption.

In this thesis we have observed that software power consumption is changing based on different test cases, and therefore, as a static feature of software changes, software metrics are not enough for uncovering the impact of software change on power consumption.

1.3 Thesis Overview

1. ***The Impact of User Choice on Energy Consumption (Chapter 3)***
   We highlight the peril that users face and the ultimate responsibility users have for the battery-life and energy consumption of their mobile devices; using multiple scenarios we show that executing a task can consume more or less energy depending on the requirements and software choices of users. We investigate multiple scenarios demonstrating that applications can consume energy differently for the same task thus illustrating the tradeoffs that end-users can make for the sake of energy consumption.

2. ***Mining Multiple Versions of Software on Energy Consumption (Chapter 4)***
   We provide a method and a case study to reveal the correlation between soft-

ware change and power consumption. To be specific, we first investigate the software power consumption evolution over multiple versions. Then, we trace the system calls invoked by a list of software versions. At last, we leverage system call invocations to model software power consumption over versions. The results show different power consumption behaviours in terms of multiple software versions and different test cases. Also, system call invocations have the potential of modeling software power consumption based on multiple versions.

1.4 Thesis Contributions

In this thesis, we investigate the impact of user choice and software change on power consumption based on empirical case studies. The contributions are as follows:

1. We define the concept of energy-efficiency for software applications.
2. We create the benchmarks for energy efficiency of software systems.
3. We perform an experiment to benchmark the energy efficiency of several software applications that shows there can be significant variation and differences in the energy efficiency of applications.
4. We demonstrate that users can have a positive impact in reducing the energy consumption of the computers they use.
5. We propose a methodology of relating software change to power consumption by tracing system calls.
6. We perform an experiment to contrast the different power consumption behaviours in terms of multiple software versions and different test cases.
7. We present the potential correlation between system call invocations and software power consumption.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 discusses the related research work relevant to this thesis from three categories, user-centric evaluation, power modeling and mining software repositories, and briefly introduces the relationship between energy consumption and software. Chapter 3 presents the first half of our study, the impact of user choice on energy consumption. In Chapter 4 we demonstrate our work about mining multiple versions of software on energy consumption, which forms the second half of this thesis. We conclude this thesis and discuss the future work in Chapter 5.

Chapter 2

Related Work

In this chapter, we discuss the related work to this thesis and explain the relationship between energy consumption and software. The related work is organized by three categories. First, we present papers related to user-centric evaluation about software energy consumption. Second, we review approaches that focus on power modeling and management for hardware and software systems. Third, we survey research work about combining software power performance with mining software repositories techniques. At last we discuss the measures of energy and energy consumption, and the choice of metrics used in this thesis.

2.1 User-Centric Evaluation

In this section, we discuss the studies of software energy consumption in the literature from the end-user perspective.

A technology website with Linux orientation run by Michael Larable, Phoronix, demonstrated interests of power management from the users' perspective [20]. The energy consumption of several Linux distributions have been benchmarked for two test scenarios, idle and under load. Both of the two test scenarios were running off AC adapter without battery and just battery. The results show the differences of energy consumption among all the Ubuntu distributions that have been tested.

Chetty *et al.* [5] investigated how people use power management strategies on their home computers by their field study of 20 households. They conducted the research in two phrase. First, all the computer usage patterns, such as which ap-

plications were used, duration and times of use and power state of the computer, were recorded by the logging software. Second, when finishing collecting usage patterns, each one whose actions were logged during the experiment did a survey regarding their power management behaviours. The results show that people tend to leave their computers on in most of the time when the machines are not active and the energy saving for a household when using current power management strategies matters to few people. The authors argued that researchers should study home power management strategies that have few changes on people's home computing habits.

Amsel *et al.* [1] implemented a tool, Green Tracker, to help software users understand the power consumption of software systems running on their computers. Green Tracker is able to collect CPU usage and makes use of average CPU usage of software to estimate the energy efficiency. By automating several typical usage patterns in Green Tracker, such as browsing websites, writing articles and playing audio, the authors could get the results about CPU usage for each application in the usage patterns. Thus in each usage pattern, software users would have a basic understanding about the power consumption of applications. Although the power model in this study only included CPU performance as the parameter, it made the first step to draw the users' attention to power consumption of software.

McLachlan *et al.* [22] sought to characterize the power consumption of various interaction techniques for finishing a single task. To be specific, the authors used 8 different interactions, such as auto scroll, mouse wheel and mouse drag to navigate a PDF document. Power consumption was collected for all the interactions separately and the results reveal that the power consumption for fulfilling a single task using different interactions varies significantly. Furthermore, the authors argued that the power consumption difference of the interaction technique depends on the number of screen updates involved. This discovery points to a power consumption optimizing direction for both users and software developers to use and implement the power efficient interaction techniques.

Compared to the previous work in terms of user-centric evaluation, we take a much broader and task oriented approach in this thesis.

2.2 Power Modeling and Management

Power modeling is an important part in both hardware and software energy accounting techniques since power regression tests are time-consuming and difficult to make. With the aid of power consumption models, we can understand and optimize the power characteristics of hardware and software systems without complicated instrumentation using power meters. There are a large body of work focusing on power modeling and management for hardware components (CPU, disks, networks, etc), desktops, servers and recently mobile devices.

2.2.1 Power Modeling for Specific Components

A lot of researchers have been studying power consumption models for one specific component, such as hard disks [9], CPU [19], networks [21], and displays [6].

Hard Disks. Greenawalt [9] has built a statistical model to optimize the energy consumption of hard disks. In the paper a disk was modeled as having four modes of operation. The power consumption of a hard disk was determined based on the combination of these four modes. The tradeoff among power consumption, system performance, and system reliability (lifetime of the hard disks) were evaluated by controlling two parameters, the rate of accessing the hard disk and the length of timeout in the case studies. The model is simulated and it does not actually test against real hard disks.

CPU. Joseph *et al.* [19] introduced the first power consumption model for microprocessors. The authors have made use of two types of information from processors to build the linear regression model for power consumption. The first type is performance-relevant event counters and the second one is signal transition statistics. This model can provide runtime power estimation and most importantly a glimpse of component power consumption for microprocessors.

Networks. Lattanzi *et al.* [21] have developed an approach to modeling and estimating the runtime power consumption of wireless network interface cards (WNICs). The authors measured the current waveform of a WNIC and applied pattern matching algorithms to correlate a pattern of current waveform with the state in the

WNIC. Each state in the WNIC would be characterized to a profile regarding the current level and timing. A tool was implemented to analyze a measured current waveform of the WNIC using generated power profile of each state and estimate the runtime power consumption of the WNIC. The model was also adapted for various working conditions and transmitting workload in this study. This paper shows that power modeling is not restricted to CPU and slicing the application to different states would be helpful to model power consumption of applications. Balasubramanian *et al.* [3] measured and compared the power consumption characteristics for three network protocols (3G, GSM and WiFi) on mobile phones. By applying Nokia Energy Profiler [24] on Nokia N95 phones, power consumption of different states for each network protocol was recorded. The results reveal that, compared to WiFi, 3G and GSM have a significant tail energy overhead (energy spent in high-power state after the completion of the data transfer). Later on, the authors proposed the power models for the three network protocols based on the data collected from the measurement. They also developed a protocol, TailEnder, which applies delay-tolerance deadlines and prefething to optimize the energy consumption of network applications.

Displays. Dong *et al.* [6] have studied the power models for the display component, which is usually made of organic light-emitting diode (OLED) in mobile devices nowadays. Three power models for OLED were proposed in this paper and they are based on pixel-level, image-level and code-level correspondingly. Pixel-level power model for OLED was formulated by linear regression based on physical measurement. The other two power models were sampled from the pixel-level one. Hence, pixel-level power model for OLED has the best accuracy but lacks of efficiency. The results show that different color components in an OLED pixel have different power characteristics. The green component has the least power consumption, then followed by the red component. The blue component has the highest power consumption among them. Based on the different inputs as well as tradeoffs between accuracy and efficiency, the authors concluded that the OLED power models can be utilized by developers working on both hardware and application levels to estimate the power consumption of GUIs.

This thesis differs from these studies because our methodology focuses on software systems and takes power consumption of the whole operating system into account, instead of one specific component.

2.2.2 Power Modeling for the Full System

A more general research focus is based on building the power models for the whole system, such as an operating system and a Java virtual machine (JVM). We review recent attempts to modeling power consumption for the full system from three aspects, utilization-based power models, instruction-based power models, and system-call-based power models [26].

2.2.2.1 Utilization-Based Power Models

In the utilization-based approach, there are often two data loggers and two steps involved to build the power models for the full system. The two data loggers include a logger for collecting utilization of each hardware component, and a logger for measuring power consumption (such as using power meters). The first step is to collect utilization statistics and the corresponding power consumption when running a list of applications under a sample of scenarios. The second step is to apply regression analysis, usually linear regression, to train a model for software power consumption based on utilization data. Then software power consumption can be predicted by training the model with utilization statistics.

Flinn *et al.* [8] modeled the power consumption of each component in a software system by designing and implementing a tool called PowerScope. By combining the current measurement and each process in the system, PowerScope would generate the profile of energy usage of each component in an application. A case study was conducted to demonstrate the usage of this tool for optimizing the energy consumption of a video application. The effectiveness of reducing the power consumption of this application was supported by the effects of varying the amount of lossy compression, reducing the display size, switching the network interface, and powering down the disk. PowerScope is mostly based on the CPU profile and the power model is highly sensitive to the physical instrument.

Gurumurthi *et al.* [11] have built a complete system power simulator, which is called SoftWatt. It was implemented on the SimOS infrastructure and is able to model the CPU, memory, and disk of the targeted application. Once the CPU profile, memory profile, and disk statistics are collected by the SimOS, SoftWatt uses analytical power models to report the power statistics.

Shye *et al.* [31] studied the power consumption behaviour in an Android smartphone, HTC Dream, with data from real user activity. The authors implemented a logger application which is able to collect system performance metrics and user activity on an Android phone, and then send all the data back to a sever. The system performance metrics include statistics about CPU, Screen, Call, EDGE, WiFi, SD Card, DSP, and System. Based on these data collected from 20 users and power measurements by replaying user activities, they built a linear regression model which can predict the power consumption of HTC Dream using system performance metrics.

Carroll *et al.* [4] have also analyzed the power consumption of an Android smartphone, the Openmoko Neo Freerunner. In this study the researchers sliced the smartphone into specific components including CPU, memory, touchscreen, and so forth. Then power consumption of each component was collected by voltage and current measurement. Also six typical usage scenarios of a smartphone, which are audio playback, video playback, text messaging, voice calls, emailing, and web browsing, were tested on the smartphone to gather the distribution of power consumption in each component. Energy model of each usage scenario was provided and the authors analyzed the impact of these scenarios in different usage patterns on the battery life. The analysis in this research was very limited to the Android smartphone they chose and the measurement could be hardly applied to other phones due to limited documentation of hardware components.

Zhang *et al.* [37] have generated two online power models for Android smartphones, PowerTutor and PowerBooter. PowerTutor is based on the combination of power models for components in Android smartphones. The authors measured the power consumption of each hardware component (CPU, WiFi, Audio, LCD, GPS, and Cellular) under extreme usage on Android smartphone and built linear

regression models for each of them regarding of their power consumption. Since the power models generated by PowerTutor varies significantly between different modules of Android smartphones. Another more general power model, PowerBooter was created in this paper. PowerBooter is based on the discharge curve of the lithium-ion battery in a specific Android smartphone. PowerTutor has better accuracy compared with PowerBooter. Whereas they both have shortcomings. PowerTutor is specific to Android smartphone modules and PowerBooter needs the discharge curve of the battery on each Android smartphone.

Dong *et al.* [7] have implemented a self-constructive energy model for Linux-based mobile systems. The energy model, called Sesame, generates energy models for mobile systems without external power measurement. Sesame collects system statistics and applies the Advanced Configuration and Power Interface (ACPI) to gather the predictors for the power model. Energy readings are collected through the smart battery interface as the responses. The linear regression power models are generated based on the collected data.

Mittal *et al.* [23] have proposed and implemented a power model, WattsOn, for mobile device emulator on Windows Phone platform. It builds upon a set of power models that focus on individual specific component, including cellular network (3G), WiFi network, display, and CPU. Application developers could be aware of energy consumption of each component in order to make better implementing decisions for reducing energy consumption of applications. They also applied resource scaling to generalize WattsOn for real phones to overcome the measurement difference between the emulator and a phone.

As mentioned in [3] and [26], some of the components (NIC, 3G, and SD Card) have tail energy phenomenon. It means that these components can stay in high power state after the completion of an operation while the utilization at that moment is zero. Thus, utilization-based power models are unable to model the tail energy phenomenon. Compared with the utilization-based power models, our methodology is based on tracing system calls, which is able to overcome the problem caused by the tail energy phenomenon [26].

2.2.2.2 Instruction-Based Power Models

For applications running in a JVM, a list of research papers have taken a different approach to utilizing the Java bytecode instructions to build energy models for software systems.

Seo *et al.* [29], [30] have implemented an energy consumption model for Java-based software systems running on distributed devices. This power model consists of three components, computational energy cost, communication energy cost, and infrastructure energy overhead. Computational energy cost refers to the energy cost related to CPU, memory, and I/O operations. This component was modeled by physical benchmarking all the instructions in a JVM and native methods for a specific hardware device. Communication energy cost is simply the energy consumption for data transformation via network. In terms of the infrastructure energy cost, it means the energy consumption caused by the JVM's garbage collection and other OS routines during the executing of a Java application. This energy consumption model could make accurate estimates which fall within 5% of the actual energy cost for application. However, it is highly dependent on the hardware and JVM.

Hao *et al.* [13] made a step further and built an energy consumption model, *eCalc*, for Android applications' CPU energy usage at the level of the whole program and the method. The approach in *eCalc* is similar to [30]. They both need to measure the energy cost of each Java instruction for a specific software environment in order to build the model. The improvement made by *eCalc* over [30] is that *eClac* also traces additional information about the paths executed during the execution of the Android application. Thus, it is able to estimate the energy consumption of methods. An extension of *eCalc* implemented by Hao *et al.* [12], which is called *eLens*, combined program analysis with instruction-based power modeling. *eLens* is able to estimate energy consumption of more hardware components besides CPU and has fine-grained energy profiling level, ranging from the whole application to the code level.

Instruction-based power models are designed for software running in a JVM. While, in this thesis, we focus on investigating the power consumption of software running in Linux-based system.

2.2.2.3 System-Call-Based Power Model

System-call-based power model was proposed by Pathak *et al.* [26]. They have applied system call tracing to model the energy consumption of applications running on smartphones. First they studied the power behavior of some components in a smartphone which show that, 1) several components have tail power states (a component stays in high power state for a period of time after active I/O activities); 2) system calls that do not imply utilization can change power states; and 3) several components do not have quantitative utilization. These observations come to the conclusion that energy models based on correlating utilization with power consumption, which use linear regression models, are not accurate. Second, this study took three steps to build the energy model by tracing system calls. The first step was to model the power states and generate finite state machines (FSMs) of each system call for each component in a smartphone. The second step was to integrate all the FSMs of system calls to model a FSM for each individual component. In the final step the FSM model of the smartphone was developed based on the FSMs in second step. At last, based on the FSM of the certain smartphone, when tracing the system calls on the smartphone, they can identify the state that system is currently in and estimate the power consumption of an application. The authors have implemented FSMs for several Windows and Android smartphones. Their results show improved accuracy compared to an approach [31] based on the linear regression model. In the following paper [25], the authors have extended their work and implemented a fine-grained energy profiler for smartphone based on FSMs. This energy profiler, *eprof*, can work on both Android and Windows Mobile phones to estimate the energy consumption of smartphone apps.

Similar to system-call-based power model, we also trace system calls to correlate them with software power consumption. However, we only care about the number of system call invocations.

2.3 Mining Software Repositories

Mining software repositories (MSR) is a research field that applies statistical analysis, data mining, machine learning, and other automated techniques to rich data extracted from version control systems, issue tracking systems, mailing list archives and other software repositories in order to discover interesting and actionable information about software systems [14]. With the help of these historical information, we can acquire the knowledge of software development processes and characteristics, which can leads to the improvement of software decision process. MSR techniques have been successfully applied to a lot of questions such as predicting software defects, locating the locations of bugs in source code, and mining the social networks in developers. Only few papers have tried to leverage MSR techniques to understand software energy performance.

Gupta *et al.* [10] have studied the power consumption of Windows phone. They combined power traces and execution logs in Windows phone to build power models. Specifically, a power trace is the measurement of power on a phone over a test session by a power meter. Execution log is the sequence of active executable files and shared libraries, which are called modules in the paper, over a certain period of time. Based on the combined data set, they used linear regression models to model and predict the power consumption of application running on Windows phone. Besides, they applied data mining techniques such as decision trees to detect the energy patterns within the data set.

Hindle [16], [15] provided a detailed methodology, called *Green Mining*, to collect power consumption of applications over multiple versions on Linux-based systems. Based on the power measurement of applications and software metrics, the author studied the correlation between software change and power consumption over versions. Although the correlation between software metrics and power consumption is very low, *Green Mining* points to a promising research direction of combining power measurement with MSR techniques.

Our work is closely based on the methodology of *Green Mining* and extends the work to find the correlation between software change and power consumption via

system call tracing.

2.4 Energy Consumption and Software

This section explains the relationship between energy consumption and software. It includes the measures of energy and energy consumption, and the choice of the metrics used in this thesis.

2.4.1 Watt and Joule

The watt is a SI unit of power, named after the Scottish engineer James Watt. It measures the rate of energy conversion, transfer or consumption. Named after the English physicist James Prescott Joule, the joule is a SI unit of energy, often defined as the work required to produce one watt of power for one second. In the context of electrical engineering, energy consumption, or often referring to power consumption, is usually discussed in terms of energy required over time to operate an electrical appliance. Using a 1 000-watt electric heater for an hour, we need 1 000 watt-hours which equals to 3 600 000 joules. This amount of energy would also light up a 100-watt bulb for 10 hours. Giving the same task of lighting up a room for 3 hours, a 15-watt compact fluorescent lamp (CFL) only consumes 45 watt-hours while a 40-watt incandescent light bulb needs 120 watt-hours.

2.4.2 Energy Consumption and Software

For most of the cases, we regard the energy consumption of software systems as the energy consumed by the software to provide a certain amount of work or service over time. In terms of comparing energy consumption among software systems, we only report the mean watts of the software instead of watt-hours. Because the tests for software systems in the same category take the same amount of time to run.

We believe that it is more user-centric to argue about the energy consumption in terms of the resource that produces it (such as battery life – the time a full battery lasts before being exhausted) than in SI energy units. For this reason we use, throughout this thesis two metrics of energy consumption: battery life consumed

and the watt.

2.5 Chapter Summary

To sum up, we have discussed the work related to this thesis from three aspects. Namely, they are user-centric evaluation, power modeling and management, and mining software repositories. In the first half of this thesis, we investigate the impact of user choice on software energy consumption. It takes a more broad and task oriented approach compared to the previous work in terms of user-centric evaluation. In the second half of this thesis, we seek to uncover the correlation between software change and power consumption. This work is built upon *Green Mining* and extends it by the idea of system call tracing.

Chapter 3

The Impact of User Choice on Energy Consumption

A lot of researchers have been focusing on building power models for mobile devices to understand application power behaviours [11], [30], [37], [7], [25], [23], [12]. Only a limited number of research work has studied the software energy consumption from the end-users' perspective [5], [1], [22]. Chetty *et al.* [5] have investigated how people use power management strategies on their home computers. Amsel *et al.* [1] have implemented a tool, Green Tracker, which made the first step to draw the end-users' attention to software energy consumption. McLachlan *et al.* [22] characterized the power consumption of various interactions techniques for finishing a single task. In the context of getting a specific service or finishing a task, users usually have a large selection of software systems, but how users' choice of the application affects their devices' battery life has not been studied yet. In this chapter, we present our work about the impact of user choice on energy consumption, which is under review for the IEEE Software.

We propose a methodology to contrast the energy consumption of software systems that can provide the same quality of service. We first define the energy efficiency of software. Then we develop three use-case scenarios, text editing, email receiving, and music playing, to simulate real users performing a task using different software systems. By applying physical power meters, we benchmark the power consumption of software systems under each use-case scenarios. We also create a model for estimating the loss of a device's battery life using the power con-

sumption measurements. At last, we show that executing a task can consume more or less energy depending on the requirements and software choices of users, thus demonstrating the tradeoffs that users can make for the sake of energy consumption.

This chapter is organized as follows. Section 3.1 introduces the definition of energy efficiency of software. The use-case scenarios applied in this study are listed in Section 3.2. Our methodology and results are presented in Section 3.3 and Section 3.4 respectively. Section 3.5 discusses the causes of energy consumption and a rating system for application energy consumption. Section 3.6 talks about the threats to validity in this research and Section 3.7 summarizes this chapter.

3.1 Energy Efficiency of Software

To this day, power benchmarking has been primarily concerned with the energy consumption of different components of equipment (such as the CPU, wireless network adapter, hard drives, etc) or the entire system. Battery life is an example of the latter, when the goal is to try to understand how efficient the entire system is with a "typical load". The subject of the benchmark is the system not the applications used to test it.

The common user's view is that, if they want to save energy (usually to extend battery life) they should use energy efficient hardware or stop using peripherals such as USB ports, or wireless network adapters and stop running applications. As we mentioned in the introduction, Intel recommends that end-user applications should be concerned with their algorithms, CPU and memory use [17]. Therefore we can expect that different applications, would have different energy consumption (as they can have different speed performance).

Energy efficiency benchmarks should be user-centric. The user has certain tasks that the software is expected to accomplish. In some cases, the software, such as an email client, is expected to run "24/7". In others, the user chooses when the software runs and where it stops. To properly benchmark the energy consumption of an application is necessary to determine the typical amount of work expected from the application. This is highly dependent on the domain of the application. In some

cases this unit of work might be measured by unit of time (such as running an email client continuously), in others, simply in terms of units of work completed (such as compressing a file). Any of the benchmarked applications should be capable of completing this amount of work. We will call this the expected quality-of-service. We will refer to the energy required to complete the expected quality of service as the *energy efficiency of an application* (and measure it in watts per unit-of-work). The energy efficiency of an application is the difference between the amount of energy that a computer consumes while running an application to complete that unit of work compared to not running the application (all other things being equal).

Any computer consumes energy whether it is idle or not ("idle" here means running a computer or an application without any workload). In the same manner, running applications use energy whether they are being used or not. Benchmarks should also be created to measure the energy consumption of idle applications. We will name this the *ghost energy consumption* of an application. As its appliance's counterpart, the ghost energy consumption is particularly worrisome because it might be more expensive than stopping and restarting the application.

3.1.1 Measuring Energy Consumption

There are many ways to measure energy consumption. One can rely on monitors in the power hardware of a computer, such as the power supply or motherboard (often exposed by interfaces such ACPI), or one can measure the wall-power consumed by a computer with a physical power meter (the power that will be billed by the energy utility company).

In this paper we eschew ACPI, because power supply measurement is rare on desktop machines, instead we opt to measure the wall power with a *Watts Up? Pro* power meter plugged between the wall socket and the computer desktop (we did not measure the display consumption).

One problem with power measurement is that repeated runs of the same test often have slightly different results. These variations occur because modern computers, whether in your pocket or on your desktop, are complicated devices responsible for multiple tasks and services at the same time. Thus to get a good idea of

the actual energy consumption, tests need to be repeated. These repeated tests are difficult to run and limit end-user's ability to discern energy efficiency.

3.1.2 Battery Life Estimate

Mobile devices often provide the information about how much battery capacity or how much battery life is left for users via ACPI. We argue that users are more familiar with the metric of battery life than the watt. Also, users care about availability and the battery life of their mobile devices, thus we can use our energy consumption measurements in a desktop to model expected battery life performance on tested and other devices.

We created a model where we converted energy consumption measurements to the expected loss of a device's battery life. Assume that we have a laptop that has a ghost energy consumption (an idle consumption) of w_{idle} watts and a battery that can output whr_{total} watt-hours of energy. The equation to convert the energy consumption to battery life is shown below:

$$\Delta T = \frac{whr_{total}}{w_{idle}} - \frac{whr_{total}}{w_{idle} + \lambda \cdot (w_{app} - w_{testIdle})} \tag{3.1}$$

Where ΔT is the reduced battery life when applying an application with the energy consumption in w_{app} watts comparing to that of the laptop model being idle; $w_{testIdle}$ is the ghost energy consumption in watts when the testbed is idle; λ is used as the scaling multiplier to tune the difference between platforms (λ equals to 1 if the laptop model is the testbed itself).

Effectively this model estimates the naive difference in energy consumption for a given device and outputs the battery life lost in hours by running an application the entire time. This measurement could be viewed as how much time you get to use the application on a plane or without wall-power. Table 3.2 describes the model parameters for 3 devices. Section 3.4.5 describes the energy consumption of our test scenarios in terms of battery life.

3.2 Use-Case Scenarios

Our motivation is that of the stranded user who understands that they will have to operate on battery power until they run out. As a stranded user we want to be productive with our dwindling laptop battery life. In this section, we demonstrate the use-case scenarios chosen for this study and implementation details of these use-case scenarios are discussed in Section 3.3.3.

3.2.1 Text Editing

First and foremost we want to produce something, such as a report, so we shall assume at the very least we want to write. Typical scenarios for writing include using a text editor, such as `gedit`, or a word processor, such as Write (which is included in LibreOffice and OpenOffice). Cloud-based word processors such as Google Docs are growing in popularity due to their ease of access, price, and availability. Each of these text editing products offer a different range of features and usability.

Our use-case scenario will be to type in text at the rate of an amateur typist entering pre-written text and then to save the document.

3.2.2 Receiving Email

Of course we must stay connected to the world, thus we expect while we are stranded we will still communicate via email. A typical email scenario would be checking the inbox for new mails. A user might use a local application such as Outlook or Thunderbird, or they might use a webmail solution such as Google Mail.

Our use-case scenario will be an email client left open and idle, constantly checking for mail as mail slowly arrives. This scenario represents what mobile devices often do: idle and wait. If an application uses a lot of energy in order to do this it would be detrimental to the end-user.

3.2.3 Playing Music

Finally, we must stay entertained on our power-socketless desert island of productivity: we want to play music from our music collection. Applications such as Amarok or Rhythmbox help users both organize collections of music and play them, while specialized software such as `mpg123` focuses solely on playing music.

Our use-case scenario will include playing a music from the hard-drive using one of these applications in the foreground.

> Thus we are stranded, at a location like a cafe, without power outlets, relying solely on battery power and attempting to write a document, read email, and listen to music one task at a time. Given these tasks, *how do our choices in terms of applications affect how long we can work on battery power?*

3.3 Methodology

In this section we present the methodology for measuring and comparing the energy consumption among applications with equivalent functionality. The general process is derived from the previous work on *Green Mining* [15] and is as follows:

1. Choose a software product.
2. Decide on the level of instrumentation.
3. Develop a use-case scenario and simulate users completing a task.
4. Build up the test bed to measure the energy consumption.
5. Run the tests and analyze results.

3.3.1 Choosing Software Products

In this study, we chose to develop tests for text editing applications, email clients, and music players. The text editing applications include the default text editor `gedit` on GNOME desktop environment in Ubuntu, the open source word processor LibreOffice Writer, which we will simply refer to as LibreOffice from now on,

Table 3.1: Applications tested

Test	Application	Version
Text Editing	`gedit`	3.4.1
	LibreOffice Writer	3.5.7.2
	Google Docs	December 2012
Email Receiving	Mozilla Thunderbird	16.0.2
	Gmail on Mozilla Firefox	December 2012 on 16.0.2
Music Playing	`mpg123`	1.12.1
	Banshee	2.4.1
	Rhythmbox	2.96

and the web-based office suite Google Docs provided by Google. In terms of the email clients, Mozilla Thunderbird, which is a free and open source email client developed by the Mozilla Foundation, as well as the free webmail service Gmail which is provided by Google are chosen to be tested. When testing Google Docs and Gmail, we made use of the Mozilla Firefox as the web browser. Three music players have been tested and they are `mpg123`, Banshee and Rhythmbox. `mpg123` is a command-line MPEG audio player. Banshee is a media player under GNOME desktop and Rhythmbox is also a music player under GNOME desktop. The tested versions of all the related applications are shown in Table 3.1.

3.3.2 Deciding on the Level of Instrumentation

The device we used to measure the energy consumption of the testbed is an AC power monitor, *Watts Up? Pro* [15]. This meter can continuously monitor and collect power measurement with an accuracy of $\pm 3\%$. This hardware can monitor real-time electricity usage and collect a variety of data, including power consumption in watts, and transmit this result over a USB-serial connection.

3.3.3 Developing Use-Case Test Cases

In this study, we sought to imitate real world users using these applications and developed three scenarios to test the energy consumption for each application in

each scenario.

3.3.3.1 Idling on the Testbed

To properly measure the energy consumption of the application on the machine where the tests were run, we had to measure its idling energy consumption, i.e. the energy it consumes when no application is explicitly run. We left the Ubuntu Unity 2D desktop running idle for a period of 5 minutes and measured its energy consumption.

During these tests and all the subsequent tests described below a GNOME terminal running the GreenLogger was left running. GreenLogger does not produce any terminal output while measuring power.

3.3.3.2 Text Editing

For text editing applications, the testing scenario is to simulate a user creating a new document and then typing text into it and finally saving the document. We built a X11::GUITest UI driver to simulate the mouse actions and typing actions that we pre-recorded based on me typing in the preamble of the GNU General Public License (GPL) [1]. The test took almost 6 minutes to type about 560 words of the preamble in the GNU GPL.

To be specific, for `gedit` and LibrefOffice, the procedure is 1) start the application, which opens a new document; 2) type the GNU GPL Preamble; 3) save the file; and, 4) close the application. For Google Docs, 1) start Firefox; 2) go to to the Google Docs web page; 3) start a new document, 4) type the document; 5) save it; and 6) close Firefox.

3.3.3.3 Receiving Email

Our scenario to test email clients was meant to simulate a user idly receiving emails using either Thunderbird or Gmail in the foreground without user interaction. In order to implement this, a separate test computer sent plain text emails, 1 per minute, to a single Gmail account. Each email client was instructed to monitor the receiving

[1] GNU GPL, `http://www.gnu.org/copyleft/gpl.html`

email account. Thunderbird connected to Gmail using the IMAP protocol. Usually emails would appear approximately 5 seconds after they were sent. Gmail was run within Firefox and communicated directly to Gmail HTTP servers.

To test Thunderbird, 1) we started Thunderbird; 2) and monitored Thunderbird for 10 minutes watching the 10 emails appear; and 3) we closed Thunderbird.

To test Gmail, 1) Firefox was started with the Gmail cookie already set; 2) the client would be instructed to type in the Gmail URL to go to the Gmail web page; 3) monitor Firefox and Gmail for 10 minutes as it received the 10 emails; and 4) close Firefox.

3.3.3.4 Playing Music

Our third scenario was to listen to music. In order to test this we needed to test music players playing a three minute long song [2].

The `mpg123` player is a command-line based player, that we tested within a GNOME Terminal. It was started with the song as a command-line parameter and would play the song as soon as it started; it terminate once the song finished playing.

Banshee and Rhythmbox have GUIs, making their testing more complicated and requiring a GUI driver. They also maintain a database of the music of the users. For this reason, before we tested each application, we added the test song to their databases (it was the only song in them). Both Banshee and Rhythmbox were started by clicking on their respective icons on the Ubuntu 12.04's Unity panel. Once each application was opened we clicked the play icon which played the song, as it was the only song in the library. Once the song was finished playing, our GUI driver clicked the close icon and shutdown the music player.

3.3.3.5 Idling Applications

In order to understand the difference between ghost energy consumption of the idle applications and compare them to the energy consumption of the testing scenarios above, we tested the energy consumption of all the applications without taking any workload but staying in the background. To be specific, we started each of

[2]The song, "Flute Solo for the Iron-blooded Loyalists", was graciously made available for these experiments by its author, You Fu.

the text editing, email, and music applications (except for `mpg123` because it is command-line based and it is not intended to run idle), and immediately iconized them, without doing any work (except for the mail clients, who continued to check for new email, but did not receive any).

3.3.4 Configuring the Testbed

Our tests are meant to test the idea that different applications consume different amounts of energy. We expect that in other computers and operating systems this difference is still observable. Hence, due to the availability of testbed when this study was conducted, we implemented our tests on a desktop machine running 32-bit Ubuntu 12.04. It has an AMD Athlon XP2800+ processor with 1-Gbyte RAM. To minimize the noise when measuring the energy consumption of tests, we turned off any services and automatic updates performed by the operating system. We also disabled the screen saver and left the screen on during the tests. Headphones were plugged in for the music tests.

We created a test-user to run our scenario tests called greenmining and this user would run the default Ubuntu 12.04 Desktop, Unity 2D.

The desktop testing machine was plugged into a *Watts Up? Pro* power meter, which was instructed to continuously log its energy consumption as RMS, with a resolution of 1 measurement-per-second. The data is recorded by GreenLogger, which is our application that records *Watts Up? Pro* readings.

3.3.5 Running the Tests and Analyzing Results

If we studied this exact scenario, e.g., running applications from each category at the same time, we would need to consider all combinations of applications and factors that could affect the energy consumption among the systems. Thus, each application is run through its associated scenarios individually so that we can avoid the power consumption overhead caused by interactions among different applications as well as services provided by the OS to run multiple applications at the same time, such as scheduling. The energy consumption of its test machine is measured by GreenLogger.

We rebooted the testbed between different application tests. The first test after reboot was discarded in order to ensure uniformity of the disk-cache for subsequent tests. Thus our tests are run with a hot disk-cache. We ran each test 40 times. This number was chosen before hand in order to ensure that even in the presence of skew we could attempt to determine normality or differences between measurements.

Our tests were divided into three types: text editing, email receiving, and music playing. For each test we chose at two or three applications to test. Each test was performed 40 times, while the energy consumption of the computer was recorded.

3.4 Results

In this section, we present the results and investigate the energy consumption of all the tested applications. Note that according to the t-test that all of the comparisons in this section are statistically ($p < 10^{-10}$) significant even after correction for multiple hypotheses.

3.4.1 Idling on the Testbed and Applications

These tests could help us benchmark the testbed's idling energy consumption and applications' ghost energy consumption. Idling on the computer used 95.3 watts on average and we set it to be the system baseline, which is zero in Figure 3.1, 3.3 and 3.5 (the distributions of each test are statistically different from any other, except for the idle and busy measurement of Gmail, which is a statistical tie). As shown in our results, the ghost energy consumption (while idle) of text editing applications, `gedit`, LibreOffice, and Google Docs are 0.2, 0.3, and 1.0 watts correspondingly. The ghost energy consumption of music playing applications, Banshee, and Rhythmbox are 0.8 and 1.1 watts.

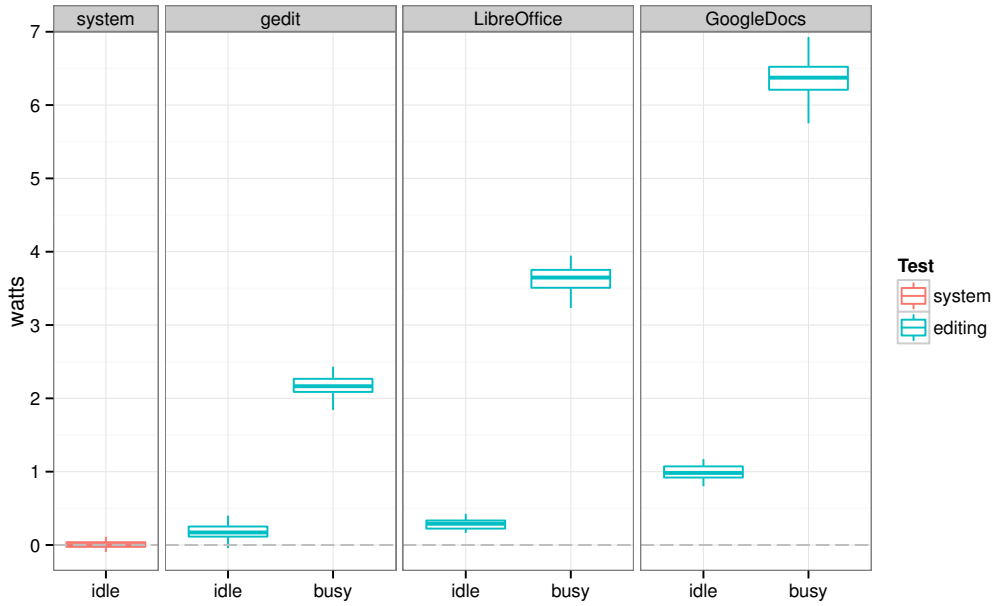

Figure 3.1: Distributions of the mean watts consumed per test: idling on testbed and text editing tests. 40 tests each, 280 tests total.

3.4.2 Text Editing

In this part, we discuss the energy consumption of three applications, `gedit`, LibreOffice, and Google Docs executing our text processing scenario.

`gedit` has limited functionality compared to LibreOffice and Google Docs which both focus more on word processing than text editing. Thus few layout features and formatting attributes are included in `gedit`. Since `gedit` is a lightweight text editor, it is unsurprising that `gedit` has the lowest energy consumption of the three. As shown in Figure 3.1, the average watts in the `gedit` tests is around 2.2 watts higher than the baseline, lower than both LibreOffice and Google Docs.

LibreOffice provides more features than `gedit`. It includes some layout features and formatting attributes like centering and making bold titles. LibreOffice also has automatic spell checking that can automatically highlight and correct misspelled words based on a dictionary. LibreOffice had 1.4 watts higher energy consumption than `gedit` did, as shown in Figure 3.1.

Google Docs is similar to LibreOffice in terms of its typesetting and layout features. Google Docs includes spell checking too. Unlike `gedit` and LibreOffice,

Google Docs automatically synchronizes and saves text that is typed. This auto-saving feature causes Google Docs to synchronize with Google's servers frequently. Thus, as we can observe from Figure 3.1, the average consumption of Google Docs is 6.4, 4.2, and 2.8 watts higher than that of the system baseline, `gedit`, and LibreOffice respectively.

Figure 3.2 shows the density plots of the energy consumption in the three text editing tests. `gedit`, LibreOffice, and Google Docs all have a start-up peak, which are heavy in terms of disk I/O, memory, and CPU use, and several centers. the centers in `gedit` and LibreOffice are similar except that `gedit` is more shrunk. Google Docs differs from the other two applications as its density plot depicts inconsistent energy consumption.

The text editor `gedit` consumes, in average, 1.4 watts less than LibreOffice, and 4.2 less than Google Docs. This is in part due to the extra features that the latter provide compared to the former, such as automatic spell checking and WYSIWYG rendering, Google Docs' use of cloud synchronization has impacted its energy efficiency as this is functionality not shared with `gedit` or LibreOffice.

3.4.3 Receiving Email

During receiving email scenario tests, Gmail used the HTTP protocol and updated the new emails very quickly. Whereas Thunderbird used IMAP protocol to retrieve the email and new emails appeared about 5 seconds later. From Figure 3.3, the energy consumption of Thunderbird and Gmail are 0.3 and 0.6 watts higher than the baseline on average, respectively, with 0.3 watts difference.

Density of the power measurements for receiving email scenario is shown in Figure 3.4. They both have relatively stable energy consumption and also the start-up peaks which are clearly noticeable.

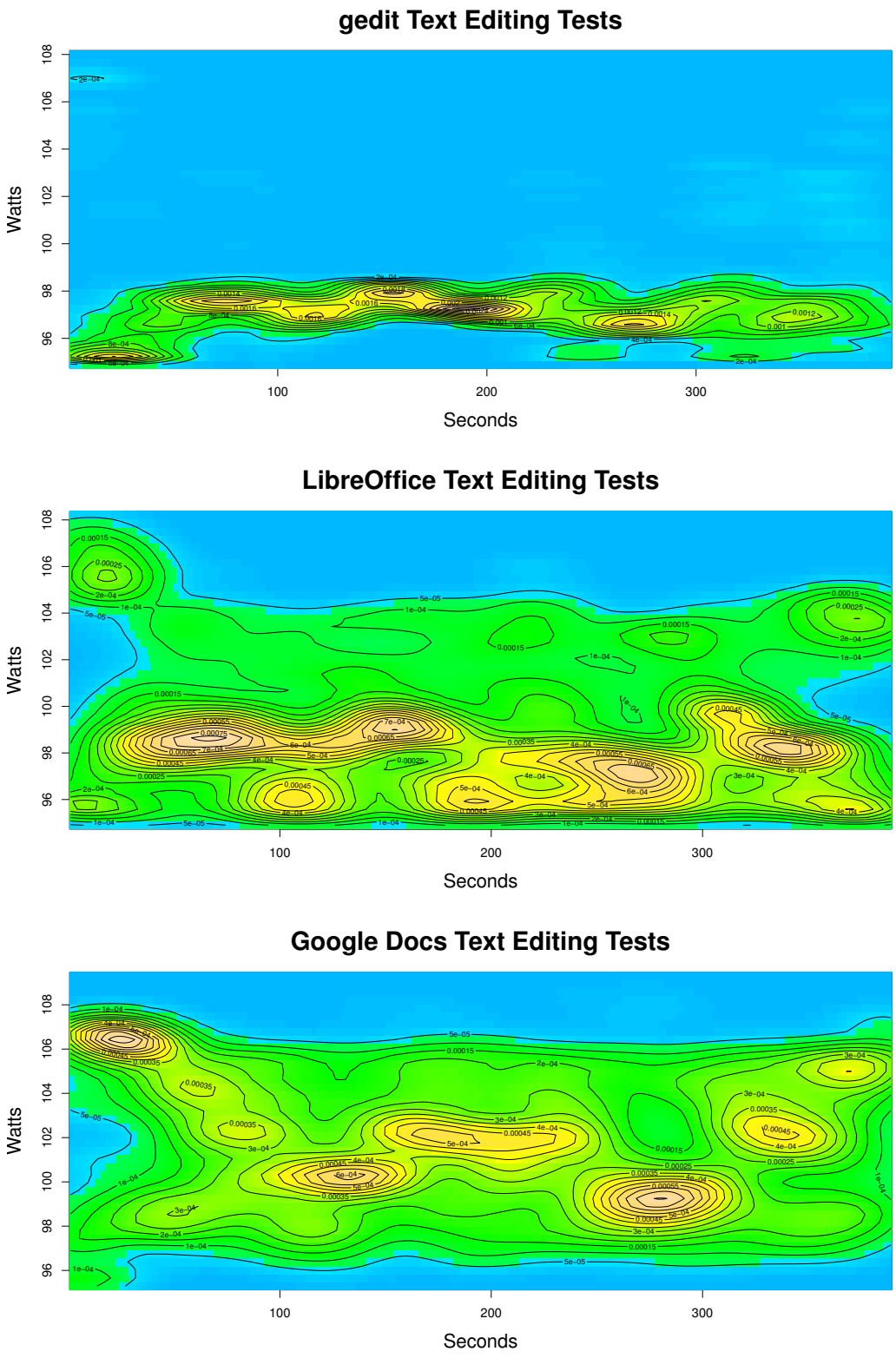

Figure 3.2: Density of the power measurements from `gedit` tests, LibreOffice tests, and Google Docs tests. The density indicates how often a measurement was taken at a certain watt at a certain second. This plot is effectively an overlay trace of how 40 tests ran in terms of watts.

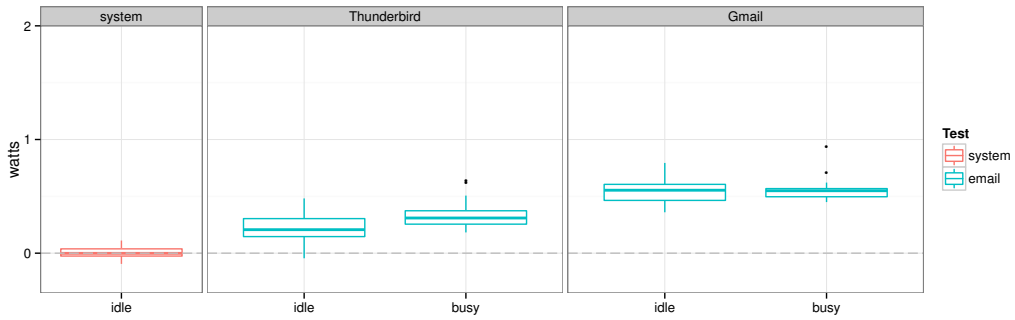

Figure 3.3: Distributions of the mean watts consumed per test: idling on testbed and email receiving tests. 40 tests each, 200 tests total.

Using different protocols to retrieve emails, Gmail and Thunderbird didn't behave the same in terms of energy consumption. Gmail had 0.3 watts higher energy usage than that of Thunderbird on average.

3.4.4 Playing Music

`mpg123` is a command-line music player, whereas Banshee and Rhythmbox both have graphical user interfaces. We used the default settings such as volume and EQ in these three application when playing the song. Based on Figure 3.5, on average, `mpg123` has the lowest energy consumption, 0.4 watts above the system baseline, compared to Banshee, 2.6 watts and Rhythmbox, 2.2 watts higher than the baseline.

As observed from the density plots in Figure 3.6, the energy consumption of the three music players seem to be flat during the music playing session. At the beginning of the tests, Banshee and Rhythmbox have start-up peaks while `mpg123` does not.

From the playing music tests, it is clear that graphical user interfaces and the music library management features would affect the energy consumption. Banshee using Mono, a .NET VM to run, perhaps that overhead caused it to use more than Rhythmbox.

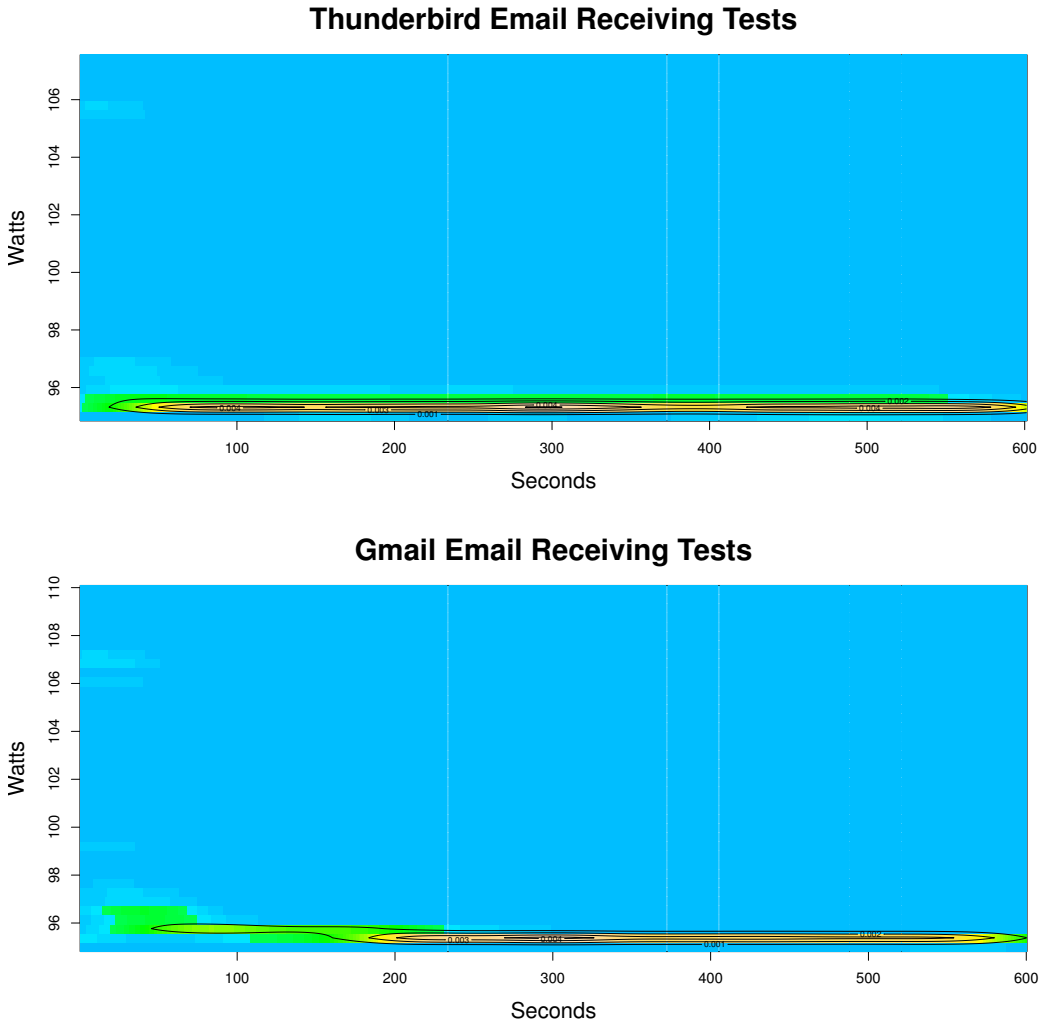

Figure 3.4: Density of the power measurements from email tests of Thunderbird and Gmail.

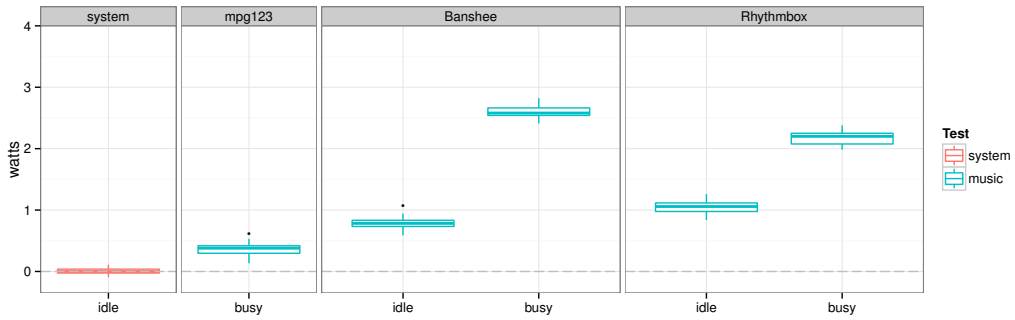

Figure 3.5: Distributions of the mean watts consumed per test: idling on testbed and playing music tests. 40 tests each, 240 tests total.

Table 3.2: battery life model parameters

Name	w_{idle} (watts)	Battery Life (hours)	whr_{total} (watt-hrs)
Big	95.3	3	285.9
X31	19.5	3.6	71

The non-GUI music player is approximately five to six times more energy efficient than the ones with a GUI. It is also remarkable that ghost energy consumption of the GUI players is 0.8 and 1.1 watts.

3.4.5 Application Energy Efficiency under Battery Life Models

In order to argue for the cost-effectiveness in choice of applications, we have built two laptop-based battery models, called Big and X31, to estimate the battery life. Big is a model of the desktop PC we used if it had a battery supply that could last 3 hours while idle. X31 model is based on our another testbed, the Lenovo X31 laptop. The parameters of the two models are listed in Table 3.2.

Figure 3.7 shows the distributions of the reduced mean battery life estimated by Big laptop model based on the desktop power measurements for all the applications. We can clearly observe the different impacts of applications on the battery life.

We can also estimate the battery life for running the applications on other battery life models using the power measurements from the desktop testbed. We implemented the identical tests on another testbed, the Lenovo ThinkPad X31, so

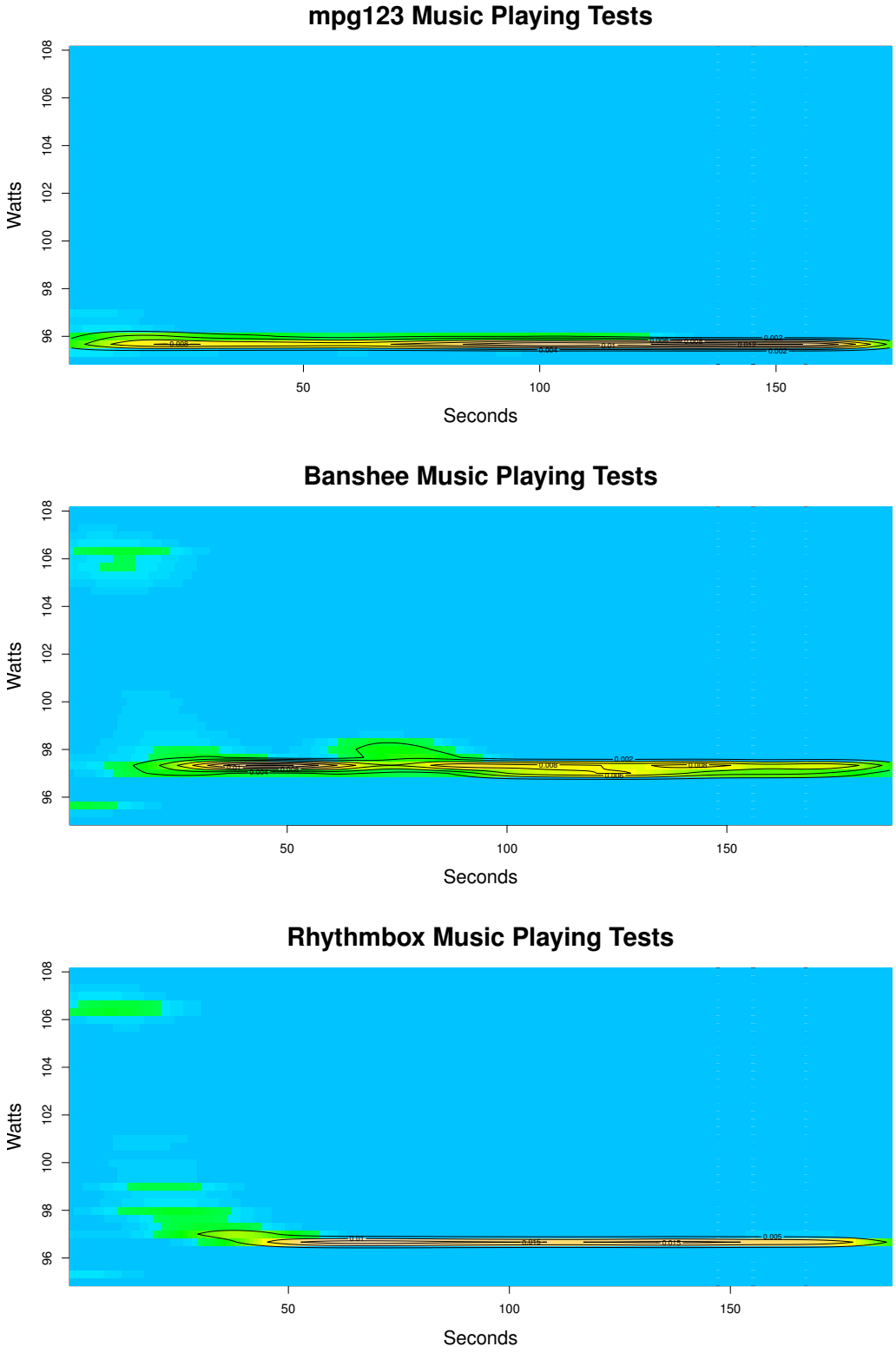

Figure 3.6: Density of the power measurements from music playing scenario tests using `mpg123`, Banshee, and Rhythmbox.

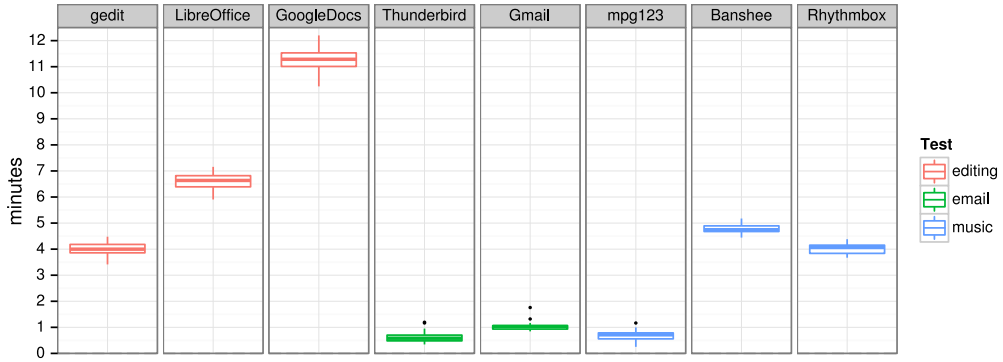

Figure 3.7: Distributions of mean battery life consumed per application using desktop results based on the Big laptop model.

as to tune the λ in Equation 3.1 for X31 laptop model based on the power measurements of our previous desktop testbed. The Lenovo ThinkPad X31 was running 32-bit Ubuntu 12.04 as well. Figure 3.8 plots the distributions of the mean power consumption for each application running on the X31 testbed. The exact reduced battery life for running the tested applications on X31 laptop model, as shown in Figure 3.9, can be computed based on the power measurements from the X31 testbed and X31 laptop model. Using the power measurements on the desktop testbed and a random value for λ, we can estimate the reduced battery life for running all the applications on X31 laptop model. Based on the evaluation method, the mean absolute error (MAE) [35], we can determine the best λ for estimating the reduced battery life using power measurements from the desktop testbed on X31 laptop model. We found that the MAE is the smallest when λ is equal to 0.79.

Figure 3.10 presents the distributions of the reduced mean battery life estimated by the desktop power measurements for all the applications ($\lambda = 0.79$). The impact of power consumption variations on battery life is more obvious than that in Figure 3.7. For example, the difference of energy consumption between `gedit` and Google Docs was only 4.2 watts in the text editing scenario, but using `gedit` will gain about 32 minutes more battery life than using Google Docs to type text in X31 laptop model. Thus we can see the impact of small changes in power on energy consumption by using battery life models.

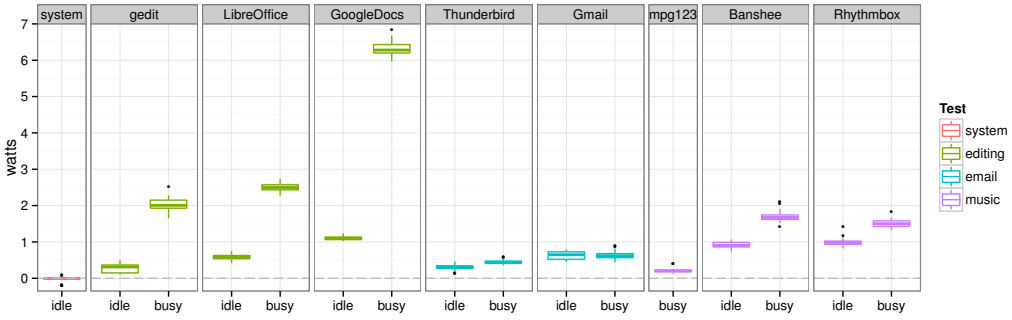

Figure 3.8: Distributions of the mean watts consumed per test: idling on testbed, text editing, email receiving and music playing tests on X31 testbed. 40 tests each, 640 tests total.

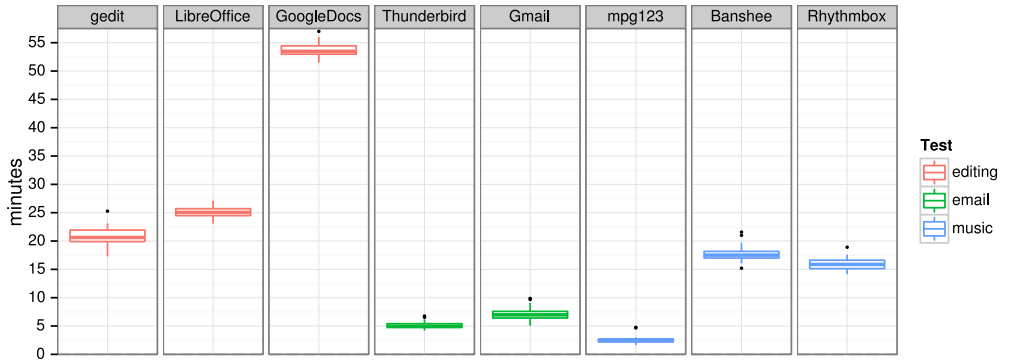

Figure 3.9: Distributions of mean battery life consumed per application using X31 results based on the X31 laptop model.

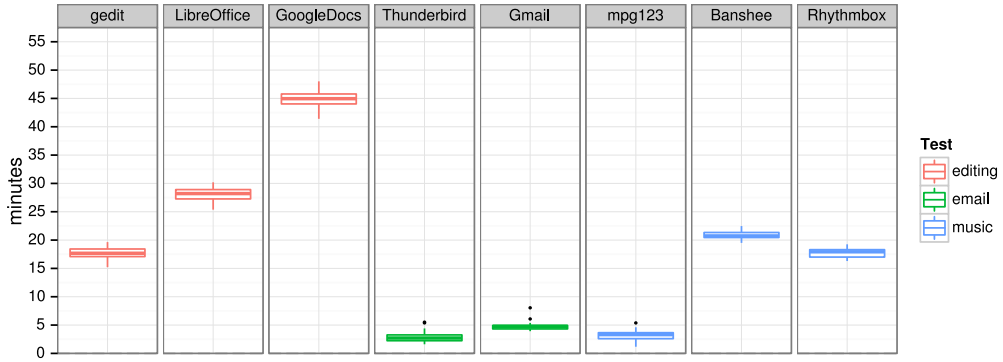

Figure 3.10: Distributions of mean battery life consumed per application using desktop results based on the X31 laptop model.

3.5 Discussion

3.5.1 Functionality Versus Consumption

It is not surprising that more functionality often leads towards an increase in energy consumption. Due to their simplicity, both `gedit` and `mpg123` are expected to perform better than their counterparts. On the other hand, the use of the cloud makes both Google Docs and Gmail less efficient than LibreOffice and Thunderbird (keep in mind that we are only measuring the local consumption—Google services require energy of the networking infrastructure and Google servers to run that we have not taken into consideration).

A user who is not concerned with energy consumption is likely to choose the software that provides the most desired features (functional and non-functional). However, a user that is concerned with energy consumption (such as our stranded user running on battery power) would consider energy consumption a priority. For example, start by typing the document in `gedit`, then spell checking it in LibreOffice and finally upload it to Google Docs (if the document must be uploaded to the cloud). Similarly, the interface is probably less important for our user than the ability to listen to music.

In the future, applications might document the impact on energy consumption for each of their features, and include different options for different levels of energy

consumption similar to the way OSes do it; e.g., a menu option for low power or normal energy consumption where some less important features, such as GUI animations, are disabled. Users who must optimize energy consumption will be capable of making informed decisions in terms of what and when to run.

Uses who are concerned about energy consumption should balance the advantages of a feature versus its energy cost, and plan their use of applications based upon these factors.

3.5.2 Causes of Energy Consumption

In the tests we conducted, the main causes of energy consumption we observed from our results were:

- Synchronization with the cloud: Google Docs suffered greatly from constantly synchronizing the document in the cloud.
- Web based applications are less efficient than their stand-alone counterparts.
- Heavy startup: Banshee and Rhythmbox perform many tasks before it is ready to perform the tasks required.
- Continuous events are expensive: Spellcheckers hurt both Google Docs and LibreOffice as they increased the number of events per keystroke. Future work should compare the energy consumption of spell checking done during vs after typing a document.
- UI updates are expensive: UI interfaces that are continuously updated consume a significant amount of energy. We were surprised by how little energy was consumed playing music using `mpg123` (less than 0.4 watts—it is clear that the hardware is optimized to play music files). The other players used more than five times more energy.

3.5.3 Ghost Energy Consumption

Many idle applications continue to consume energy. As we demonstrated in Figure 3.1, 3.3 and 3.5, a lot of applications use power even if they are not finishing any

tasks. Users tend to keep applications open, ready to be used. Since idleness often makes up a size-able percentage of time for many of the tasks it is quite important. Application developers should consider balancing the benefits and disadvantages of using idle-time for any further processing. Also application developers should consider reducing the number of events that occur during idle time, since for certain applications idle-time dominates (such as our email tests).

3.5.4 Application Ratings

As the results herein demonstrate, different applications have different energy consumption profiles, even when they are asked to performed the same task. An even bigger concern is that applications consume energy even when they are idle. We expect that, as hardware and operating systems become more efficient, the focus will turn towards the energy consumption of applications.

In general, users are unlikely to have the equipment, the expertise or the time to measure the energy consumption of applications. This is complicated by the lack of simple-to-use user-space tools that can help estimate how much energy an application consumes, such as those that exist to monitor the overall operating system (such as Intel's PowerInformer [3]; or the monitoring tools for software developers, such as Intel's Power Checker [4] and Microsoft's Event Tracing [5]). As energy continues to become a growing concern, both users and developers will be expected to report the energy efficiency of the applications they use/create. This will create a positive feedback-loop: users will expect energy efficiency, and developers will be expected to deliver it.

We propose the creation of *Software Application Energy Consumption Ratings* (SAECR). A SAECR has three main goals: 1) to define a framework and a methodology for consistent measuring of consumption of applications; 2) to create guidelines for the creations of benchmarks that represent typical user needs; and 3) to simplify the reporting of results in a manner that is easy-to-understand by typical

[3] http://software.intel.com/en-us/articles/intel-powerinformer/

[4] http://software.intel.com/partner/app/software-assessment

[5] http://msdn.microsoft.com/en-us/library/bb968803%28v=vs.85%29.aspx

Table 3.3: An example of software application energy consumption ratings ranging from A to C. A means the most energy efficient and C means the least energy efficient.

Task	Application	Rating
Text Editing	`gedit`	A
	LibreOffice Writer	B
	Google Docs	C
Email	Mozilla Thunderbird	A
	Gmail on Mozilla Firefox	B
Music Playing	`mpg123`	A
	Banshee	C
	Rhythmbox	B

users. A basic SAECR is shown in Table 3.3 based on our current results.

SAECR would be similar to those in other areas, such as those in networking (the Network and Telecom Equipment Energy and Performance Assessment, fostered by the Energy Consumption Rating Initiative [6]). It is not unfeasible that in the future Government organizations become involved, and extend their energy rating programs, such as Energy Star [7] (originally American but now used in Canada, the European Union and other countries), the Canadian EnerGuide Label [8], and the European EU Directive 92/75/EC (European Union Energy Label), to include software.

We believe that a SAECR can start with two benchmarks. The first, measuring the ghost energy consumption of applications. The second, applicable to applications that are expected to continuously perform the same task without user interaction (such as playing a movie, playing sound, a desktop widget, etc). In both cases the benchmark is straightforward (run the application and measure its consumption). These tests will allow users to compare the energy efficiency of similar applications. As mentioned before, this comparison will force developers to improve the energy efficiency of applications they develop and will allow users to determine

[6] `http://www.ecrinitiative.org/`
[7] `http://www.energystar.gov/`
[8] `http://oee.nrcan.gc.ca/equipment/appliance/15538`

what applications they should use (or stop using) when they are concerned with energy consumption (e.g. running on battery power).

3.6 Threats to Validity

Construct validity is threatened by the accuracy of our measurements, *Watts Up? Pro* have limited resolution (0.1 watt), and the test-cases we posed. Were our test cases representative of real users? We tried to make the test cases realistic, and we recognize they might not cover all users. We used UI recording tools in order to ensure the input was realistic, but the UI driver could provide excessive overhead and use different kinds of interrupts when compared with keyboards and mice. Other construct validity threats relate to our idea of idle behaviour where we measure the entire system and then attribute energy usage during our tests as the usage on top of the idle usage. Our networked tests do not measure the energy used in the server-side (such as Google's servers).

Internal validity is threatened by our battery life estimation models, which are simplifications of battery life on laptops. Our assumption that 1 Watt above idle on the desktop is similar in scale to 1 Watt above idle on a laptop can be challenged, but we allowed for tuning the scale of the effect. We argue that the direction and relative magnitude are close but the numbers are not necessarily as accurate as running on the actual device. To address the effect of a web browser on a scenario we measured an idle Firefox to help explain if Firefox itself was the primary perpetrator of energy consumption.

External validity was bolstered by the breadth of the scenarios we tested. But external validity was harmed by not using multiple software and hardware platforms while measuring the energy consumption of the scenarios. However, our tests are meant to illustrate that different applications consume different amounts of energy. We expect that in other computers and operating systems this difference is still observable (even if the measured values are different).

3.7 Chapter Summary

Very frequently users have a choice of what application to use to complete a task. In this chapter we have proposed a user-centric method to measure the energy consumption of applications based upon the scenarios that correspond to tasks that uses are expected to complete. We demonstrate its effectiveness by defining three such scenarios, and an implementation of benchmarks to measure the consumption during each of them.

The results indicated that different applications can have dramatically different energy consumption when performing the same task (e.g. a command-line music player uses more than five times less energy than a GUI one). We also found that web-based applications tend to consume more energy than non-web based, and that idle applications can incur a significant amount of ghost energy consumption.

Unfortunately it is not trivial for users to know which applications are more energy efficient. We expect future work to be directed towards the creation of benchmarks and reporting mechanisms (similar to Energy Star) that inform developers and users of the energy efficiency of their applications. This will likely generate pressure on the developers to improve the energy efficiency of the applications they develop.

Users, by making a conscious decision to use applications that are optimized towards energy consumption, can improve the battery life of their mobile devices, and perhaps more importantly, contribute towards helping the environment.

Chapter 4

Mining Multiple Versions of Software on Energy Consumption

In the previous chapter, we studied the impact of user choice on energy consumption of software systems. As the builders of software, software developers are responsible for the energy behavior of the software shipped to users. Most importantly, when making a change in the software, developers do not want to jeopardize the energy efficiency of software. Although a lot of power models have been studied from different levels of granularity [11], [37], [7], [25], [23], [30], [12], when it comes to make power-efficient changes in software, informed decisions can be answered by few of the existing studies.

In order to help developers understand the software power behavior when making changes, we provide a method and a case study to reveal the correlation between software change and power consumption. To be specific, we first investigated the software power consumption evolution over multiple versions. Then, we traced the system calls invoked by a list of software versions. At last, we leveraged system call invocations to model software power consumption over versions. The results show different power consumption behaviours in terms of multiple software versions and different test cases. Also, system calls act as the entry points into the OS kernel. System call invocations have the potential of modeling software power consumption based on multiple versions.

This chapter is organized as follows. Section 4.1 introduces system calls and their interactions with both the user applications and the OS kernel. Our method-

ology is presented in Section 4.2 and our case study and results are explained in Section 4.3. Section 4.4 discusses about the relationship between software changes and power consumption based on our results. The threats to validity in this study are mentioned in Section 4.5 and we conclude this chapter in Section 4.6.

4.1 System Calls

System calls are standard functions implemented between the OS kernel and user processes. They are entry points to the OS kernel and provide services for user processes when the processes need to access services in kernel mode. Usually such services include communicating with the hardware (such as accessing the hard disk), creating and executing new processes, and communicating with integral kernel services [32]. Figure 4.1 presents the relationships among user applications, C library functions, system calls and the OS kernel [32]. A library function and a system call can both be invoked by the application code. While only system calls can interact with the OS kernel. If a library function wants services from the OS kernel, it needs to call system calls, too.

We can expect different revisions of software invoke different system calls during their execution if they differ from each other in which services to get and how to get the services from the OS kernel. As an essential interface sitting between the application and the OS kernel that triggers hardware utilization and other kernel services, *can system calls provide a set of features for making predictions of software power consumption?*

4.2 Methodology

In this section, we explain the methodology for collecting and analyzing the software power consumption among multiple versions of software. The general process is derived from the previous work on *Green Mining* and also modified from the methodology in Section 3.3.

1. Choose and build multiple versions of a software product.

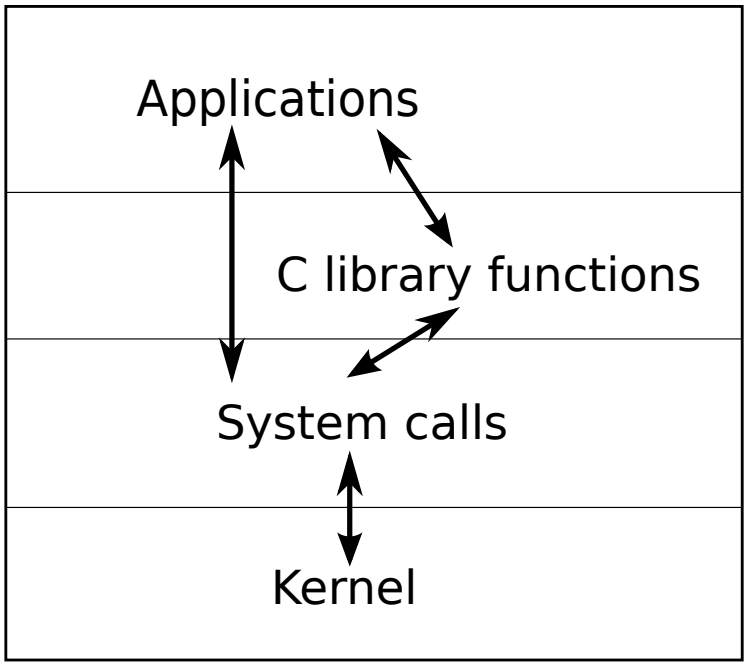

Figure 4.1: This diagram shows how applications, C library functions, system calls, and kernel interact with each other.

2. Decide on the level of instrumentation.

3. Develop the test cases to run on the software.

4. Run the tests and analyze results.

4.2.1 Choosing and Building Multiple Versions of a Software Product

We chose `gedit`, one of the text editors studied in Chapter 3, as the software to be tested. This general purpose text editor has been developing for more than 10 years. It is written in C and Python, and is the default text editor for GNOME desktop.

In order to build multiple versions of `gedit`, we relied on the Git repository of `gedit` [1]. When choosing commits to compile `gedit`, we only considered releases. From release 0.7.9 to release 3.7.3 (commits starting from June 2000 to February 2013), we were able to build 39 revisions of `gedit`, covering most of the `gedit` 2 (18 builds) and `gedit` 3 versions (21 builds).

[1] Git repository of `gedit`, `https://git.gnome.org/browse/gedit/`

4.2.2 Deciding on the Level of Instrumentation

We recorded the power consumption of `gedit` and system calls invoked by `gedit` for each `gedit` revision. In terms of the power measurement, we utilized the same power meter as we did in Chapter 3. For recording system calls on the testbed, we applied the debugging utility for Linux, *strace* [2]. It is able to trace the name of each system call, its arguments and its return value called by a process or a program. Figure 4.2 shows a part of *strace* output for the Linux command `date`. The listed system calls invoked by a program could help us detect unexpected behaviours and thus *strace* is often used to debug a program.

```
read(3, "\nMST7MDT,M3.2.0,M11.1.0\n", 4096) = 24
close(3)                                = 0
munmap(0xb7715000, 4096)                = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7715000
write(1, "Tue Jul  2 19:35:57 MDT 2013\n", 29) = 29
close(1)                                = 0
munmap(0xb7715000, 4096)                = 0
close(2)                                = 0
```

Figure 4.2: An example of *strace* partial output for the Linux command `date`.

4.2.3 Developing the Test Cases

Changes in software are scattered with various features. In order to correlate software changes with software power consumption, we need to implement a set of test cases that are able to trigger different features. In this thesis, we developed two test cases for `gedit`. The first one is about text editing, which is forked from Section 3.3.3. The second test case is about syntax highlighting. `gedit` is able to highlight syntax for a number of program languages and text markup formats. We will present the details of each test case in the next section.

4.2.4 Running the Tests and Analyzing Results

The testbed is a Lenovo X31 laptop, which was configured and used in Section 3.4.5. The configuration of the testbed stays the same. For each test case, we ran 11 tests over each `gedit` version we built. The first test was to trace the system calls and

[2] strace, `http://sourceforge.net/projects/strace/`

the rest of the tests were to measure the power consumption of `gedit`. System calls for each version tend to be stable and we just traced them once for each revision of `gedit`. For power measurement, in order to determine normality and differences between measurements, we chose the number of tests to be 10. So in total we have 390 tests in terms of power measurement.

Hence, for each `gedit` revision under each test case, there is one record of system calls and ten records of power measurement. We also grouped the system calls by names and counted the number of invocations of each system call for each `gedit` revision to form the counted system calls matrix.

4.3 Case Study

In this section we present the results of our two `gedit` case studies. Each case study intends to focus on one specific functionality in `gedit` so as to discover the correlation between `gedit` power consumption with software changes in `gedit`. The first case study focuses on the text editing functionality and the second case study focuses on the syntax highlighting feature.

4.3.1 Text Editing

The test scenario for text editing is the same as we implemented in Section 3.3.3. Figure 4.3 displays the results of each `gedit` version's power consumption. Each boxplot in the figure consists of the mean power consumption of every `gedit` version in 10 tests and there are 390 tests in total. The red line draws the fluctuation of all the mean power consumption in 39 `gedit` versions. The plot clearly shows the different power consumption between version 2 and version 3 of `gedit`. Within each major version, different revisions of `gedit`'s power consumption vary in a limited interval.

Applying Student's t-test to the tests for each pair of `gedit` versions, we can determine if the comparisons of the mean power consumption among versions are statistically significant. Figure 4.4 shows the pairwise Student's t-test results. There are two obvious clusters in terms of the p-values. The power consumption

of `gedit` 2 versions and 3 versions are totally different since the p-values between each version's mean power consumption are 0. However, in each major release, most of the `gedit` versions' power consumption are not significantly different, except three versions, 3.0.1, 3.0.2, and 3.0.3.

By comparing the system call counts for each `gedit` version in text editing test, we can investigate how the system calls invoked by each `gedit` version vary under the test case. There are 79 different system calls in the counted system call matrix and for each `gedit` version there is one system call vector which has 79 dimensions. All the system calls are listed in Appendix A.1. We applied cosine similarity to each pair of the system call vectors. Figure 4.5 reports the pairwise cosine similarities for each `gedit` version. It shows the similar pattern compared to the pairwise Student's t-test of power consumption in Figure 4.4. Although the cosine similarities are all close to 1, they show two clusters in `gedit` 2 versions and 3 versions, which implies the system calls invoked by the two major versions vary from each other. For example, the number of invocations of the system call, `mmap2` (map files or devices into memory), increases from around 300 in `gedit` 2 versions to 500 in `gedit` 3 versions. While the system call `fsync` (synchronize a file's in-core state with storage device) shows a decreasing trend. The number of `fsync` invocations drops from 6 in `gedit` 2 versions to 1 in `gedit` 3 versions.

Furthermore, we applied linear regression to the `gedit` power consumption and system call invocations in each `gedit` version in order to have a closer investigation of their relationship. A large number of the system call counts are highly correlated with each other (most of the Spearman's correlations $|\rho| > 0.8$). So we only modeled `gedit` power consumption over versions based on individual system call. Table 4.1 presents 10 system calls, their Spearman's correlations ρ with `gedit` power consumption, their associated R^2 values, and coefficients in linear regression with relatively high R^2 values. The descriptions of these system calls are listed in Table 4.2. In addition, the total number of system call invocations for each `gedit` version has a small negative coefficient, -3.93$\times 10^{-5}$ and a R^2 of 0.549 in the linear regression model for `gedit` power consumption. It is not surprising that system calls related to memory and I/O operations have higher R^2 values since the

Table 4.1: Some of the system calls and the associated Spearman's correlation ρ with `gedit` power consumption, R^2 values, and coefficients in linear regression for the text editing test case. For each system call we build a model of the form: $y = b_1 \cdot x + b_0$, where y is `gedit` power consumption and x is the number of system call invocations. All the results are statistically significant ($p < 1.00 \times 10^{-8}$).

System Call	ρ	R^2	Coefficient
`shmget`	0.864	0.987	0.157
`getrlimit`	-0.864	0.987	-0.0784
`mmap2`	-0.728	0.986	-1.64×10^{-3}
`ftruncate`	-0.871	0.986	-0.0534
`mprotect`	-0.706	0.984	-5.46×10^{-3}
`fsync`	0.856	0.978	0.327
`close`	-0.659	0.968	-6.66×10^{-3}
`read`	-0.720	0.654	-4.15×10^{-5}
`writev`	-0.719	0.632	-4.08×10^{-5}
`poll`	-0.710	0.632	-2.06×10^{-5}

test scenario is to edit text in a file.

In the text editing test case, `gedit` power consumption stays in the same level in terms of 2 major versions and 3 major versions respectively. There have not been any important changes in terms of the basic feature, text editing, among versions in each major release. However, there is a large gap (about 1.5 watts) between the two major versions' power consumption. The linear regression modeling results show a list of system calls that can be better fitted with `gedit` power consumption are related to memory management and file operations.

4.3.2 Syntax Highlighting

This case study is to investigate the relationship between `gedit` power consumption under another feature, syntax highlighting, and the system calls invoked by various `gedit` versions. The test case intends to simulate a real user reading through a variety of programming and text markup language code. There are six files to read and each file is a source code (C, Java, Perl, and Python) or text from a markup

Table 4.2: Selected system calls with their descriptions from the text editing test case.

System Call	Description [33]
`shmget`	allocates a shared memory segment
`getrlimit`	get resource limits
`mmap2`	map files or devices into memory
`ftruncate`	truncate a file to a specified length
`mprotect`	set protection on a region of memory
`fsync`	synchronize a file's in-core state with storage device
`close`	close a file descriptor
`read`	read from a file descriptor
`writev`	write data into multiple buffers
`poll`	wait for some event on a file descriptor

Figure 4.3: Distributions of the mean watts consumed per version of `gedit` running 10 editing text test (390 tests in total). The X axis represents the version numbers and the Y axis is the power consumption.

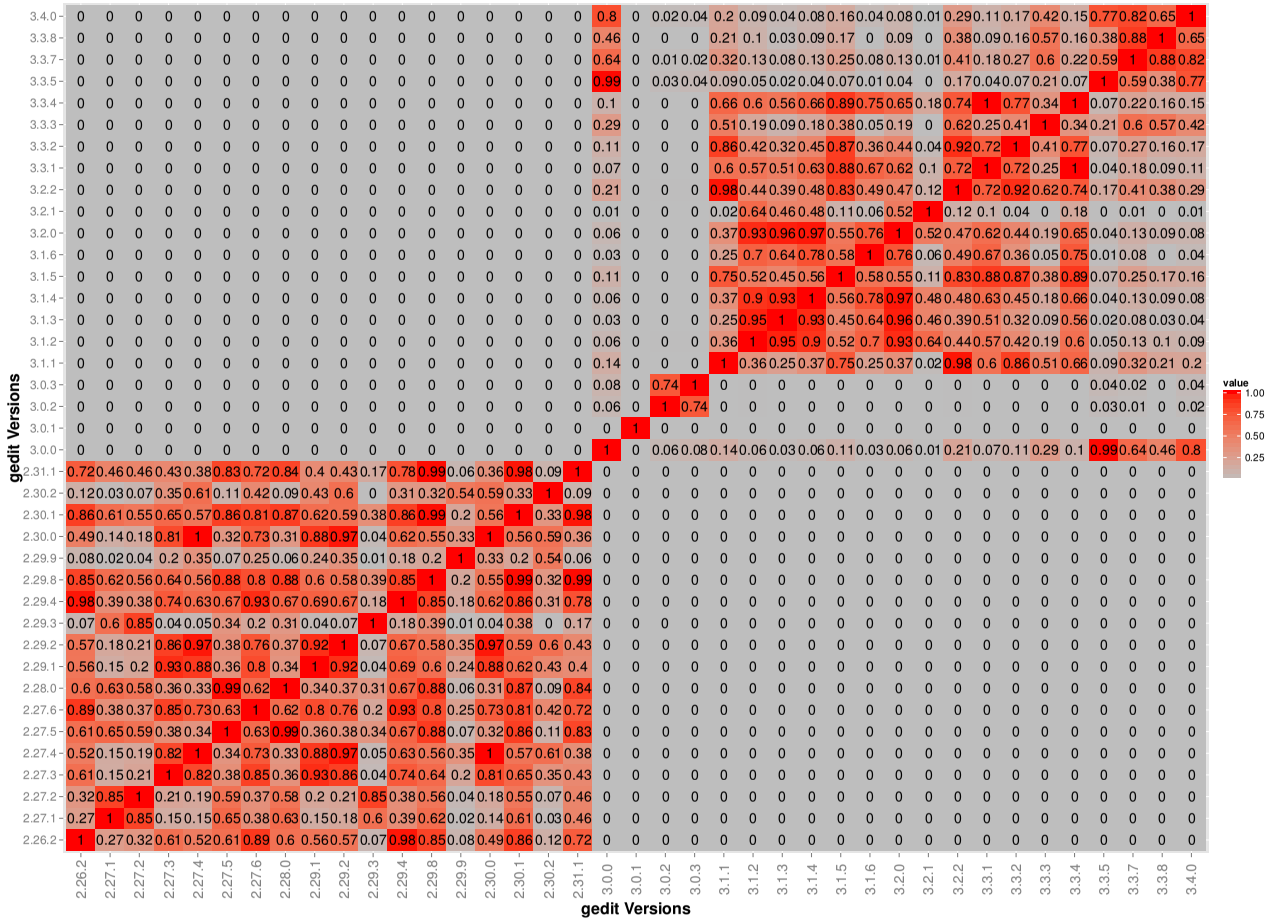

Figure 4.4: Pairwise Student t-test for each `gedit` version's mean power consumption in 10 text editing tests. The X axis and the Y axis represent the version numbers. Each value in the plot is the p-value. "Grey" means the p-value is close to 0 and "red" means the p-value is close to 1.

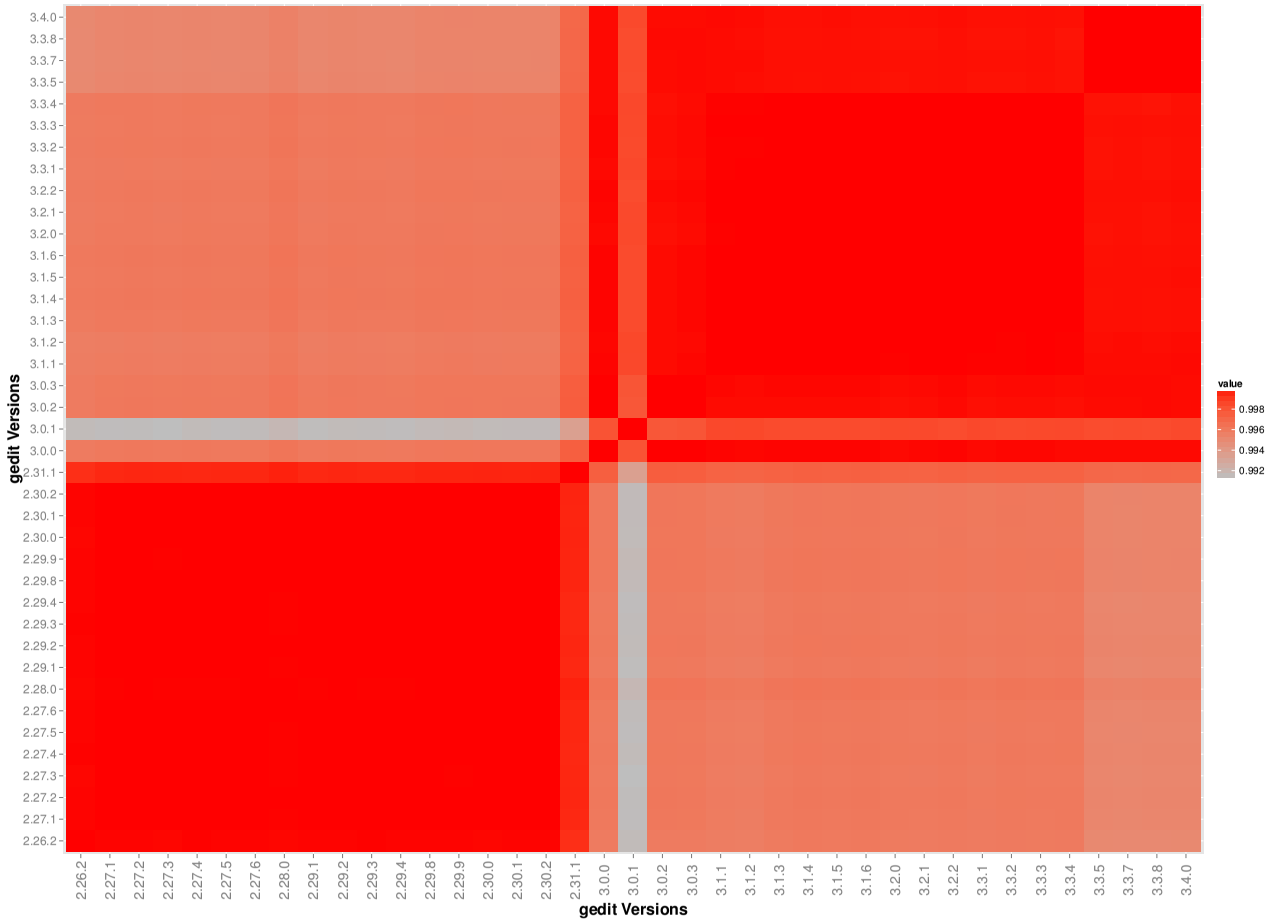

Figure 4.5: Pairwise cosine distance for each `gedit` version's system call vector in text editing tests (79 system calls). The X axis and the Y axis represent the version numbers. "Grey" means smaller cosine similarities and "red" means larger cosine similarities.

language document (HTML and LaTex) that has more than 300 lines. The test took more than 7 minutes to go through all the six files.

To be specific, the test procedure is 1) open the six files in `gedit`; 2) scroll down to go through the first file in every few seconds until reach the last line; 3) move to the next file and repeat step 2; and 4) close `gedit` when finish going through the last file.

The distribution of the mean power consumption in each version of `gedit` running syntax highlighting test is shown in Figure 4.6. Each boxplot represents the mean power consumption distribution of the 10 tests for each `gedit` version and therefore we have 390 tests in total. The variations of the mean power consumption among 39 version is pictured by the red line. There are two levels within `gedit` version 2 and also version 3 respectively. In version 2, it is increasing trend, while in version 3 the trend is the opposite. This is different from the pattern in Figure 4.3.

The pairwise Student's t-test results of each `gedit` version's mean power consumption for 10 tests in Figure 4.7 confirm our observation from Figure 4.6. Within `gedit` 2 versions there are two clear clusters in terms of p-values from t-test, which tells us the power consumption has two levels that are statistically significant in `gedit` 2 versions. The pattern of p-values in `gedit` 3 version is more complex than that of `gedit` 2 versions. The p-values form three clusters, which suggests there are three levels of power consumption that are statistically significant in `gedit` 3 versions. Moreover, some of the `gedit` 3 versions have similar power consumption compared to several early `gedit` 2 versions.

We also counted the number of each system call's invocations for each `gedit` version under the test case so as to determine how the system calls change among different versions. In total, there are 77 system calls invoked by each `gedit` version and all the system calls are listed in Appendix A.2. The cosine similarities between each pair of system call vectors for each `gedit` version are shown in Figure 4.8. Although it shows two levels of power consumption in Figure 4.6 for `gedit` 2 versions, the number of system call invocations in these versions tend to be similar. While in `gedit` 3 versions, the variations of system call counts correspond with the fluctuation of power consumption and the pairwise cosine similar-

ities group three clusters. A typical instance that confirms the variation of system call invocations is the system call, `brk` (change data segment size). The number of `brk` invocations in `gedit` 2 versions is above 200 and in `gedit` 3 versions it stays below 150. It is still changing within 3 versions. It is around 120 in the early `gedit` 3 versions and then it increases to about 150. In the latest four `gedit` versions, it drops below 100.

We also applied linear regression to the `gedit` power consumption and the number of system call invocations so as to discover the important system calls. Only one independent variable was taken into consideration when the model was built since the system call counts suffer high correlation with each other (most of the Spearman's correlations $|\rho| > 0.8$). Table 4.3 shows the selected 10 system calls, their Spearman's correlations with `gedit` power consumption, R^2 values, and their corresponding coefficients in linear regression. The descriptions of these system calls are listed in Table 4.4. Under this test case, the total number of system call invocations for each `gedit` version is not legitimate for modeling the power consumption because the p-value is quite large.

To summarize, `gedit` power consumption fluctuates among versions in the syntax highlighting test case. In `gedit` 2 versions, the power consumption increases from version 2.28.0 and drops down from version 2.31.1. In `gedit` 3 versions the power consumption varies from each other at early versions and then remains flat. At last the latest four versions stay in the lowest power consumption level in this test. The system calls that have better linear regression modeling results are also related to memory management and file operations but different from the specific ones in text editing test case.

4.4 Discussion

4.4.1 System Calls and Power Consumption

The two test cases discussed above are basically about writing/reading files in `gedit`. It is not surprising that those system calls related to memory management

Table 4.3: Some of the system calls and the associated Spearman's correlation ρ with `gedit` power consumption, R^2 values, and coefficients in linear regression for the syntax highlighting test case. For each system call we build a model of the form: $y = b_1 \cdot x + b_0$, where y is `gedit` power consumption and x is the number of system call invocations. All the results are statistically significant ($p < 4.50 \times 10^{-4}$).

System Call	ρ	R^2	Coefficient
`_llseek`	-0.922	0.561	-0.0373
`brk`	0.770	0.539	2.77×10^{-3}
`mmap2`	-0.747	0.466	-1.65×10^{-3}
`dup2`	-0.827	0.459	-0.0196
`ftruncate`	-0.827	0.459	-0.0522
`open`	-0.655	0.381	-8.80×10^{-4}
`fsync`	0.686	0.371	0.0486
`close`	-0.654	0.346	-9.59×10^{-4}
`fstat64`	-0.692	0.289	-1.75×10^{-3}
`eventfd2`	-0.624	0.287	-4.40×10^{-3}

Table 4.4: Selected system calls with their descriptions from the syntax highlighting test case.

System Call	Description [33]
`_llseek`	reposition read/write file offset
`brk`	change data segment size
`mmap2`	map files or devices into memory
`dup2`	duplicate a file descriptor
`ftruncate`	truncate a file to a specified length
`open`	open and possibly create a file or device
`fsync`	synchronize a file's in-core state with storage device
`close`	close a file descriptor
`fstat64`	get file status
`eventfd2`	create a file descriptor for event notification

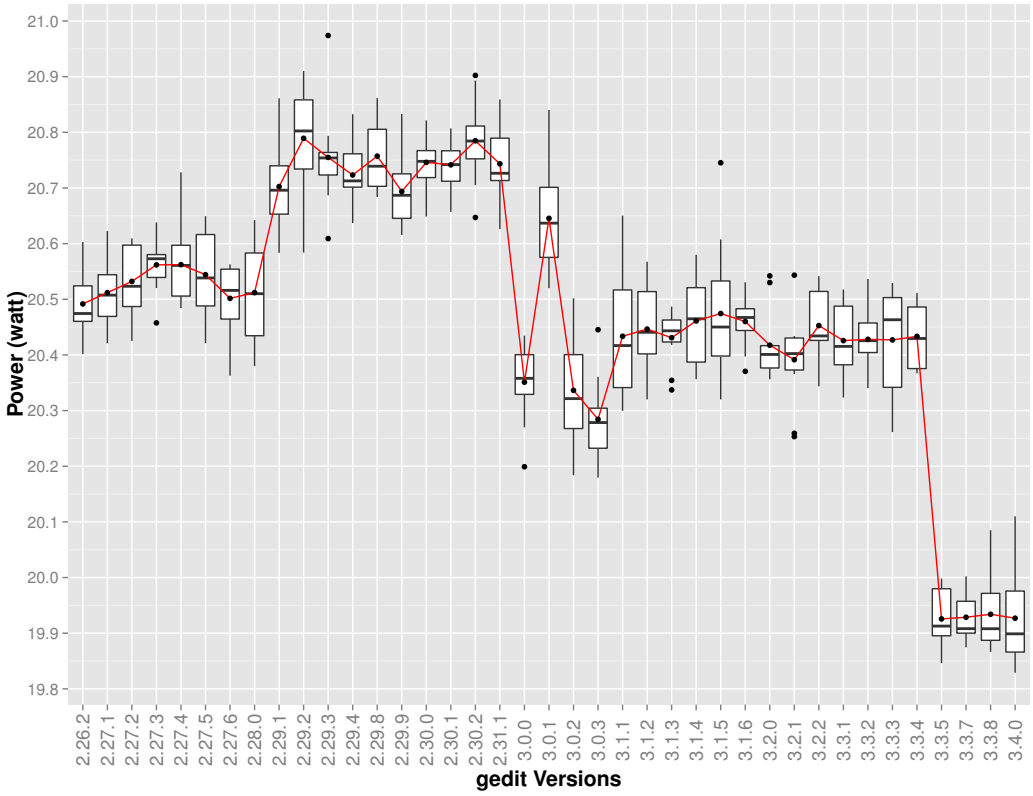

Figure 4.6: Distributions of the mean watts consumed per version of `gedit` running 10 syntax highlighting tests (390 tests in total). The X axis represents the version numbers and the Y axis is the power consumption.

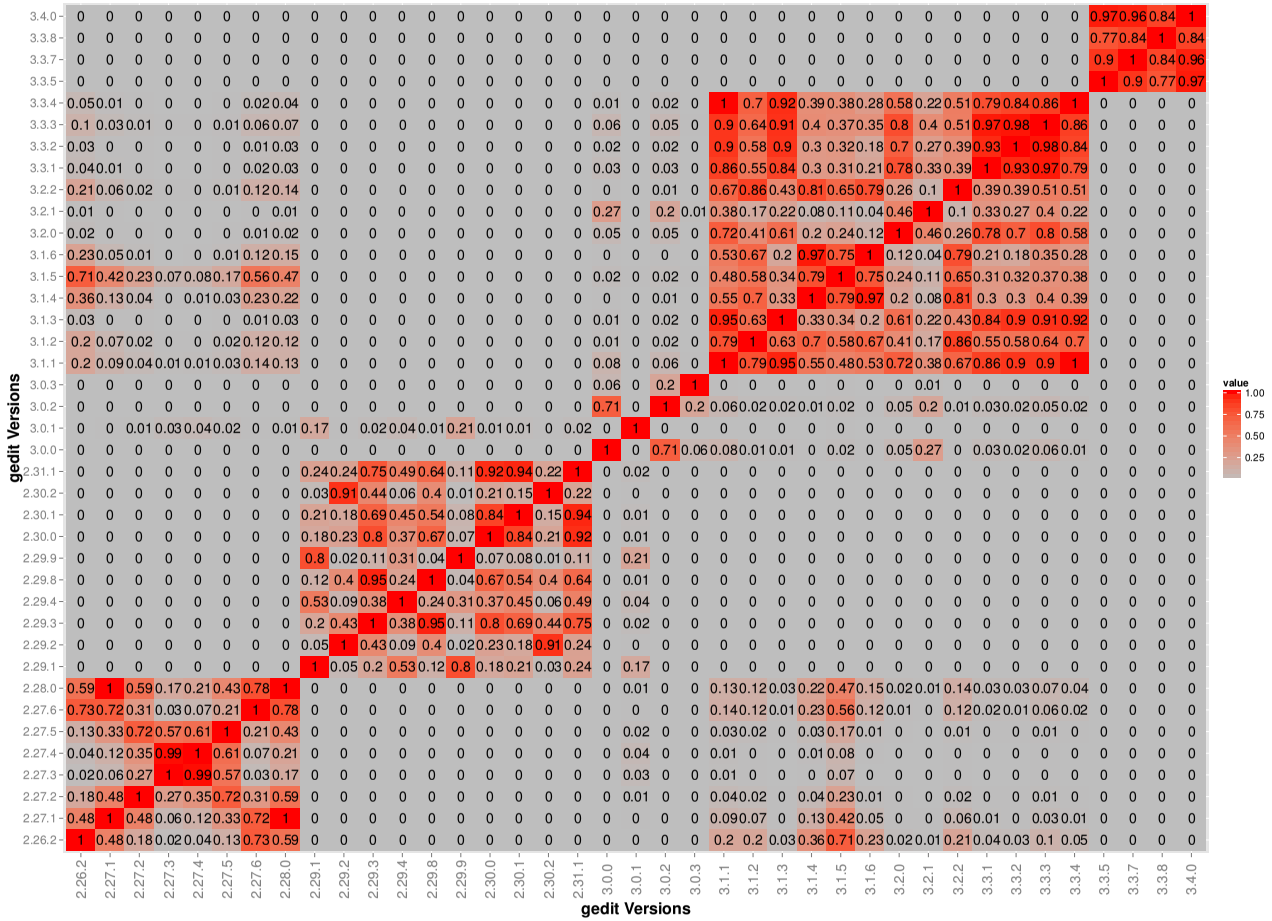

Figure 4.7: Pairwise Student t-test for each `gedit` version's mean power consumption in 10 syntax highlighting tests. The X axis and the Y axis represent the version numbers. Each value in the plot is the p-value. "Grey" means the p-value is close to 0 and "red" means the p-value is close to 1.

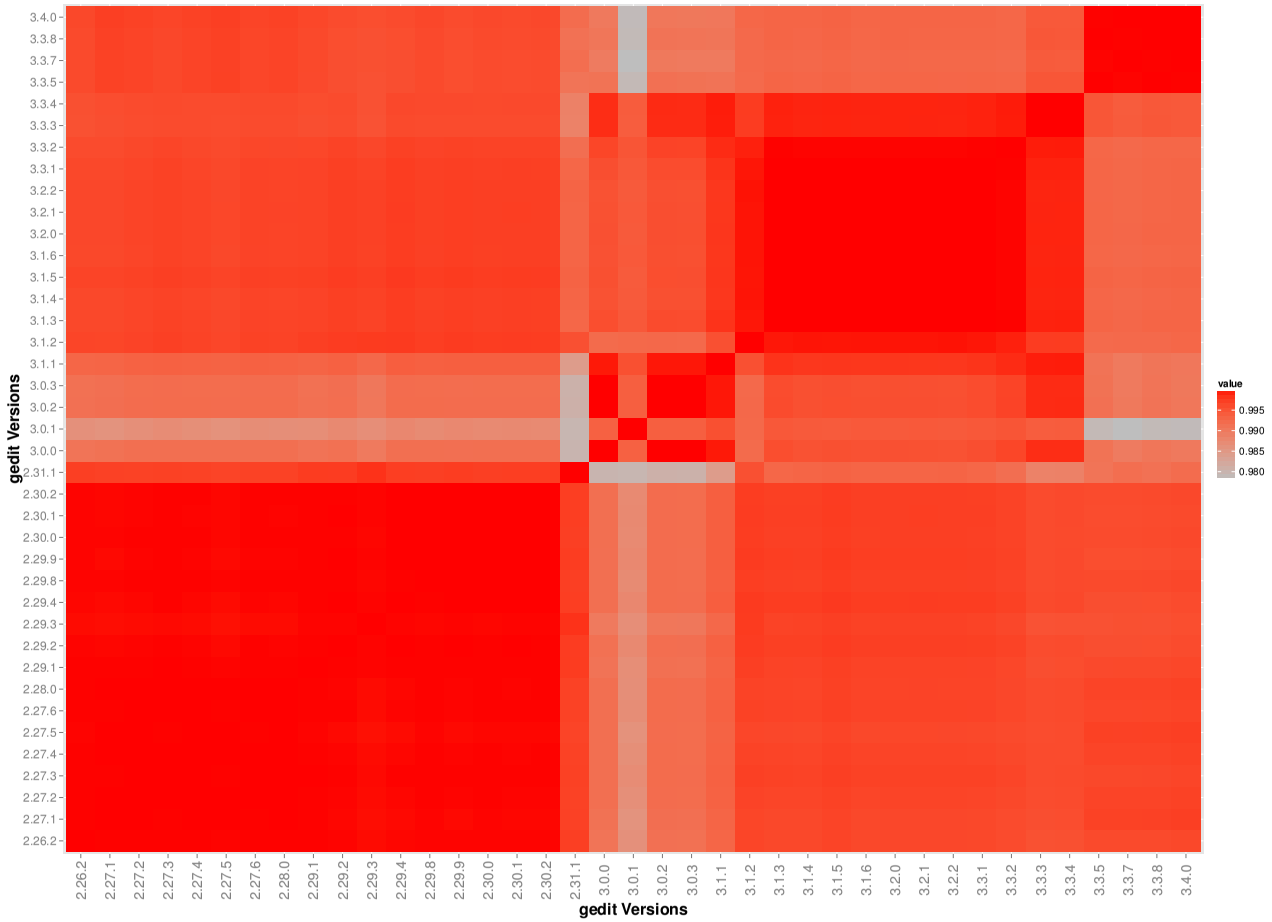

Figure 4.8: Pairwise cosine distance for each `gedit` version's system call vector in syntax highlighting tests (77 system calls). The X axis and the Y axis represent the version numbers. "Grey" means smaller cosine similarities and "red" means larger cosine similarities.

give us better performance in power modeling, since they are correlated with intensive I/O operations. The system calls that have relatively better modeling results in the two test cases do not overlap with each other that much and hence different test cases tend to get different services from OS kernel. Furthermore, we can make use of the variations in system call invocations to locate the software changes that would be responsible for power exhaustion. On the other hand, most of the system calls have high correlation with each other in each test case, which suggests that the code coverage of each test case might be limited. More test cases and more test cases for various applications over versions are definitely needed for obtaining the hidden relationship between all kinds of system calls and software power consumption.

4.4.2 Software Changes and Power Consumption

From the case study section, we've learnt the different power behaviors among multiple `gedit` versions for finishing the same functionality. A deeper investigation about software changes in `gedit` was conducted by checking the `diff` in source files between each two continuous versions. From `gedit` 2.31.1 to 3.0.0, which are the the last `gedit` 2 version and the first `gedit` 3 version in our `gedit` compiled binaries correspondingly, there has been a lot of UI updates and bug fixes. Moreover, a list of dependencies have been improved, such as `glib`, `gtk` and `gtksourceview`. In both test cases, the power consumption has a large drop between these two `gedit` versions. Possibly, the improved dependencies are closely related to the power consumption decrease. From Figure 4.3 and Figure 4.6, we can observe that the power consumption of `gedit` 3.0.1 is not consistent with its neighbors. `gedit` 3.0.1 has been applied style improvements using an external CSS file. The following versions deleted this CSS file and changed the way to do styling. This unique change might be responsible for the inconsistent power characteristics of `gedit` 3.0.1 compared to other versions.

4.5 Threats to Validity

Since we utilized the same power measurement setup and UI recording tools as we did in the previous chapter, construct validity is threatened by the accuracy of our power meter and the overhead from UI driver. Other construct validity threats come from the assumption that system call invocations tend to be stable in each run of test case for the same software version. Base upon this assumption, system call tracing and power measurement for each software version are separated under each test case in order to avoid power overhead in the system call tracing test run. Therefore, the power consumption to correlated with system calls is estimated by other 10 test run of power measurement, instead of the actual software power consumption when running system call tracing under the test cases.

Internal validity is weakened by the high correlation between each pair of system call invocation counts under each test case. We were only able to model `gedit` power consumption based on one system call each time. Also, `gedit` did not have much variation, except on major releases. Thus, the relationship between system calls and software power consumption is inconclusive.

External validity is harmed by both the limited test cases for `gedit` and the lack of divers software to be tested. We implemented two test cases for `gedit` but they are not inevitably covering the changed code. External validity would be improved by implementing more detailed test cases to cover wider breath of code as well as adding various software to run test on.

4.6 Chapter Summary

In this chapter, we have proposed a method to investigate the impact of software changes on software power consumption as well as the relationship between system call invocations and software power consumption over versions. We implemented our method on multiple versions of the text editor, `gedit`. Two test cases about text editing and syntax highlighting have been studied in our case study. The power consumption of `gedit` versions in both test cases has been changing over versions with decreasing trend. We also applied linear regression modeling on the system

call invocations to predict `gedit` power consumption. The results show a high correlation between the invocation counts of some system calls related to memory management and file operations, and `gedit` power consumption over multiple versions.

The high correlations between each pair of system call invocation suggest the limited breadth of invoked system calls under the test cases. A broader range of test cases for `gedit` and other software systems are needed to produce a larger dataset.

This is the first study that tries to correlate power consumption of multiple software versions with system call invocations. Since system calls sit in the middle of user applications and the OS kernel. It is possible for us to model software power consumption over versions and also trace back to software in order to locate software changes that are responsible for power consumption variations using system calls. Therefore, our results lead to a promising direction for discovering the specific relationship between software change and software power consumption.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we have studied software energy consumption from two perspectives, user choice and software change. Very frequently users have a choice of what application to use to complete a task, and therefore we have investigated the impact of user choice on software energy consumption in the first half of the thesis. As the builders and maintainers of software, software developers are continuously making changes to software. Thus in the second half of this thesis, we have also uncovered the impact of software change on energy consumption.

For the impact of user choice on software energy consumption, we have proposed a user-centric method to measure the energy consumption of applications based upon the scenarios that correspond to tasks that users are expected to complete. We demonstrate its effectiveness by defining three such scenarios, and an implementation of benchmarks to measure the consumption during each of them. The results indicated that different applications can have dramatically different energy consumption when performing the same task (e.g. a command-line music player uses more than five times less energy than a GUI one). We also found that web-based applications tend to consume more energy than non-web based, and that idle applications can incur a significant amount of ghost energy consumption.

In terms of the impact of software change on energy consumption, we have proposed a method to investigate the relationship between system call invocations and software power consumption over versions. We implemented our method on

multiple versions of the text editor, `geidt`. Two test cases about text editing and syntax highlighting have been studied in our case study. The power consumption of `gedit` versions in both test cases has been changing over versions with decreasing trend. We also applied linear regression modeling on the system call invocations to predict `gedit` power consumption. The results show a high correlation between the invocation counts of some system calls related to memory management and file operations, and `gedit` power consumption over multiple versions.

In summary, we have investigated multiple use-case scenarios demonstrating that applications can consume energy differently for the same task thus illustrating the tradeoffs that end-users can make for the sake of energy consumption. We also studied the impact of software change on power consumption, which shows that system call invocations have a high correlation with the `gedit` power consumption over multiple versions.

5.2 Future Work

Unfortunately it is not trivial for users to know which applications are more energy efficient. We expect future work to be directed towards the creation of benchmarks and reporting mechanisms (similar to Energy Star) that inform developers and users of the energy efficiency of their applications. This will likely generate pressure on the developers to improve the energy efficiency of the applications they develop.

In addition, this is the first study that tries to correlate power consumption of multiple software versions with system call invocations. Since system calls sit in the middle of user applications and the OS kernel. It is possible for us to model software power consumption over versions and also trace back to software in order to locate software changes that are responsible for power consumption variations using system calls. Limited by the tested application and test cases in the thesis, more test cases and more test cases for various applications over versions are definitely needed for obtaining the hidden relationship between all kinds of system calls and software power consumption.

Bibliography

[1] Nadine Amsel and Bill Tomlinson. Green Tracker: A Tool for Estimating the Energy Consumption of Software. In *Proceedings, CHI EA*, pages 3337–3342, New York, NY, USA, 2010. ACM.

[2] Apple Inc. Apple Press Info. `http://www.apple.com/pr/library/2013/06/10Apple-Unveils-iOS-7.html`, 2013.

[3] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun 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*, IMC '09, pages 280–293, New York, NY, USA, 2009. ACM.

[4] Aaron Carroll and Gernot Heiser. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[5] Marshini Chetty, A.J. Bernheim Brush, Brian R. Meyers, and Paul Johns. It's Not Easy Being Green: Understanding Home Computer Power Management. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1033–1042, New York, NY, USA, 2009. ACM.

[6] Mian Dong, Yung-Seok Kevin Choi, and Lin Zhong. Power Modeling of Graphical User Interfaces on OLED Displays. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 652–657, New York, NY, USA, 2009. ACM.

[7] Mian Dong and Lin Zhong. Self-Constructive, High-Rate Energy Modeling for Battery-Powered Mobile Systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.

[8] Jason Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[9] Paul M. Greenawalt. Modeling Power Management for Hard Disks. In *MASCOTS '94., Proceedings of the Second International Workshop on*, pages 62 –66, Jan 1994.

[10] Ashish Gupta, Thomas Zimmermann, Christian Bird, Nachippan Naggapan, Thirumalesh Bhat, and Syed Emran. Detecting Energy Patterns in Software

Development . Technical Report MSR-TR-2011-106, Microsoft Research, 2011.

[11] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, Mahmut Kandemir, Tao Li, and Lizy Kurian John. Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 141–, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating Mobile Application Energy Consumption using Program Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.

[13] Shuai Hao, Ding Li, William G.J. Halfond, and Ramesh Govindan. Estimating Android Applications' CPU Energy Usage via Bytecode Profiling. In *First International Workshop on Green and Sustainable Software (GREENS), in conjunction with ICSE 2012*, June 2012.

[14] Ahmed E. Hassan. The Road Ahead for Mining Software Repositories. In *Proceedings of the Future of Software Maintenance (FoSM) at the 24th IEEE International Conference on Software Maintenance*, pages 48–57, 2008.

[15] Abram Hindle. Green Mining: A Methodology of Relating Software Change to Power Consumption. In *MSR*, pages 78–87, 2012.

[16] Abram Hindle. Green Mining: Investigating Power Consumption across Versions. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1301–1304, Piscataway, NJ, USA, 2012. IEEE Press.

[17] Intel Corporation. Developing Green Software. `http://software.intel.com/en-us/articles/developing-green-software`, June 2011.

[18] Jon Destouche. BlackBerry is Jammin Hope You Like Jammin To. `http://www.pocketberry.com/2013/05/14/blackberry-is-jammin-hope-you-like-jammin-to/`, 2013.

[19] Russ Joseph and Margaret Martonosi. Run-Time Power Estimation in High Performance Microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, ISLPED '01, pages 135–140, New York, NY, USA, 2001. ACM.

[20] Michael Larabel. Ubuntu's Power Consumption Tested. `http://www.phoronix.com/scan.php?page=article&item=878`, Oct 2007.

[21] Emanuele Lattanzi, Andrea Acquaviva, and Alessandro Bogliolo. Run-Time Software Monitor of the Power Consumption of Wireless Network Interface Cards. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 3254 of *Lecture Notes in Computer Science*, pages 352–361. Springer Berlin / Heidelberg, 2004.

[22] Ross McLachlan and Stephen Brewster. Towards New Widgets to Reduce PC Power Consumption. In *Proceedings of the 2012 ACM annual conference extended abstracts on Human Factors in Computing Systems Extended Abstracts*, CHI EA '12, pages 2153–2158, New York, NY, USA, 2012. ACM.

[23] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering Eevelopers to Estimate App Energy Consumption. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[24] Nokia Corporation. Nokia Energy Profiler. `http://www.developer.nokia.com/Resources/Tools_and_downloads/Other/Nokia_Energy_Profiler`, 2009.

[25] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the Energy Spent inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[26] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-Grained Power Modeling for Smartphones using System Call Tracing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM.

[27] Paul Thurrott. Microsoft by (Some of) the Numbers. `http://winsupersite.com/windows/microsoft-some-numbers`, 2013.

[28] Ronald Lye. 6 Alternatives to the Mac App Store. `http://www.usingmac.com/2013/7/24/6-alternatives-to-the-mac-app-store`, 2013.

[29] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An Energy Consumption Framework for Distributed Java-Based Systems. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 421–424, New York, NY, USA, 2007. ACM.

[30] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 97–113, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[32] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[33] The Linux man-pages project. Linux Man Pages Online. `http://man7.org/linux/man-pages/`, 2013.

[34] Ubuntu Press Pack. Canonical & Ubuntu Fast Facts. http://assets.ubuntu.com/sites/ubuntu/latest/u/files/section/devices/Ubuntu-fast_facts.pdf, 2013.

[35] Graham Upton, Ian Cook, and Ian T. Cook. *A Dictionary of Statistics*. OUP Oxford, 2008.

[36] Victor H. Android's Google Play beats App Store with over 1 million apps, now officially largest. http://goo.gl/lvw2tT, 2013.

[37] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

Appendix A

A.1 System Calls Traced in Text Editing Tests

1. access
2. bind
3. brk
4. chdir
5. chmod
6. clock_getres
7. clock_gettime
8. clone
9. close
10. connect
11. dup2
12. eventfd2
13. execve
14. fcntl64
15. fstat64
16. fstatfs64
17. fsync
18. ftruncate
19. ftruncate64
20. futex
21. getcwd
22. getdents
23. getdents64
24. getegid32
25. geteuid32
26. getgid32
27. getpeername
28. getpgrp
29. getpid
30. getppid
31. getresgid32
32. getresuid32
33. getrlimit
34. getsockname
35. gettimeofday
36. getuid32
37. inotify_add_watch
38. inotify_init1
39. inotify_rm_watch
40. ioctl
41. lgetxattr
42. listen
43. lstat64
44. madvise
45. mmap2

46. mprotect
47. munmap
48. open
49. openat
50. pipe
51. poll
52. prctl
53. read
54. readlink
55. recv
56. recvmsg
57. rename
58. rt_sigaction
59. rt_sigprocmask
60. send
61. sendmsg
62. set_robust_list
63. set_thread_area
64. set_tid_address
65. shmat
66. shmctl
67. shmget
68. sigreturn
69. socket
70. stat64
71. statfs64
72. tgkill
73. time
74. uname
75. unlink
76. waitpid
77. write
78. writev
79. _llseek

A.2 System Calls Traced in Syntax Highlighting Tests

1. access
2. bind
3. brk
4. chdir
5. chmod
6. clock_getres
7. clock_gettime
8. clone
9. close
10. connect
11. dup2
12. eventfd2
13. execve
14. fcntl64
15. fstat64
16. fstatfs64
17. fsync
18. ftruncate
19. futex
20. getcwd
21. getdents
22. getdents64
23. getegid32
24. geteuid32

25. getgid32
26. getpeername
27. getpgrp
28. getpid
29. getppid
30. getresgid32
31. getresuid32
32. getrlimit
33. getsockname
34. gettimeofday
35. getuid32
36. inotify_add_watch
37. inotify_init1
38. inotify_rm_watch
39. ioctl
40. lgetxattr
41. listen
42. lstat64
43. madvise
44. mmap2
45. mprotect
46. munmap
47. open
48. openat
49. pipe
50. poll
51. prctl
52. read
53. readlink
54. recv
55. recvmsg
56. rename
57. rt_sigaction
58. rt_sigprocmask
59. send
60. sendmsg
61. set_robust_list
62. set_thread_area
63. set_tid_address
64. shmat
65. shmctl
66. shmget
67. sigreturn
68. socket
69. stat64
70. statfs64
71. time
72. uname
73. unlink
74. waitpid
75. write
76. writev
77. _llseek