

RICE UNIVERSITY

Energy accounting and optimization for mobile systems

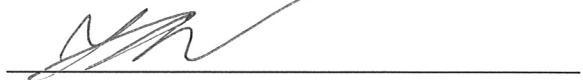
by

Mian Dong

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

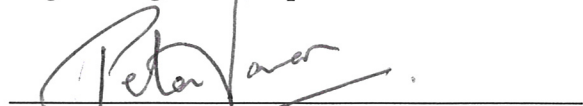
APPROVED, THESIS COMMITTEE:



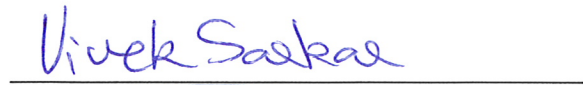
Lin Zhong, Chair
Associate Professor of Electrical and
Computer Engineering and Computer
Science



Joseph R. Cavallaro
Professor of Electrical and Computer
Engineering and Computer Science



Peter J. Varman
Professor of Electrical and Computer
Engineering and Computer Science



Vivek Sarkar
Professor of Electrical and Computer
Engineering and Computer Science

Houston, Texas

April, 2013

ABSTRACT

Energy accounting and optimization for mobile systems

by

Mian Dong

Energy accounting determines how much a software process contributes to the total system energy consumption. It is the foundation for evaluating software and has been widely used by operating system based energy management. While various energy accounting policies have been tried, there is no known way to evaluate them directly simply because it is hard to track every hardware use by software in a heterogeneous multi-core system like modern smartphones and tablets.

In this thesis, we provide the ground truth for energy accounting based on multi-player game theory and offer the first evaluation of existing energy accounting policies, revealing their important flaws. The proposed ground truth is based on Shapley value, a single value solution to multi-player games of which four axiomatic properties are natural and self-evident to energy accounting. To obtain the Shapley value-based ground truth, one only needs to know if a process is active during the time under question and the system energy consumption during the same time. We further provide a utility optimization formulation of energy management and show, surprisingly, that energy accounting does not matter for existing energy management solutions that control the energy use of a process by giving it an energy budget, or budget based energy management (BEM). We show an optimal energy management (OEM) framework can always outperform BEM. While OEM does not require any form of energy accounting, it is related to Shapley value in that both require

the system energy consumption for all possible combination of processes under question. We provide a novel system solution that meet this requirement by acquiring system energy consumption *in situ* for an OS scheduler period, i.e.,10 ms.

We report a prototype implementation of both Shapley value-based energy accounting and OEM based scheduling. Using this prototype and smartphone workload, we experimentally demonstrate how erroneous existing energy accounting policies can be, show that existing BEM solutions are unnecessarily complicated yet underperforming by 20% compared to OEM.

Acknowledgments

I would most like to thank my advisor Lin Zhong for his advice and guidance. He made invaluable contributions not just to this thesis, but also to my development as a research professional. He exhibited a constant optimism in the ultimate usefulness and success of my research that helped to sustain me as I worked through the hard problems. I am indebted to the other members of my thesis committee, Joseph Cavallaro, Peter Varman, and Vivek Sarkar, for helping me select interesting research directions to explore. Through many discussions with them, this thesis was improved in both content and presentation.

I have built upon the work of the many members of the RECG. Clayton Shepard conducted the LiveLab project and collected a large scale web browsing trace that was used by the Chameleon project. Zhen Wang contributed a lot in analyzing the web browsing trace and providing numerous discussions and comments on web browser design. Xiaozhu Lin, Hang Yu, Ardalan Amiri Sani, Robert LiKamWa, and Siqi Zhao also provided many useful comments on earlier drafts of some material that is reflected in this work.

Finally, I would like to especially thank my family for giving me continuous love, care, and support. I could not have done it without them.

Contents

Abstract	ii
Acknowledgments	iv
List of Illustrations	xi
List of Tables	xvii
1 Introduction	1
2 System Energy Accounting: A Game Theory Perspective	6
2.1 Energy Accounting	6
2.1.1 Problem Definition and Notations	6
2.1.2 Existing Energy Accounting Policies	7
2.2 Shapley Value as Ground Truth	9
2.2.1 Shapley Value Primer	9
2.2.2 Shapley Value based Energy Accounting	11
2.2.3 Existing Policies Against Shapley Value	12
2.3 Summary	14
3 Energy Management as Utility Optimization	15
3.1 Theoretical Foundation	15
3.2 Budget based Energy Management (BEM)	17
3.2.1 Problem Formulation	17
3.2.2 Performance Analysis	20
3.3 Optimal Energy Management (OEM)	23
3.3.1 Problem Formulation	23

3.3.2	Performance Analysis	23
3.4	Remarks	29
3.5	Summary	30
4	From Theory to Practice: The Key is <i>In Situ</i> $E(\mathcal{S})$ Estimation	32
4.1	Challenges and Solutions	32
4.2	System Energy Modeling Review	34
4.2.1	System Energy Model Primer	34
4.2.2	Prior Work on System Energy Modeling	35
4.2.3	Limitations of Prior Work	37
4.3	Battery Interface Characterization	39
4.3.1	Smart Battery Interface Primer	39
4.3.2	Experimental Setup	40
4.3.3	Technical Observations	40
4.4	Summary	45
5	<i>In Situ</i> $E(\mathcal{S})$ Estimation using Existing Battery Interface	46
5.1	Design of Sesame	46
5.1.1	Design Goals and Architecture	46
5.1.2	Data Collector	47
5.1.3	Model Constructor	51
5.1.4	Model Manager for Adaptability	56
5.2	Implementation of Sesame	57
5.2.1	Data Collector	57
5.2.2	Model Constructor	58
5.3	Evaluation	59
5.3.1	Systems and Benchmarks	59
5.3.2	Overhead	61
5.3.3	Accuracy and Rate	63

5.3.4	Model Convergence and Adaptation	65
5.3.5	Field Study	66
5.4	Summary	69
6	<i>In Situ</i> $E(\mathbb{S})$ Estimation using Improved Battery Interface	71
6.1	Improved Battery Interface	71
6.1.1	Architecture	71
6.1.2	Measurement Requirement	72
6.1.3	Synchronization	76
6.2	Estimation based on Partial Information	77
6.2.1	Recursive Definition	78
6.2.2	Linear Estimation	79
6.3	Implementation	80
6.4	Evaluation	81
6.4.1	Accuracy of $E(\mathbb{S})$	82
6.4.2	Evaluating Existing Policies	84
6.4.3	BEM vs. OEM	85
6.4.4	Practical Limitations of OEM	87
6.5	Summary	89
7	Power Modeling and Optimization for OLED Displays	90
7.1	Overview	90
7.2	Background and Related Work	93
7.2.1	Display System	93
7.2.2	OLED Display	95
7.2.3	Color Space	96
7.2.4	Related Work	97
7.3	Experimental Setup	98
7.3.1	Subjects	98

7.3.2	Benchmarks	99
7.3.3	OLED Display	99
7.3.4	Power Measurement	99
7.4	Pixel-Level Power Model	100
7.4.1	Power Model of OLED Module	101
7.4.2	Power Model of RGB Component	101
7.4.3	Estimation vs. Measurement	102
7.5	Image-Level Power Model	103
7.5.1	Color Histogram-based Method	104
7.5.2	Sampling-based Method	105
7.6	Code-Level Power Model	111
7.6.1	Power by GUI Object	112
7.6.2	Power by Composition of Objects	113
7.6.3	Experiment Results	114
7.7	Structured Transformation	115
7.7.1	Background/Foreground Transformation	116
7.7.2	Theme-based Transformation	116
7.8	Unstructured Transformation	117
7.8.1	Color Counting	118
7.8.2	Color Mapping	118
7.8.3	Evaluation	122
7.8.4	Discussion	125
7.9	Summary	127
8	A Color-Adaptive Web Browser for Mobile OLED Displays	129
8.1	Overview	129
8.2	Background and Related Work	132
8.2.1	Web Browser	132

8.2.2	Related Work	134
8.3	Motivational Studies	135
8.3.1	OLED Display Power	135
8.3.2	Web Usage by Smartphone Users	137
8.3.3	User Preference of Color Transformation	142
8.4	Design of Chameleon	145
8.4.1	Key Design Decisions	145
8.4.2	Architecture	146
8.4.3	Color Power Model Construction	149
8.4.4	Color Contribution Collection	149
8.4.5	Color Mapping Optimization	155
8.4.6	Color Transformation Execution	156
8.5	Implementation of Chameleon	156
8.5.1	Automatic Power Model Construction	158
8.5.2	Color Contribution Collection	159
8.5.3	Color Mapping Optimization	160
8.5.4	Color Transformation Execution	161
8.6	Evaluation	162
8.6.1	Experimental Setup	162
8.6.2	Power Reduction	162
8.6.3	Overhead	164
8.6.4	Field Trials	166
8.7	Discussion	169
8.8	Summary	170
9	Related Work	172
9.1	Energy Accounting Policies	172
9.2	Energy Modeling Methods	174

9.3	Energy Management Approaches	175
10	Conclusion	177
10.1	Contributions	177
10.1.1	Theoretical Framework	177
10.1.2	<i>In Situ</i> $E(S)$ Estimation	178
10.1.3	OLED Display Power Optimization	179
10.2	Suggestions for Future Work	179
	Bibliography	182

Illustrations

3.1	Illustration of 4 applications and their executions under BEM policy is divided into 4 epochs.	19
3.2	Algorithm for solving Problem BEM (budget-based energy management) defined in Section 3.2.	21
3.3	Algorithm for solving Problem OEM (optimal energy management) defined in Section 3.3.	24
4.1	Linear system energy models based on CPU utilization with (left) different hardware configurations and (right) different usage.	38
4.2	Current readings from accurate external measurement (DAQ) and battery interfaces of Nokia N85 (top) and Lenovo ThinkPad T61 (bottom).	42
4.3	Error vs. rate for battery interface readings. Dashed lines are calculated by comparing current readings at the highest rate against measurement.	44
5.1	Architecture of Sesame with three major components: data collector, model constructor and model manager.	48
5.2	Illustration of the total-least-squares method using linear regression with a single predictor x as example. The total-least-squares method minimizes the total squares $(\sum_j d_j^2)$ instead of $(\sum_j r_j^2)$ as regular regression does.	50
5.3	Illustration of model molding. Low-rate data points are generated by averaging high-rate data points (left); linear models generated by two sets of data points are identical (right).	53

5.4	Estimation errors for T61 at different rates and with different selections of predictors through PCA-based predictor transformation.	54
5.5	Overhead of Sesame in terms of execution time. Predictor collection is most expensive among the three steps. PCA and the bundled system call (syscall) help reduce the overhead of predictor collection significantly. . . .	62
5.6	Model molding improves accuracy and rate. Sesame is able to increase the model rate to 100 Hz.	64
5.7	Model adaptation to (top) system configuration change and (bottom) usage change. Error threshold is 10%.	65
5.8	Errors of the models generated in the field study evolve in the first five days. Errors are calculated relative to the averaged readings from battery interfaces in every 100 seconds in laptops and in every 1000 seconds in N900s.	69
5.9	Comparison with readings from battery interfaces. Errors are calculated relative to accurate external measurement.	70
6.1	Architecture of the <i>in situ</i> high-speed, high-accuracy power measurement solution: The battery interface measures power periodically and stores the measurement data with timestamps in its local memory; the mobile system also maintains a timestamped data structure of application combination executed in the OS kernel, utilizes a driver to read the energy measurement data structure from the local memory in the battery interface, and combines the two data structures to get all the $E(\mathbb{S})$	73
6.2	Smartphone workload power characterization for sampling rate: 200 Hz sampling rate is able to cover 99% of the signal information of the power trace (left); 200 Hz sampling rate introduces 1% error in energy measurement of each 10 ms (right).	75

6.3	Smartphone workload power characterization for resolution: dynamic range of power trace is around 100 dB and requires a 16-bit ADC for measurement (left); using a 8-bit ADC introduces 0.1% error in energy measurement of each 10 ms (right).	76
6.4	Data transfer delay calibration.	78
6.5	Estimation from partial information. Columns are filled by recursive definition and rows are filled by linear estimation.	79
6.6	Prototype: a TI PandaBoard ES and a MAXIM DS2756 battery fuel gauge evaluation module.	81
6.7	Error in obtaining $E(\mathbb{S}, \sigma)$: Over 90% of energy measurement by the battery interface are with error less than 5% and synchronization does improve accuracy (left); estimation has a median error of 10% using one third of total $E(\mathbb{S}, \sigma)$ (right).	83
6.8	Footprint of all the missing $E(\mathbb{S}, \sigma)$ (grey cells) and obtained $E(\mathbb{S}, \sigma)$ from measurement (white cells) in all three scenarios.	84
6.9	Energy accounting results for Shapley value (S) and Policy I-V. The energy consumption (Y axis) is normalized by the total system energy consumption.	85
6.10	Optimal utility values achieved by OEM and BEM with various energy accounting policies. All numbers are first normalized by that achieved by OEM for the same scenario.	87
6.11	Execution time of the optimization solver increases along with the number of variables increases. A large number of CPU cores make the scalability even more challenging.	88
7.1	Display system in a typical mobile system. The tree power models proposed require input of different abstractions and provide different accuracy-efficiency tradeoffs.	94

7.2	Measurement setup of the QVGA μ OLED module (left) and the Nokia N85 (right) used in our experimental validation.	100
7.3	Intensity Level vs. Power Consumption for the R, G, and B components of an OLED pixel: (a) μ OLED, sRGB; (b) μ OLED, linear RGB; (c) N85, sRGB; (d) N85, linear RGB.	103
7.4	Histogram of percent errors observed for the 300 benchmark images using our pixel-level power model.	104
7.5	Tradeoff between computation time and error of the color histogram-based method.	106
7.6	Comparison of four different sampling methods in image-based power models.	110
7.7	Comparison of three power models.	115
7.8	Power consumption and user review scores of GUIs with different themes.	117
7.9	Unstructured transformed GUIs with different settings.	124
7.10	Power reduction vs. relative review score.	125
7.11	Power reduction and computation time with various color level.	126
8.1	A simplified workflow of a web browser.	134
8.2	Pixel power models of three OLED displays. The power consumption by each color component of a pixel is a linear function of its corresponding linear RGB value.	136
8.3	OLED color power model in CIELAB. Given the lightness, or L^* , power consumption can differ as much as 5X between two colors with different chromaticity.	137

- 8.4 Web usage by smartphone users extracted from the LiveLab traces. The most visited websites of each user account for a high percentage of his web usage (top); a website uses a very small number of colors, about 1500 colors on average and 6500 at most, compared to all the available colors (10^7) (bottom). 139
- 8.5 To apply a user defined CSS will lose color information of different CSS boxes (b); Inversion with images makes them unusable (c); Inversion without the background images in two buttons makes them undistinguishable from inverted texts (e); Inversion without the foreground logo image makes it clash with the inverted background color (e). 140
- 8.6 Original ESPN homepage and its four color transformations generated by four algorithms used in the user study. 142
- 8.7 User study results. User favorite algorithm (top); average score of each website by User 1 (bottom). 144
- 8.8 Architecture of Chameleon and how it interacts with the browser engine. . . 148
- 8.9 Box plots for inter-series interval (left) and series duration (right) calculated using different timeout values. The bottom and top edges of a box are the 25th and 75th percentile; the white band in a box is the median, and the ends of the whiskers are maximum and minimum, respectively. . . . 152
- 8.10 How Chameleon counts pixels. (1) Copy the pixels from the frame buffer to the main memory; (2) Count the pixels in the main memory and save the results to a small Hash Table; (3) Merge the colliding entry of the small Hash Table to the per-website Hash Table; (4) Merge the entire small Hash table to the per-website Hash Table when all the pixels have been examined. 153
- 8.11 Display power reduction by training Chameleon with different weeks of data from the LiveLab traces. 163
- 8.12 Worst case time overhead of Chameleon. 165

8.13 Average percentage of usage time of color maps generated by each
algorithm by a participant. 168

8.14 Evolution of average number of color map switches per website by each
participant along time. 169

Tables

2.1	Notations	7
2.2	Axioms Violated by Existing Energy Accounting Policies	12
4.1	Battery interfaces of popular mobile systems	41
5.1	Specifications of platforms used in Sesame evaluation	60
5.2	Benchmarks used in Sesame evaluation	61
5.3	Specifications of laptops used in field study	67
6.1	Smartphone Specifications	74
6.2	Sufficient sampling rate for various smartphone applications(Unit:Hz) . . .	74

Chapter 1

Introduction

The ability to account system resource usage by software is the key to the design and optimization of computers. While modern computers can account CPU and memory usage by software easily, they can not yet do so for energy, an increasingly important system resource due to electricity and thermal concerns. Given the system energy consumption E and the set of processes $\mathbb{N} = \{1, 2, \dots, n\}$ that are active during time T , *energy accounting* is to determine the energy contribution by each process in \mathbb{N} . Energy accounting is the foundation for evaluating software by their energy use, e.g., [1, 2], and for operating system (OS) based energy management that achieves certain trade-off between energy consumption and process utility by controlling how processes are scheduled, e.g., [3, 4].

This thesis provides the first formal treatment of energy accounting and answer two fundamental questions about it for the first time: (i) *how to evaluate an energy accounting policy?* and (ii) *is energy accounting necessary at all for energy management?* Energy accounting has been a long-standing, hard problem for multiprocessing systems where multiple processes can be active at the same time. In the past decade, many energy accounting policies have been tried for various purposes as exemplified recently by [5, 1, 2, 6], including the “Battery” feature in recent versions of Android. However, there is no known *ground truth* against which one could evaluate a policy directly. The fundamental reason is that *it is practically infeasible to track how software uses each every hardware component in a heterogeneous multicore system* like modern smartphones and tablets. In most cases, only the total system energy consumption can be measured, and the usage of a few hardware

components, e.g., CPU and memory, can be attributed to a process. As a result, energy accounting policies are usually evaluated for limited corner cases against derivative properties, e.g., by how accurately a policy can predict the total system energy consumption when a single process is running.

Toward answering the question “how to evaluate an energy accounting policy?” we provide the ground truth for energy accounting based on multi-player game theory and offer the first evaluation of existing energy accounting policies, revealing their important flaws. Our key insight is that energy accounting can be formulated as a game theory problem: when multiple players participate in a game and the game produces a surplus, how to divide the surplus among the players? In energy accounting, the game is the system during T , players processes, and energy consumption the surplus. We show Shapley value [7], a single value solution to this problem, provides a natural, unique way to distribute the system energy consumption to the processes. We show how existing energy policies violate the four self-evident axioms required by Shapley value. This is discussed in Chapter 2.

Toward answering the question “is energy accounting necessary for energy management?” we formulate energy management as a utility optimization problem and show, surprisingly, that energy accounting does not matter for widely used budget based energy management (BEM) framework [3, 4, 8]. In the BEM framework, OS first assigns each running process an energy budget and then charges each process from its budget based on energy accounting. A process will not be scheduled if it runs out of its budget. We further prove that this surprising result is due to the fact that BEM is sub-optimal and subsequently provide an optimal energy management (OEM) framework. This is discussed in Chapter 3.

While OEM does not require any form of energy accounting, it is related to Shapley value in an interesting way: both OEM and Shapley value require the system energy consumption for all possible combinations of processes under question, i.e., $E(\mathbb{S}) \forall \mathbb{S} \subseteq \mathbb{N}$.

Acquiring such data, however, is extremely challenging in practice because (i) the number of the process combinations can be large ($2^{|\mathbb{N}|}$); (ii) $E(\mathbb{S})$ depends on the process execution dynamics and hardware states. In order to address these challenges, we provide a novel system solution to acquire system energy consumption *in situ* for very short time intervals, down to the OS scheduler period, i.e., 10 ms. To realize such *in situ* $E(\mathbb{S})$ estimation in 10 ms is very technically challenging in mobile system. As we experimentally demonstrated, state-of-the-art system energy modeling and existing smart battery interface, two potentially possible methods of *in situ* $E(\mathbb{S})$ estimation, are both incapable of providing an accurate power estimation at each 10 ms. This is discussed in Chapter 4.

To tackle this challenge, we propose a self-modeling paradigm, Sesame, in which a mobile system automatically generates its system energy model by itself. Our solution, Sesame, leverages the possibility of self power measurement through the smart battery interface and employs a suite of novel techniques to achieve accuracy and rate much higher than that of the smart battery interface. We report the implementation and evaluation of Sesame on a laptop and a smartphone. The experiment results show that Sesame is able to generate system energy models of 88% accuracy at one estimation per 10 ms. This is discussed in Chapter 5.

We further advance Sesame by providing a mobile system solution of *in situ* $E(\mathbb{S})$ estimation based on an improved battery interface that is capable of power measurement with an accuracy over 95% at one sample per 5 ms. Together with our enhancement to the Shapley value theory to work with a partially defined multi-player game, we finally realize Shapley value based energy accounting and OEM based scheduling on mobile systems. We report a prototype implementation based on Google Android, Texas Instruments PandaBoard ES with OMAP4460, and MAXIM DS2756 battery fuel gauge. Using this prototype and smartphone workload, we experimentally demonstrate how erroneous exist-

ing energy accounting policies can be, how little difference in energy management can they make, and show that OEM outperforms BEM in scheduling by 20%. This is discussed in Chapter 6.

Emerging organic light-emitting diode (OLED) based displays obviate external lighting, and consume drastically different power when displaying different colors, due to their emissive nature. Such huge power difference poses a huge challenge on $E(\mathbb{S})$ estimation and thus energy accounting and management. To address this challenge, we consider OLED display as a separate system from the whole mobile system and apply special power optimization on the OLED display. We first conduct a comprehensive treatment of power modeling of OLED displays, providing models that estimate power consumption based on pixel, image, and code, respectively. Then, based on the models, we propose techniques that adapt GUIs based on existing mechanisms as well as arbitrarily under usability constraints. Finally, based on the power modeling and optimization techniques, we develop Chameleon, a color adaptive web browser that renders webpages with power-optimized color schemes under user-supplied constraints. According to measurements with OLED smartphones, Chameleon is able to reduce average system power consumption for web browsing by 40 percent and is able to reduce display power consumption by 64 percent without introducing any noticeable delay. OLED display power modeling and optimization is discussed in Chapter 7 while design and evaluation of Chameleon is discussed in Chapter 8.

Our results mandate a rethinking of energy accounting and its application in energy management. Not only can existing energy accounting policies be off-the-mark by a large degree (now we know why many Android users feel the “Battery” feature is inaccurate), but also existing budget based energy management may be unnecessarily complicated yet underperforming. While our prototype and evaluation are based on battery-powered mobile

systems, most of our results are directly applicable to any multiprocessing computer system where energy consumption is a system concern and can be measured for the whole system but not for each every hardware component.

Chapter 2

System Energy Accounting: A Game Theory Perspective

In this chapter we seek to answer a long-standing question about system energy accounting with theory evaluation. That is, how to evaluate an energy accounting policy? Our answer to the question is: Shapley value based energy accounting should serve as the ground truth for energy accounting.

2.1 Energy Accounting

We next discuss the energy accounting problem and related work.

2.1.1 Problem Definition and Notations

Energy accounting is to tell how much a process* contributes to the system energy consumption. Let T denote the time during which the system energy consumption E must be attributed; $\mathbb{N} = \{1, 2, \dots, n\}$ denote the processes that are active during this time. An energy accounting policy determines the contribution ϕ_i by process $i \in \mathbb{N}$ to the system energy consumption E . Mathematically a policy is a family of mapping $\{\phi_i(E(\mathbb{N})), \forall i \in \mathbb{N}\}$ from total energy consumption $E(\mathbb{N})$ to per-process energy consumptions ϕ_i . Note that T does not have to be a single continuous time interval; it can be a collection of time intervals.

*We use *process* to denote the elementary software principal of system resource usage in this work. One may well use other terms such as *thread* and *task*. A program or operating system may consist of multiple *processes*.

A self-evident property that an energy accounting policy must have is: $\sum_i \phi_i = E(\mathbb{N})$. That is, the energy contributions by all processes must add up to be the same as the total system energy consumption. We call this property *Efficiency* according to [7].

Table 2.1 summarizes the main notations used in this thesis.

Table 2.1 : Notations

Symbol	Description
T	Time in which system energy consumption is under question
\mathbb{N}	A set of n processes that are active during T
m	Max number of concurrent processes allowed in the system
\mathbb{S}	A subset of \mathbb{N} ; a <i>coalition</i> of processes
$E(\mathbb{S})$	System energy consumption in T if only \mathbb{S} are active
ϕ_i	Contribution by process i to $E(\mathbb{N})$
σ	System state that is out of the control of the processes
$E(\mathbb{S}, \sigma)$	System energy consumption when \mathbb{S} are active with state σ

2.1.2 Existing Energy Accounting Policies

We next discuss the major types of energy accounting policies reported in the literature.

Policy I: $\phi_i = E(\mathbb{N})/|\mathbb{N}|$. This policy states that the energy contributed by process i is equal to the total energy consumption divided by the number of processes. That is, each process gets an equal share of the total energy consumption.

Policy II: $\phi_i = E(\{i\})$. This policy states that the energy contributed by process i is the same as its stand-alone energy consumption $E(\{i\})$ when only process i is running in the system. WattsOn [2] utilizes such policy to obtain the energy consumption of each mobile app to enable third-party app developers to improve energy efficiency of their products.

Policy III: $\phi_i = E(\mathbb{N}) - E(\mathbb{N} \setminus \{i\})$. This policy states that the energy contributed by process i is equal to its marginal energy contribution, which is the difference between system energy consumption when process i is running and when it is not running, while all other conditions remain unchanged [9].

Policy IV: The energy contributed by process i is equal to the sum of system energy consumption over all scheduling intervals when process i is scheduled in CPU. PowerScope [10] used this policy to account energy consumption of each software running in a system. Obviously, this approach will fail to work when more than one process can be running at the same time.

Policy V: This sophisticated policy relies on a system energy model $E = f(x_1, x_2, \dots, x_p)$ where $x_k, k = 1, 2, \dots, p$, are predictors that can be obtained in software such as CPU and memory usage. The system figures out how much process i contributes to each of the predictors, $x_{k,i}$ and determines the energy contribution by i by plugging $x_{k,i}$ back into f , i.e., $\phi_i = f(x_{1,i}, x_{2,i}, \dots, x_{k,i}, \dots, x_{p,i})$. To have the Efficiency property described above, however, f must be a linear function, i.e., $E = \beta_0 + \sum_{k=1}^p \beta_k \cdot x_k$ and there must be a heuristic to split the constant β_0 among the processes. Variants of Policy V have been used by Android [11], PowerTutor [12] and AppScope [6] for smartphones, by ECOSysem [3] for laptops and by Joulemeter [13] for servers. Policy V, however, is fundamentally limited because how it determines the energy contribution by a process severely depends on how it estimates the system energy consumption. In particular, the estimation can only take into account resource usage that can be attributed to individual processes; and the estimation must employ

a linear model. These two limitations can be very problematic for modern mobile systems where many hardware components including some major energy consumers are invisible to the OS [14], e.g., system bus and graphics, and others whose usage cannot be easily attributed to a process, e.g., GPS and display. Moreover, it has been shown that linear models are inadequate for estimating system energy consumption [15, 16]. As a result, Policy V will not only significantly underestimate the energy contribution by a process and but also be unable to attribute a significant portion of system energy consumption to running processes. Due to the lack of ground truth for energy accounting, the problems of Policy V have never been quantified.

In the rest of the thesis, we will reveal the fundamental flaws of these energy accounting policies analytically (Section 2.2.3) and quantitatively (Section 6.4.2).

2.2 Shapley Value as Ground Truth

Existing work on energy accounting does not have a ground truth to directly evaluate an energy accounting policy. Rather, energy accounting policies are evaluated based on some derivative properties. The first key contribution of this thesis is to introduce Shapley value as the ground truth for energy accounting. We will discuss how the theory can be applied to energy accounting and demonstrate how existing energy accounting policies lack one or more of the four desirable properties that uniquely define Shapley value. In Chapter 6, we will provide a practical implementation for Shapley value based energy accounting.

2.2.1 Shapley Value Primer

We observe a problem similar to energy accounting has been extensively studied in game theory: how to determine the contribution of each individual player in a game with multiple players? Shapley value [7] is the mostly celebrated single value solution to such problem.

Consider a game where a set of n players, $\mathbb{N} = \{1, \dots, n\}$, collectively generate a grand surplus $v(\mathbb{N})$. Let $v : 2^{\mathbb{N}} \rightarrow \mathbb{R}$ be a *characteristic function* that describes the payoff $v(\mathbb{S})$ a subset of players $\mathbb{S} \subseteq \mathbb{N}$ can gain by forming a coalition. A powerful result proven by Shapley in 1953 [7] defines Shapley value $\phi_i(v)$ for $i = 1, \dots, n$ as the *unique* way to distribute the grand surplus $v(\mathbb{N})$ among the n players that satisfies four simple axioms:

Efficiency: The sum of share of all players is the grand surplus, i.e. $\sum_i \phi_i(v) = v(\mathbb{N})$. This is the Efficiency property mentioned in Section 2.1.2.

Symmetry: Symmetric players receive equal shares, i.e., if $v(\mathbb{S} \cup \{i\}) = v(\mathbb{S} \cup \{j\}) \forall \mathbb{S} \subseteq (\mathbb{N} \setminus \{i, j\})$, then $\phi_i(v) = \phi_j(v)$. That is, if replacing player i with player j in any coalition does not change the game surplus and vice versa, the shares of the two players should be identical.

Null Player: Player i receives zero share if he does not change the payoff in any coalition \mathbb{S} , i.e., if $v(\mathbb{S} \cup \{i\}) = v(\mathbb{S}) \forall \mathbb{S} \subseteq \mathbb{N}$, then $\phi_i(v) = 0$. That is, if adding a player to any coalition does not change the game surplus, the player should receive zero.

Additivity: The share to player i in a sum of two games is the sum of the allocations to the player in each individual game, i.e., $\phi_i(v) + \phi_i(w) = \phi_i(v + w)$, $\forall i$. That is, if we view two games as a single game, the share a player receives from the combined game should be the same as the sum of what he would receive from each of the two original games.

Shapley value is the unique distribution function that satisfies the above four axioms:

$$\phi_i(v) = \sum_{\mathbb{S} \subseteq \mathbb{N} \setminus \{i\}} \frac{v(\mathbb{S} \cup \{i\}) - v(\mathbb{S})}{(|\mathbb{N}| - |\mathbb{S}|) \binom{|\mathbb{N}|}{|\mathbb{S}|}}, \quad (2.1)$$

where $|\mathbb{S}|$ is the cardinality of \mathbb{S} , i.e., the number of members of \mathbb{S} ; similarly, $|\mathbb{N}|$ is the number of members of \mathbb{N} , or n ; $\binom{|\mathbb{N}|}{|\mathbb{S}|}$ is $|\mathbb{N}|$ -choose- $|\mathbb{S}|$. Shapley value captures the ‘‘average marginal contribution’’ of player i , averaging over all the different sequences according to

which the grand coalition could be built up from the empty coalition.

To calculate Shapley value $\phi_i(v)$, one need not only the grand surplus when all n players play, $v(\mathbb{N})$, but also the surplus when any subset of the n players play, or $v(\mathbb{S})$ for all $\mathbb{S} \subseteq \mathbb{N}$. Shapley value has been widely applied for solving benefit/cost sharing problems in diverse fields, including computer science, e.g., [17, 18].

2.2.2 Shapley Value based Energy Accounting

We argue Shapley value provides a natural and intuitive ground truth for energy accounting. We treat the time interval T in which the system under question consumes energy as the game; the processes active in the same time interval \mathbb{N} the players; and total system energy consumption, $E(\mathbb{N})$, the grand surplus to distribute. Thus, the energy contribution ϕ_i by process i is given by the Shapley value $\phi_i(v)$ of the game with $v(\mathbb{S}) = E(\mathbb{S}) \forall \mathbb{S} \subseteq \mathbb{N}$ in Equation (2.1). Note that to calculate the Shapley value to distribute $E(\mathbb{N})$ to all processes in \mathbb{N} , one must know the system energy consumption in the same time interval T if only a subset of \mathbb{N} are active during T , i.e., $E(\mathbb{S}) \forall \mathbb{S} \subseteq \mathbb{N}$. This poses a significant challenge to the practical use of Shapley value and will be the focus of Chapter 6.

Policy Shapley: $\phi_i = \phi_i(E)$ which is the Shapley value of the game described above.

Since Shapley value is uniquely defined given the four axioms, we show that these four axioms are self-evident and logical for energy accounting. (i) *Efficiency* requires the sum of the energy contributions by all processes be equal to the total system energy consumption. This is the Efficiency property we already discussed in Section 2.1.2. (ii) *Symmetry* requires that if replacing one process with the other under any circumstances does not change the system energy consumption, the energy contributions by two processes should be identical. (iii) *Null Player* requires that if adding a process under any circumstances will not change

the system energy consumption, the energy contribution by this process should be zero. Finally, (iv) *Additivity* says that if we break T into multiple time intervals and apply the same accounting policy to them, the energy attributed to a process in T should be equal to the sum of energy attributed to it in these shorter time intervals.

We argue that the four axioms of Shapley value are both self-evident and desirable for energy accounting for its applications in software evaluation and energy management.

2.2.3 Existing Policies Against Shapley Value

We next reveal the flaws of the five types of existing policies against the four axioms that uniquely define Shapley value. Table 2.2 summarizes how they violate the four axioms. In Section 6.4, we will offer quantitative evidence regarding their flaws.

Table 2.2 : Axioms Violated by Existing Energy Accounting Policies

Policies	Efficiency	Symmetry	Null Player	Additivity
I			×	
II	×		×	
III	×			×
IV		×	×	
V	×	×	×	

Policy I violates *Null Player* because a process always gets positive energy contribution according to it.

Policy II violates *Efficiency*. For example, consider a system with two processes. Ac-

According to Policy II, energy contributed by process 1, or process 2, is equal to its stand-alone energy consumption $E(\{1\})$, or $E(\{2\})$, respectively. However, to satisfy *Efficiency* requires $\phi_1 + \phi_2 = E(\{1\}) + E(\{2\}) = E(\{1,2\})$. Unfortunately, the last equality does not hold in many cases, e.g., when processes 1 and 2 share some system resources, one usually observes $E(\{1\}) + E(\{2\}) > E(\{1,2\})$.

Policy III violates *Efficiency*. Using the same example above, we now have $\phi_1 = E(\{1,2\}) - E(\{2\})$ and $\phi_2 = E(\{1,2\}) - E(\{1\})$. Thus, the sum of shares is $\phi_1 + \phi_2 = 2E(\{1,2\}) - E(\{1\}) - E(\{2\})$. Again, $E(\{1\}) + E(\{2\}) = E(\{1,2\})$ is necessary for *Efficiency* to hold. The LEA²P platform [19] utilizes an improved Policy III that normalizes all the $\phi_i = E(\mathbb{N}) - E(\mathbb{N} \setminus \{i\})$ to enforce *Efficiency*. However, the normalization will lead to violation of *Additivity* because different normalization factors may be used in separate time intervals.

Policy IV does not work for multiprocessing systems. To extend Policy IV for multiprocessing systems, one can attribute energy consumption to processes based on their CPU usage, as in [20]. Such extension, however, violates *Null Player* because a process always has positive CPU usage and thus positive energy contribution according to the policy. It further violates *Symmetry* due to its heavy dependency on timing of energy consumption. For instance, the same process may be charged different energy bills when it consumes energy through asynchronous I/O operations and suffers random delays, as mentioned by [3].

Policy V violates the *Efficiency*, *Symmetry*, and *Null Player*. First, the inaccuracy in its energy model as highlighted in Section 2.1.2 may lead to violation of *Efficiency*. Second, there may exist processes that make use of different hardware components, but make the same energy contributions in all coalitions. These processes are indistinguishable with respect to their energy behaviors and therefore should be charged equal energy bills according to *Symmetry*. However, if their predictor models and invisible components are constructed

differently, Policy V would dictate that these processes have different energy contributions only based on the predictors and thus violates *Symmetry*. Finally, a process always gets positive energy contribution because it requires CPU time and system resources that all contribute to positive predictors of energy models.

2.3 Summary

This chapter began with the problem definition of system energy accounting and reviewed existing energy accounting policies. Then the chapter proposed Shapley value as the ground truth and analytically showed how existing energy accounting policies violate the four self-evident axioms that define the Shapley value.

Chapter 3

Energy Management as Utility Optimization

In this chapter, we seek to answer another research question regarding energy accounting. That is, is energy accounting necessary at all for energy management? To answer this question, we provide a utility optimization framework for energy management, which enables innovative algorithms for optimal energy management under various fairness criteria and energy constraints. A somehow surprising result from our analysis shows that for the popular budget based energy management, the choice of energy accounting policy does not matter at all.

3.1 Theoretical Foundation

Operating system based energy management has widely utilized energy accounting of various sophistication. It seeks to control the energy use by each process and schedule their executions to maximize aggregate system utility under energy constraints. We first show energy management can be formulated as a utility optimization problem given the energy capacity by scheduling processes, i.e., by determining each process's run time.

Process Run Time: We consider n independent processes, denoted by $\mathbb{N} = \{1, 2, \dots, n\}$, and a system that is capable of running up to m processes concurrently during a scheduler period, i.e., a set of processes \mathbb{S} can be scheduled simultaneously only if $|\mathbb{S}| \leq m$. Suppose that during the time horizon of energy management, a coalition of concurrent processes \mathbb{S} receive total run time $T_{\mathbb{S}}$. Note that $T_{\mathbb{S}}$ does not have to be a single continuous time interval;

it can be a collection of time intervals. Then, the run time received by an individual process i is given by

$$t_i = \sum_{\mathbb{S}: i \in \mathbb{S}, |\mathbb{S}| \leq m} T_{\mathbb{S}}. \quad (3.1)$$

Here the summation is over all feasible process coalitions satisfying $|\mathbb{S}| \leq m$ and containing process i . We refer to $T_{\mathbb{S}}$ as the *scheduling decision* for coalition \mathbb{S} .

Energy Capacity: The total system energy consumption must be bounded by available energy capacity, C . The energy consumption of a set of processes \mathbb{S} during a scheduling period T_0 is denoted by $E(\mathbb{S})$, and its average power $E(\mathbb{S})/T_0$. Since $T_{\mathbb{S}}$ is the total time that \mathbb{S} is scheduled, our energy management is subject to the energy capacity constraint, i.e.,

$$\sum_{\mathbb{S}: |\mathbb{S}| \leq m} E(\mathbb{S}) \cdot T_{\mathbb{S}}/T_0 \leq C. \quad (3.2)$$

Utility Function: In our optimization framework, each process is assigned a utility function $U_i(t_i)$, which measures the utility of process i receiving run time t_i . Such a utility optimization has been widely used in wireless communications and networking [21]. We make the common assumption that $U_i(t_i)$ is monotonically increasing and concave [22].

Problem Formulation: Energy management is to make scheduling decisions, i.e., $T_{\mathbb{S}}$, in order to maximize the aggregate utility of all processes, i.e., $\sum_{i=1}^n U_i(t_i)$, under energy capacity constraint C .

One example of the utility function is the so-called α -fair utility function, defined as $U_i(x) = x^{(1-\alpha)}/1 - \alpha$ for $\alpha \geq 0$ and $\alpha \neq 1$, and $U_i(x) = \log(x)$ for $\alpha = 1$. By choosing different values of α , one can set different objective functions. For instance, when $\alpha = 0$, the aggregate utility becomes the aggregate run time of all the processes, i.e., $\sum_i t_i$. When $\alpha \rightarrow +\infty$, it reduces to $\min_i t_i$, which optimizes max-min fair allocations. When $\alpha = 1$, it becomes $\sum_i \log(t_i)$, which enforces a proportional fairness preventing processes from “starving” while improving the aggregate run time.

3.2 Budget based Energy Management (BEM)

We consider budget based energy management (BEM), a widely used energy management approach [3, 4, 8], where each process receives an independent energy budget and disburses it according to a process-specific policy. The use of per-process budget not only enables a single energy source to be shared by diverse applications in a fair and simple fashion, but also offers composability among different processes. For instance, in a smartphone, an energy budget can be reserved for voice calls to guarantee a minimal amount of talk time.

3.2.1 Problem Formulation

BEM relies on energy accounting to split total energy consumption among individual processes. Let $\phi_i(E(\mathbb{S}))$ denote the energy consumption attributed to process i according to the energy accounting policy during a scheduling period in which \mathbb{S} are scheduled to run and $i \in \mathbb{S}$. After this scheduling period, $\phi_i(E(\mathbb{S}))$ will be deducted from process i 's energy budget B_i . When a process runs out of budget, it will not be scheduled any more.

We assume that each process is scheduled with a known probability p_i , until its budget runs out. To determine process run time, we suppose that processes reach zero energy budget in the following order: $\pi(1), \pi(2), \dots, \pi(n)$, where $\pi(k)$ be the k th process that runs out of its budget (If multiple processes reach zero budget simultaneously, their ordering can be arbitrary). Then, the sequence of positive-budget processes $\mathbb{A}_k = \{\pi(k), \pi(k+1), \dots, \pi(n)\}$ for $k = 1, \dots, n$ form a contraction with $\mathbb{A}_{k+1} \subseteq \mathbb{A}_k$. We can partition the system execution into n distinct intervals, each of length T_k and containing a set of $|\mathbb{A}_k| = n - k + 1$ positive-budget processes.

Now we derive energy consumption and run time for BEM. Since scheduling periods in modern systems are sufficiently short, we assume that the system state out of the control of processes remains the same during a scheduling period. Further, we assume the time scale

of energy management is much larger than that of a scheduling period. That is, the energy budget for processes are determined for periods much longer than a scheduling period. The Law of Large Number applies and allows us to compute the approximated run time that each feasible subset of processes, $\mathbb{S} \subseteq \mathbb{A}_k$ and $|\mathbb{S}| \leq m$, receive during T_k :

$$\begin{aligned} T_{\mathbb{S},k} &= T_k \cdot \text{Prob}\{\mathbb{S} \text{ is selected}\} \\ &= \frac{T_k \Delta_{\mathbb{S},k}}{\sum_{\mathbb{S}:\mathbb{S} \subseteq \mathbb{A}_k, |\mathbb{S}| \leq m} \Delta_{\mathbb{S},k}} \end{aligned} \quad (3.3)$$

where $\Delta_{\mathbb{S},k} = \prod_{i \in \mathbb{S}} p_i \prod_{j \in \mathbb{A}_k \setminus \mathbb{S}} (1 - p_j)$ is the probability that only subset \mathbb{S} is selected without constraint $|\mathbb{S}| \leq m$. Therefore, each process i receives aggregate run time from all feasible coalitions with $i \in \mathbb{S}$ as follows:

$$T_{i,k} = \sum_{\mathbb{S}:\mathbb{S} \subseteq \mathbb{A}_k, |\mathbb{S}| \leq m, i \in \mathbb{S}} T_{\mathbb{S},k} = T_k \frac{\sum_{\mathbb{S}:\mathbb{S} \subseteq \mathbb{A}_k, |\mathbb{S}| \leq m, i \in \mathbb{S}} \Delta_{\mathbb{S},k}}{\sum_{\mathbb{S}:\mathbb{S} \subseteq \mathbb{A}_k, |\mathbb{S}| \leq m} \Delta_{\mathbb{S},k}} \quad (3.4)$$

We consider n independent processes, denoted by $\mathbb{N} = \{1, 2, \dots, n\}$. Under BEM policy, each process is scheduled to run until its energy budget reaches zero. To determine process run time, we suppose that processes deplete their energy budgets in the following order: $\pi(1), \pi(2), \dots, \pi(n)$, where $\pi(k)$ be the k th process that runs out of energy budget (If multiple processes reach zero budget simultaneously, then their ordering can be arbitrary). Then, the execution of all n processes can be divided into n epochs, T_1, T_2, \dots, T_n , where each epoch T_k represents the time interval between successive energy depletion of processes $\pi(k-1)$ and $\pi(k)$. It is easy to verify that the sequence of active processes $\mathbb{A}_k = \{\pi(k), \pi(k+1), \dots, \pi(n)\}$ within each epoch T_k form a contraction with $\mathbb{A}_{k+1} \subseteq \mathbb{A}_k$ for $k = 1, \dots, n$.

Figure. 3.1 illustrates our model for $n = 4$ processes. Given process $\pi(1) = \{1\}$ is the first to run out of budget, epoch T_1 consists of 4 active processes $\mathbb{A}_1 = \{1, 2, 3, 4\}$, while epoch T_2 consists of 3 active processes $\mathbb{A}_2 = \{2, 3, 4\}$. Similarly, $\pi(2) = \{4\}$ and

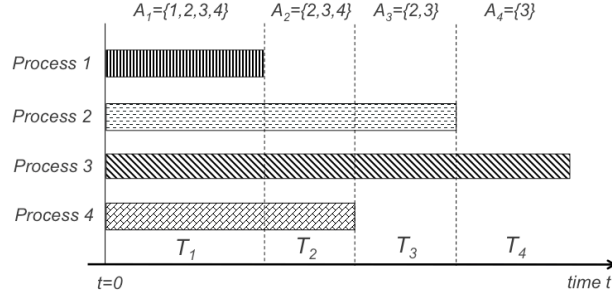


Figure 3.1 : Illustration of 4 applications and their executions under BEM policy is divided into 4 epochs.

$\pi(3) = \{2\}$ means that epoch T_3 contains 2 active processes $\mathbb{A}_3 = \{2,3\}$ and epoch T_4 contains a single process $\mathbb{A}_4 = \{3\}$.

Similarly, process i is charged an energy cost by aggregating its attributions from all feasible coalitions \mathbb{S} during T_k :

$$\Phi_i(\mathbb{A}_k, T_k) \triangleq \sum_{\mathbb{S}: \mathbb{S} \subseteq \mathbb{A}_k, |\mathbb{S}| \leq m} \phi_i(E(\mathbb{S})) \cdot T_{\mathbb{S},k} / T_0 \quad (3.5)$$

The objective of BEM is to maximize the aggregate utility by apportioning system energy budget among individual processes. We formulate it as the following optimization:

Problem BEM :

$$\text{maximize} \quad \sum_{i=1}^n U_i(t_i) \quad (3.6)$$

$$\text{subject to} \quad t_i = \sum_k T_{i,k}, \quad \forall i, \quad (3.7)$$

$$\sum_k \Phi_i(\mathbb{A}_k, T_k) \leq B_i, \quad \forall i \quad (3.8)$$

$$\mathbb{A}_k = \{\pi(k), \pi(k+1), \dots, \pi(n)\}, \quad \forall k \quad (3.9)$$

$$\sum_{i=1}^n B_i \leq C \quad (3.10)$$

$$\text{variables} \quad \pi, T_k, B_i. \quad (3.11)$$

where both $T_{i,k}$ and Φ_i are functions of T_k , as given in (3.4) and (3.5), respectively. The optimization is carried out over all ordering π of n processes. Note that π and T_k together determine process scheduling, i.e., $T_{\mathbb{S},k}$, $\forall k$ and $\forall \mathbb{S} \subseteq \mathbb{N}$, $|\mathbb{S}| \leq m$, as shown in (3.3).

3.2.2 Performance Analysis

We next analyze the performance of BEM, i.e., the maximal utility values that can be achieved by BEM. We prove a somehow counter-intuitive result, which shows that the optimal utility value of Problem BEM can be achieved independent of energy accounting policy ϕ_i as long as the Efficiency property in Section 2.2.1 is satisfied, i.e., $\sum_i \phi_i(E(\mathbb{S})) = E(\mathbb{S})$, $\forall \mathbb{S}$. We perform our analysis in two steps. First, we seek to find optimal solution of Problem BEM given an energy accounting policy. Second, we show that the optimal utility value can be achieved in BEM regardless of what energy accounting policy is used.

Optimal Solution of Problem BEM: We next present an algorithmic solution to Problem BEM. First, due to the Efficiency property, all system energy consumption is accurately accounted for in Problem BEM. Therefore, energy constraints (3.8) and (3.10) must both be met with exact equality at optimum, since it is non-optimal to have leftover energy go wasted. For any given ordering of processes π and resulting $\mathbb{A}_k = \{\pi(k), \dots, \pi(n)\}$, Problem BEM reduces to the following utility maximization with equality linear constraints:

$$\text{maximize} \quad \sum_{i=1}^n U_i(t_i) \quad (3.12)$$

$$\text{subject to} \quad t_i = \sum_k T_{i,k}, \quad \forall i, \quad (3.13)$$

$$\sum_k \sum_{\mathbb{S} \subseteq \mathbb{A}_k, |\mathbb{S}| \leq m} E(\mathbb{S}) T_{\mathbb{S},k} / T_0 = C, \quad (3.14)$$

$$\text{variables} \quad t_i, T_k. \quad (3.15)$$

where (3.14) is derived by replacing the inequality in (3.8) and (3.10) and combining

them using the Efficiency property. Again, both $T_{i,k}$ and $T_{\mathbb{S},k}$ are functions of T_k , as given in (3.4) and (3.5), respectively. When the utility function is concave, the optimization above can be easily solved by standard convex optimization algorithms, e.g., Newton method and interior point method [22]. After obtaining the optimal solution, optimal per-process budget B_i^* for each process i can be computed by $B_i = \sum_k \Phi_i(\mathbb{A}_k, T_k)$ from constraint (3.8). The algorithm for solving Problem BEM is summarized in Figure. 3.2 and its complexity is analyzed below.

BEM-opt

Initialize utility $U_{\text{best}} = -\infty$ and budget $B_{i,\text{best}} = 0$.

for each permutation π

Generate $\mathbb{A}_k = \{\pi(k), \pi(k+1), \dots, \pi(n)\}, \forall k$

Compute optimal utility $\sum_i U_i(t_i^*)$ by solving (3.12)-(3.15)

Find energy budget $B_i^* = \sum_k \Phi_i(\mathbb{A}_k, T_k^*)$

if $U_{\text{best}} < \sum_i U_i(t_i^*)$

Update $U_{\text{best}} \leftarrow \sum_i U_i(t_i^*), B_{i,\text{best}} \leftarrow B_i^*$

end if

end for

Figure 3.2 : Algorithm for solving Problem BEM (budget-based energy management) defined in Section 3.2.

Remark 1 Substituting $T_{\mathbb{S},k}$ in Equation (3.3) and $T_{i,k}$ in Equation (3.4) into Equation (3.13) and Equation (3.14), it is easy to verify that for any given ordering π , Problem BEM is a convex optimization with $n + 1$ linear constraints over $2n$ variable, i.e., $t_i, \forall i$ and $T_k, \forall k$. Problem BEM can be solved efficiently by the standard Newton Method in [22], which is guaranteed to converge to the optimal energy management solution.

Energy Accounting Policy does not Matter for BEM: Problem BEM formulated in (3.6-3.11) clearly relies on the choice of energy accounting policy $\phi_i(\cdot), \forall i$. However, in the following we prove a somehow counter-intuitive result, which shows that the optimal utility value and process run time of Problem BEM is irrelevant of energy accounting policies.

Theorem 1 Problem BEM is independent of energy accounting, i.e., for an arbitrary energy accounting function that satisfies the Efficiency property, there always exists a set of energy budget $B_i, \forall i$, which achieve the same optimal utility value.

Proof of Theorem 1

Proof 3.1 We have shown that for an arbitrary permutation π , Problem BEM is equivalent to a convex optimization in (3.12)-(3.15). It is easy to see that the convex optimization does not depend on the choice of energy accounting policy. Therefore, the optimal utility value and process run time, which are maximized over all possible permutations, are irrelevant of the energy accounting function used.

End of Proof

The intuitive explanation of the proof is as follows. In particular, we show that for a feasible set of run time $t_i, \forall i$, there always exists a way to partition total energy C among individual budgets $B_i, \forall i$ accordingly, no matter what energy accounting policy is used. Suppose one can achieve an optimal utility value with run time $t_i, \forall i$, using an ordering π and a specific energy accounting policy. Then the corresponding energy budget of each process should be calculated from (3.5) and (3.8). If one chooses to use another energy accounting policy and would like to achieve the same optimal utility value, the corresponding energy budget should be calculated in the same way, with the new energy accounting policy. The key idea is that as long as the Efficiency property holds, the sum of individual energy budget would remain unchanged and satisfy the total energy constraint $\sum_i B_i = C$.

To summarize, for any energy accounting policy, one can always reverse-engineer an optimal BEM solution and calculate a corresponding energy partition to achieve the optimal utility value.

3.3 Optimal Energy Management (OEM)

3.3.1 Problem Formulation

We consider an optimization which aims at maximizing aggregate process utilities under a sum energy constraint C . The Optimal Energy Management problem, denoted by Problem OEM, is formulated as follows:

Problem OEM :

$$\text{maximize} \quad \sum_{i=1}^n U_i(t_i) \quad (3.16)$$

$$\text{subject to} \quad t_i = \sum_{\mathbb{S}:i \in \mathbb{S}, |\mathbb{S}| \leq m} T_{\mathbb{S}}, \quad \forall i, \quad (3.17)$$

$$\sum_{\mathbb{S}} E(\mathbb{S}) \cdot T_{\mathbb{S}} / T_0 \leq C, \quad (3.18)$$

$$\text{variables} \quad T_{\mathbb{S}}. \quad (3.19)$$

3.3.2 Performance Analysis

When utility functions are concave, Problem OEM is a convex optimization with linear constraints, which can be solved efficiently using standard algorithm, e.g., the Newton Method [22]. The algorithm for solving Problem OEM is summarized by Figure 3.3 and its complexity is analyzed below.

Remark 2 The algorithm for solving Problem OEM is summarized in Figure 3.3. It is easy to verify that the inequality constraint Equation (3.18) in Problem OEM must be

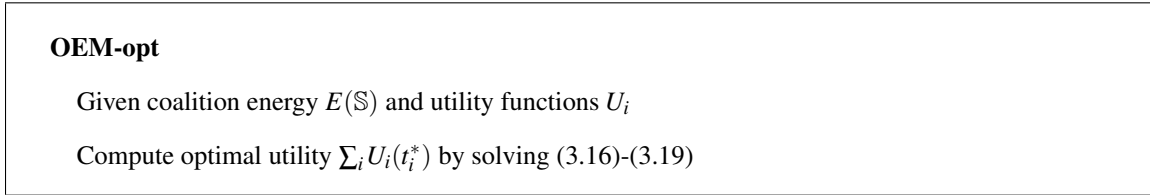


Figure 3.3 : Algorithm for solving Problem OEM (optimal energy management) defined in Section 3.3.

satisfied with exact equality at optimum, because underspending energy always results in non-optimal for maximizing process run time. Therefore, Problem OEM is a convex optimization with $n + 1$ linear constraints over $n + 2^{\min(n,m)}$ variables, i.e., $t_i, \forall i$ and $T_{\mathbb{S}}, \forall |\mathbb{S}| \leq m, \mathbb{S} \subseteq \mathbb{N}$. It can be solved by the standard Newton Method in [22]. Further, it can be shown that for $\alpha \geq 0$ Problem OEM satisfies the convergence conditions in [22] and its convergence can be derived in closed form for arbitrary tolerance. While Problem OEM dominates Problem BEM in terms of achievable utility, it has higher complexity since there are $n + 2^{\min(n,m)}$ optimization variables, instead of $2n$ variables for each ordering π in Problem BEM.

While implementing the solution of Problem OEM in practice requires logging the run time of all feasible coalitions and results in higher overhead than that of BEM, it is guaranteed to outperform BEM in energy management.

Theorem 2 Problem OEM always dominates Problem BEM. Its optimal utility value is higher than or equal to that of Problem BEM.

Proof of Theorem 2

Proof 3.2 We prove Theorem 2 by showing that any optimal solution for Problem BEM is also feasible for Problem OEM. Let π, T_k, B_i be a maximizer for Problem BEM and

achieves process run time t_i . It remains to find $T_{\mathbb{S}}$, in Problem OEM, which are able to achieve the same process run time t_i .

Toward this end, we start with Equation (3.14) that is shown to be satisfied by π, T_k, B_i of Problem BEM. We have

$$\begin{aligned}
C &= \sum_k \sum_{\mathbb{S}: \mathbb{S} \subseteq \mathbb{A}_k, |\mathbb{S}| \leq m} E(\mathbb{S}) T_{\mathbb{S},k} / T_0 \\
&= \sum_{\mathbb{S}: |\mathbb{S}| \leq m} \sum_{k: \mathbb{S} \subseteq \mathbb{A}_k} E(\mathbb{S}) T_{\mathbb{S},k} / T_0 \\
&= \sum_{\mathbb{S}: |\mathbb{S}| \leq m} E(\mathbb{S}) \cdot \sum_{k: \mathbb{S} \subseteq \mathbb{A}_k} T_{\mathbb{S},k} / T_0
\end{aligned} \tag{3.20}$$

where $T_{\mathbb{S},k}$ is a function of T_k defined by (3.3). The result implies that the following choice of $T_{\mathbb{S}}$ satisfies the energy budget constraint $\sum_{\mathbb{S}} E(\mathbb{S}) \cdot T_{\mathbb{S}} / T_0 \leq C$:

$$T_{\mathbb{S}} = \sum_{k: \mathbb{S} \subseteq \mathbb{A}_k} T_{\mathbb{S},k}. \tag{3.21}$$

Such $T_{\mathbb{S}}$ also achieves the same utility value as Problem BEM. To show this, we use (3.4) in the last step below and, for each user i , obtain:

$$\begin{aligned}
\sum_{\mathbb{S}: i \in \mathbb{S}, |\mathbb{S}| \leq m} T_{\mathbb{S}} &= \sum_{\mathbb{S}: i \in \mathbb{S}, |\mathbb{S}| \leq m} \sum_{k: \mathbb{S} \subseteq \mathbb{A}_k} T_{\mathbb{S},k} \\
&= \sum_{k: i \in \mathbb{A}_k} \left[\sum_{\mathbb{S}: i \in \mathbb{S}, \mathbb{S} \subseteq \mathbb{A}_k, |\mathbb{S}| \leq m} T_{\mathbb{S},k} \right] \\
&= \sum_{k: i \in \mathbb{A}_k} T_{i,k}
\end{aligned} \tag{3.22}$$

which is exactly constraint (3.7) in Problem BEM since $t_{i,k} = 0$ for any inactive process $i \notin \mathbb{A}_k$. Therefore, the choice of $T_{\mathbb{S}}$ here satisfies constraints (3.17)-(3.18) of Problem OEM and is able to achieve the same optimal utility value as Problem BEM. It implies that the optimal utility of Problem OEM can be no less than that of Problem BEM.

End of Proof

The intuitive explanation of the proof is as follows. Suppose there exists an optimal solution to Problem BEM. That is, to achieve the optimal BEM utility value, one divides total energy constraint C into individual budgets $B_i, \forall i$, schedules processes based on their budget and priority, and charges them accordingly until their energy budget reduces to zero. If all the scheduling decisions and run time (i.e. \mathbb{S} and $T_{\mathbb{S}}$) are recorded during the entire process, one can apply them in Problem OEM and achieve the same process run time, resulting the same utility value. This is feasible because Problem OEM optimizes over all possible process coalitions and has a significant larger number of “control knobs” than Problem BEM does. Therefore, any optimal BEM utility is achievable in OEM. Optimal OEM utility must be higher than or equal to that of Problem BEM. In Section 6.4.3, we provide numerical examples where OEM strictly dominates BEM. Furthermore, we provide the following theorem, which establishes a sufficient condition for BEM to coincide with OEM.

Theorem 3 If $n \leq m$ and coalition energy is sub-modular, i.e.,

$$E(\mathbb{S}_1 \cup \mathbb{S}_2) + E(\mathbb{S}_1 \cap \mathbb{S}_2) \leq E(\mathbb{S}_1) + E(\mathbb{S}_2) \forall \mathbb{S}_1, \mathbb{S}_2 \subseteq \mathbb{N}, \quad (3.23)$$

Problem BEM achieves the same optimal utility value as Problem OEM does.

Proof of Theorem 3

Proof 3.3 To prove Theorem 3, we show that an optimal OEM solution can be converted into a feasible BEM solution under the conditions. We start by considering an optimal OEM solution where k distinct coalitions $\mathbb{S}_1, \mathbb{S}_2, \dots, \mathbb{S}_k$ receive positive run time. It can be shown that for any two coalitions \mathbb{S}_i and \mathbb{S}_j for $i, j \in \{1, \dots, k\}$, we must have either $\mathbb{S}_i \subseteq \mathbb{S}_j$ or $\mathbb{S}_j \subseteq \mathbb{S}_i$. Otherwise, the following procedure can be performed to construct a contradiction. If there exists a pair of coalitions with run time $T_{\mathbb{S}_i} \leq T_{\mathbb{S}_j}$ that violate the

statement, in the optimal OEM solution we can replace the pair of coalitions by $\mathbb{S}_i \cap \mathbb{S}_j$ and $\mathbb{S}_i \cup \mathbb{S}_j$, both with run time $T_{\mathbb{S}_i}$, and \mathbb{S}_j with run time $T_{\mathbb{S}_j} - T_{\mathbb{S}_i}$. The replacement achieves the same run time for processes and satisfy

$$(\mathbb{S}_i \cap \mathbb{S}_j) \subseteq \mathbb{S}_j \subseteq (\mathbb{S}_i \cup \mathbb{S}_j). \quad (3.24)$$

Due to the sub-modular condition, we have

$$E(\mathbb{S}_i \cap \mathbb{S}_j) + E(\mathbb{S}_i \cup \mathbb{S}_j) \leq E(\mathbb{S}_i) + E(\mathbb{S}_j). \quad (3.25)$$

which implies that such a replacement does not increase any energy consumption.

We can continue the procedure until all such coalition pairs are replaced. It is guaranteed to converge because each replacement strictly increases the variance of cardinality distribution $|\mathbb{S}_1|, \dots, |\mathbb{S}_k|$, which is in turn upper bounded. Therefore, after the procedure converges, coalitions in the optimal OEM solution must form a sequence of contraction $\mathbb{S}_{\pi_1} \subseteq \mathbb{S}_{\pi_2} \subseteq \dots \subseteq \mathbb{S}_{\pi_k}$ for some ordering π_1, \dots, π_k . It is easy to see that choosing $A_{k-i} = \mathbb{S}_{\pi_i}$ for $i = 1, \dots, k$ generates a feasible solution for Problem BEM and achieves the same optimal utility value.

End of Proof

This theorem states that when all n processes can be scheduled concurrently and the coalition energy is sub-modular, Problem OEM and Problem BEM become equivalent and both achieve the same utility value.

When max-min fair OEM that maximizes the minimum run time, i.e., $\sum_i U_i(r_i) = \min_i t_i$, is concerned, we can bound the performance gap between OEM and BEM as follows.

Theorem 4 For max-min fair OEM (i.e., with a utility $\sum_i U_i(r_i) = \min_i t_i$) and equal process priority, the optimal utility U_{BEM} is bounded by U_{OEM} :

$$U_{\text{OEM}} \cdot \frac{E_{\min}}{E_{\text{avg}}} \leq U_{\text{BEM}} \leq U_{\text{OEM}} \quad (3.26)$$

where $E_{\min} = \min_{\mathbb{S}:|\mathbb{S}|\leq m} E(\mathbb{S})$ is the minimum energy consumption and $E_{\text{avg}} = \sum_{\mathbb{S}:|\mathbb{S}|\leq m} E(\mathbb{S}) / \binom{n}{\min(m,n)}$ is the average energy consumption per coalition.

Proof of Theorem 4

Proof 3.4 The second inequality has been proven in Theorem 4 for arbitrary utility functions. To prove the first inequality, for max-min fair utility, both BEM and OEM should assign equal run time to all processes, respectively. This is because if there exists $t_i < t_j$, we can transfer a sufficiently small energy budget from process j to process i to increase run time of process i and therefore improve the max-min utility function. For BEM, it implies that all process energy budgets should run out simultaneously in order to maximize the minimum run time. We conclude that $A_k = \{\cdot\}$ for $k = 1, \dots, n-1$ and $A_n = \{1, \dots, n\}$. Therefore, we have $T_k = 0$ for $k < n$. Due to equal scheduling priorities, all coalitions of size $|\mathbb{S}| = \min(n, m)$ receive equal amount of run time during interval T_n , i.e., $T_{\mathbb{S},n} = T_n / \binom{n}{\min(m,n)}$, in which $\binom{n}{\min(m,n)}$ is the number of coalitions of of size $\min(n, m)$.

Plugging these results into the energy constraint (3.14), we obtain

$$\begin{aligned} C &= \sum_k \sum_{\mathbb{S}: \mathbb{S} \subseteq A_k, |\mathbb{S}| \leq m} E(\mathbb{S}) \cdot T_{\mathbb{S},k} / T_0 \\ &= \sum_{\mathbb{S}: |\mathbb{S}| \leq m} E(\mathbb{S}) \cdot T_{\mathbb{S},n} / T_0 \\ &= \sum_{\mathbb{S}: |\mathbb{S}| \leq m} E(\mathbb{S}) \cdot \frac{T_n / T_0}{h(\min(m, n), n)} \\ &= E_{\text{avg}} \cdot T_n / T_0 \end{aligned} \quad (3.27)$$

$$= E_{\text{avg}} \cdot \frac{n}{T_0 \cdot \min(m, n)} U_{\text{BEM}} \quad (3.28)$$

where the second step uses the fact that $A_k = \{\cdot\}$ for $k \leq n - 1$ and $A_n = \mathbb{N}$. Due to the symmetric process priorities, the sum of process run time $\min(m, n) \cdot T_n$ (i.e., there are $\min(m, n)$ concurrent processes for T_n seconds) are evenly shared by all processes. It implies $U_{\text{BEM}} = t_i = \min(m, n) \cdot T_n/n$, which is used in the last step above.

For OEM, since all t_i are equal, we can derive that

$$\begin{aligned}
U_{\text{OEM}} &= \frac{1}{n} \sum_i t_i \\
&= \frac{1}{n} \sum_i \sum_{\mathbb{S}: i \in \mathbb{S}, |\mathbb{S}| \leq m} T_{\mathbb{S}} \\
&= \frac{1}{n} \sum_{\mathbb{S}: |\mathbb{S}| \leq m} \sum_{i: i \in \mathbb{S}} T_{\mathbb{S}} \\
&= \frac{1}{n} \sum_{\mathbb{S}: |\mathbb{S}| \leq m} \min(m, n) \cdot T_{\mathbb{S}} \\
&\leq \frac{\min(m, n)}{n} \sum_{\mathbb{S}: |\mathbb{S}| \leq m} \frac{E(\mathbb{S})}{E_{\min}} \cdot T_{\mathbb{S}} \\
&\leq \frac{T_0 \cdot \min(m, n)}{nE_{\min}} \sum_{\mathbb{S}: |\mathbb{S}| \leq m} E(\mathbb{S}) \cdot T_{\mathbb{S}}/T_0 \\
&\leq \frac{T_0 \cdot \min(m, n)}{nE_{\min}} \cdot C
\end{aligned} \tag{3.29}$$

where the first step uses constraint (3.17), the forth step is due to the fact that $|\mathbb{S}| = \min(n, m)$, and the last step uses constraint (3.17). Combining (3.28) and (3.29) completes the proof of Theorem 4.

End of Proof

3.4 Remarks

In this chapter, we made three assumptions to facilitate the analysis. First, in BEM we assumed that each process i has a known probability p_i to be scheduled if the number of active processes in the system exceeds the maximum that can be scheduled concurrently within

one scheduling period. If scheduling probabilities further vary by time or energy budget, we need to further partition each execution interval T_k with processes \mathbb{A}_k into sub-intervals, in which the scheduling probabilities are constant. Even though this would greatly increase the complexity of Problem BEM, it remains to be a convex optimization and can be solved by the proposed algorithm.

Second, we assumed that the time scale of energy management is sufficiently larger than a single scheduling period. Thus, the Law of Large Number can be applied. This assumption is generally true in modern systems where energy management works at the time scale of tens of seconds or even minutes while a scheduling period is one hundredth of a second, i.e., 10 ms.

Finally, we have ignored the uncertainty of user interaction on energy management. We assume that user preference can be quantified by deterministic utility functions, while stochastic models of user behaviors are beyond the the scope of this dissertation. In practice, energy cost $E(\mathbb{S})$ may further depend on the hardware state in a mobile system such as CPU frequency, LCD brightness, and WiFi active/idle mode, etc. In Chapter 6, we will show that such dependency can be factored into our framework by incorporating hardware state as a condition into $E(\mathbb{S})$.

3.5 Summary

This chapter began with the theoretical foundation that formulates system energy management as an utility optimization problem. Using the optimization framework, this chapter analyzed budget based management (BEM), a widely used energy management method in system research and demonstrated that energy accounting does not matter at all as long as it satisfies the Efficiency property. Finally, this chapter proposed a novel optimal energy management (OEM), which always dominates BEM by optimal utility value while requir-

ing as much as information as needed by Shapley value based accounting is needed, i.e.,
 $E(S) \forall S \subseteq N$.

Chapter 4

From Theory to Practice: The Key is *In Situ* $E(\mathbb{S})$ Estimation

Chapter 2 and Chapter 3 have theoretically demonstrated the superiority of Shapley value based energy accounting and OEM based scheduling, respectively. To calculate Shapley value and OEM both require the system energy consumption for all possible combinations of processes under question, i.e., $E(\mathbb{S}) \forall \mathbb{S} \subseteq \mathbb{N}$. Acquiring such data, however, is extremely challenging in practice because (i) the number of the process combinations can be large ($2^{|\mathbb{N}|}$); (ii) $E(\mathbb{S})$ depends on the process execution dynamics and hardware states. The solution to address these challenges is to *acquire system energy consumption in situ for very short time intervals, down to the OS scheduler period, i.e., 10 ms*. In this chapter, we review two candidate approaches for *in situ* energy estimation, i.e., system energy model and smart battery interface, and demonstrate why neither of them alone is sufficient to fulfill the role due to inaccuracy and low rate of energy estimation.

4.1 Challenges and Solutions

Applying Shapley value to per-process energy cost and OEM faces a number of significant challenges. Given n processes and given $E(\mathbb{S})$ for all $\mathbb{S} \subseteq \mathbb{N}$, in theory, both calculating the energy contribution by a process based on Shapley value (Section 2.2) and the optimal energy management (Section 3.3) are straightforward. The key problem, however, is to obtain $E(\mathbb{S})$ for all $\mathbb{S} \subseteq \mathbb{N}$, which poses three system challenges. (i) First, there are 2^n subsets

for n processes. In order to apply Shapley value and conduct optimal energy management, one must obtain $E(\mathbb{S})$, the system energy consumption, for 2^n different coalitions, i.e., \mathbb{S} , which can be practically very difficult, if possible at all. (ii) Second, $E(\mathbb{S})$ depends on not only which processes are running, but also the dynamics of process execution such as the CPU time of each process and the execution order. Therefore, $E(\mathbb{S})$ is not a fixed number but a random variable. The variance of $E(\mathbb{S})$ introduces uncertainty in energy accounting by Shapley value. (iii) Finally, $E(\mathbb{S})$ is further affected by the hardware state in a mobile system such as CPU frequency, LCD brightness, and WiFi active/idle mode. This will introduce even more variance in $E(\mathbb{S})$.

The first two challenges can be tackled by estimating system energy consumption, E , *in situ* for very short time intervals, down to the OS scheduling period, which is 10 ms in most modern mobile systems, such as Android and iOS. Fewer processes can run simultaneously in a shorter time interval. In a time interval of 10 ms, n is not many more than the number of processing units. Moreover, in a shorter time interval, the process execution dynamics have less impact on energy consumption. By monitoring the system energy consumption *in situ* for an extended period of time, one can potentially acquire $E(\mathbb{S})$ for many different \mathbb{S} . In [16], the authors showed that system energy consumption for 10 ms time intervals can be estimated with about 80% accuracy *in situ* using power readings from the battery interface available in commodity mobile systems. In Section 6.1, we present a new system solution that is able to measure system energy consumption at 200 Hz with 95% accuracy.

Finally, to address the third challenge, one can incorporate hardware states as a condition into $E(\mathbb{S})$. Let σ denote current hardware state. We can estimate system energy consumption for a coalition \mathbb{S} given the hardware state σ , denoted by $E(\mathbb{S}, \sigma)$. As a result, one can apply Shapley value separately to time intervals with a given hardware state. However, such an approach will further increase the number of $E(\mathbb{S}, \sigma)$. As a result, there

are many $E(\mathbb{S}, \sigma)$ that can not be covered in historical measurement data. Therefore, we employ two solutions to estimate unknown $E(\mathbb{S}, \sigma)$ from measured $E(\mathbb{S}, \sigma)$ in historical data. This is is described in Section 6.2 in Chapter 6.

4.2 System Energy Modeling Review

We next discuss why prior system energy modeling method is insufficient to provide $E(\mathbb{S})$

4.2.1 System Energy Model Primer

An energy model estimates the energy consumption by a mobile system in a given time period. Mathematically, an energy model can be represented by a function f such that

$$E = f(x_1, x_2, \dots, x_p),$$

where E is the energy consumption in time interval T and x_1, x_2, \dots, x_p are the system behavior data for the same period that can be read in software. To use the terminology of regression analysis, E , is the response of the model and x_1, x_2, \dots, x_p are the predictors. Most system energy models employ a linear function for f . The most important metrics of a model are *accuracy* and *rate*, or the reciprocal of the time interval T for which the energy consumption is modeled. While the importance of high accuracy is obvious, a high rate is also important. As discussed in section 4.1, Shapley value based energy accounting and OEM based scheduling both require energy estimation for time intervals of at least 10 ms, or at a rate no lower than 100 Hz.

Existing system energy modeling approaches are however fundamentally limited in two important ways. First, while cycle-accurate hardware power models exist, all reported system energy models except [23], as will be discussed in Section 4.2.2, have a low rate. That is, they estimate energy consumption for a time interval of one second or longer. In another

word, they estimate energy at a rate of 1 Hz or lower. While they work well for certain objectives, e.g., thermal analysis/management and battery lifetime estimation, energy models of 1 Hz are fundamentally inadequate for Shapley value based energy accounting and OEM based scheduling. Second, all existing methods except [13, 24, 5] generate the model of a system in the lab using high quality external power measurements. Such methods are not only labor-intensive but also produce fixed energy models that are determined by the workload used in model construction. Modern mobile systems such as smartphones and laptops challenge such methods as will be analyzed in Section 4.2.3.

4.2.2 Prior Work on System Energy Modeling

There is an extensive body of literature on system energy modeling. All existing system energy modeling methods except [23] give energy estimation at a low rate, such as one per minute [25], one per several seconds [26], and one per second (1 Hz) [27, 12, 28]. In contrast, Sesame achieves a rate as high as 100 Hz, or one per 10 ms, as required by energy accounting in multitasking systems. It is worth noting that energy models for a system component, e.g., processor [29, 30] and memory [31], can achieve much higher rates, often cycle-accurate. Yet such models require circuit-level power characterization, which is impractical to system energy modeling.

A major limitation of these models is that all of them are generated by a fixed set of benchmark applications. As shown by [32, 33], smartphone usage is diverse among different users. Different usage will lead to different energy models (See Section 4.2.3). Therefore, a model generated by a fixed set of benchmark applications can be less accurate when a user runs a new application. On the contrary, Sesame utilizes real usage of every individual user to generate personalized energy models and to adapt such models for potentially higher accuracy.

All existing methods except [13, 24, 5] employ external assistance, e.g., a multi-meter, to measure the power consumption of the target system. iCount [24] leverages the special property of switching regulators to measure the energy output of a switching regulator and account the energy consumption in a wireless sensor network [34]. This approach, however, requires special circuitry that are not general available in computer systems. JouleMeter [13] and PowerBooter [5] build power models for virtual machines using power sensors coming with servers and for smartphones using battery sensors, respectively. They do share our philosophy of self-modeling but fall short of achieving our goal for three important reasons. (i) They both have a low rate, 0.5 Hz [13] and 0.1 Hz [5]. (ii) Both employ fixed predictors, identified through careful, manual examination of the system power characteristics. In contrast, we target at adaptive models that statistically select the best predictors. Such adaptation is important to mobile devices where hardware/usage diversity is high. (iii) Both employ an expensive calibration to construct the model, making it inherently inefficient for adaptation. Moreover, PowerBooter requires the battery discharging curve, which is battery-dependent and, therefore, compromises the goal of self-modeling.

In [23], the authors view a system as a finite-state-machine (FSM) and associate each state with a fixed power number. They trace system calls to determine which state the system is in and thus obtain the corresponding power number. This FSM model achieves high accuracy at a high rate of 20 Hz (80% of the estimation has an error less than 10%). However, this approach requires external hardware to measure power as well as a sophisticated calibration procedure that requires device specific hardware information. In contrast, Sesame does not require any external hardware or additional calibration.

4.2.3 Limitations of Prior Work

We next highlight the dependencies of energy models on the hardware configuration and usage of a mobile system and therefore motivate the proposed self-modeling approach. Out of simplicity, we use the 1 Hz energy model based on CPU utilization as used in [27, 35]. Our target system is a Lenovo ThinkPad T61 laptop. We measure its power with a USB-2533 Data Acquisition System from Measurement Computing along with another PC.

Dependency on Hardware

Energy models may change due to hardware change. For example, if more memory is added, the system power consumption will likely increase, given the same CPU utilization. We study three hardware configurations of the T61 laptop with only minor differences, i.e., fixed CPU frequency of 2 GHz, fixed CPU frequency of 800 MHz; and DVS (Dynamic Voltage Scaling) enabled. Similar to existing modeling methods [27, 35], we build a calibrator program that switches between busy loops and idle with a controllable duty cycle to produce workload with different CPU utilization. At each level of CPU utilization, we measure the laptops power consumption. Figure 4.1(left) presents the linear energy model for each of the hardware configurations. It clearly shows that there are considerable differences (up to 25%) among three models. This simple example demonstrates the generic dependency of energy models on hardware configuration. We note that the model difference in this particular example can be eliminated by adding CPU frequency as a predictor. However, it is impractical to include all predictors that can account for every hardware detail because not all hardware components are visible to the OS and collecting the values of a large number of predictors would incur significant overhead.

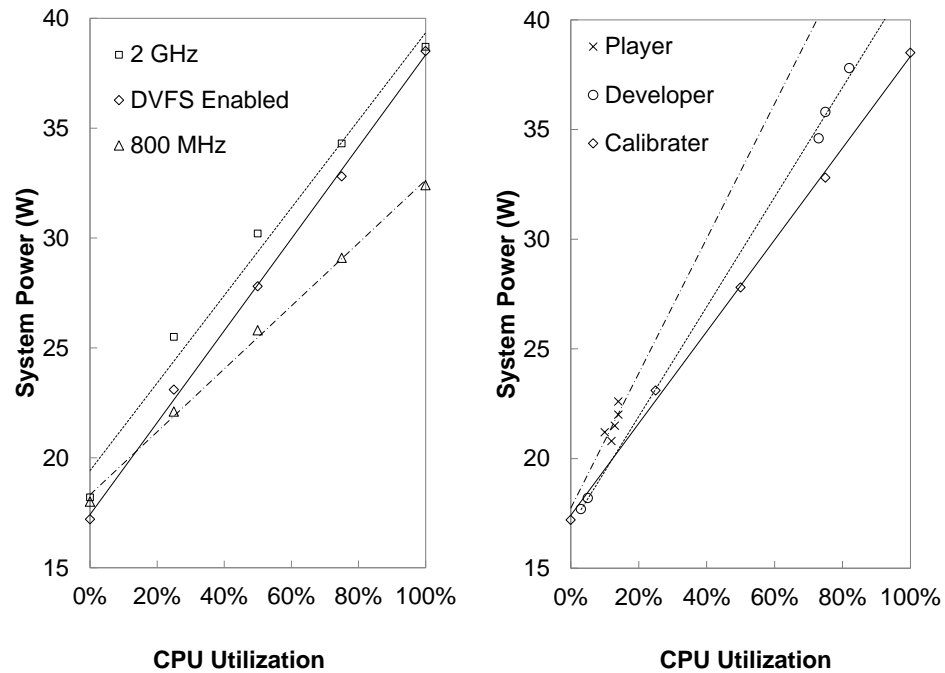


Figure 4.1 : Linear system energy models based on CPU utilization with (left) different hardware configurations and (right) different usage.

Dependency on Usage

Using the same setup, we next illustrate how usage of the system can impact its energy model. Different software may invoke different hardware components. Since some components are invisible to the OS, difference in invisible components induced by software will not be noticed by the OS. To demonstrate the usage dependency, we build three linear models by running three different benchmarks: the calibrater described above, a SW Developer benchmark (Developer), and a Media Player benchmark (Player), both of the latter two from the Linux Battery Life Toolkit (BLTK) [36]. As shown in Figure 4.1(right), the difference among their estimations can be as high as 25%. Again, one may argue that

such differences can be reduced by incorporating as many benchmarks as possible into a factory-built model. However, using more benchmarks will lead to a model that generalizes decently but does not achieve high accuracy for the most common usage. As the usage of mobile systems by different users can be very different [32] and the usage by the same user can evolve over time [37], factory-built models are unlikely to provide an improved accuracy for all users and all usage.

The dependencies of the system energy models on hardware configuration and usage suggest “personalized” models be built for a mobile system, which is only possible through a self-modeling approach.

4.3 Battery Interface Characterization

We next provide an experimental characterization of smart battery interfaces available in modern mobile systems. We focus on their limitations in accuracy and rate in order to suggest solutions.

4.3.1 Smart Battery Interface Primer

Most batteries of modern mobile systems employ a smart battery interface in order to support intelligent charging, battery safety, and battery lifetime estimation.

A smart battery interface includes both hardware and software components. The key hardware of a smart battery interface is a fuel gauge IC that measures battery voltage, temperature, and maybe current. The fuel gauge IC estimates the remaining battery capacity with one of the following two methods. The first employs a built-in battery model that estimates remaining capacity using the voltage and temperature, e.g., Maxim MAX17040. The other method measures the discharge current using a sense resistor, e.g., Maxim DS2760 and TI BQ20z70. By periodically measuring the discharge current, the fuel gauge IC can

count the total charges drawn from the battery and calculate the remaining capacity accordingly. Apparently, a smart battery interface with current measurement capability is much better for self energy measurement.

In software, the fuel gauge IC exposes several registers that store the values of temperature, voltage, remaining capacity, and sometimes discharge current, of the battery and updates these registers periodically. The OS, often through ACPI, provides a battery driver that can read such registers through a serial bus such as SMBus and I²C. Linux employs a virtual file system for the readings from the battery driver so that applications can access battery information through a standard file system read API.

4.3.2 Experimental Setup

To investigate the accuracy and rate of battery interface readings, we have characterized three types of mobile systems, i.e., laptop, smartphone and PDA, as summarized by Table 2. In the characterization, we employ a set of benchmark applications for each of the mobile system. For laptops and the smartphone with Maemo, we use the BLTK [36]. For other smartphones, we use JBenchmark [38] and 3DMarkMobile [39]. Having each benchmark application running, we measure the discharge current out of the battery using 1K Hz sampling rate, average every ten samples and use the data, at a rate of 100 Hz, as the ground truth (DAQ) to compare with the readings from battery interfaces. The measurement is performed at the output of the battery with the system being powered by the battery.

4.3.3 Technical Observations

We make the following observations. *First, there is a huge variety in the smart battery interfaces on mobile systems.* We identify three types. (i) The voltage-based interface gives the remaining battery capacity based on the battery output voltage, e.g., Nokia N810 and

Table 4.1 : Battery interfaces of popular mobile systems

System	OS	Current	Rate (Hz)
Lenovo ThinkPad T61	Linux	Yes	0.5
Dell Latitude D630	Linux	Yes	1
Nokia N85/N96	Symbian	Yes	4
Nokia N810/N900	Maemo	No	0.1
Google G1/Nexus One	Android	Yes	0.25

N900. (ii) The instant interface gives the instantaneous discharge current, such as Nokia N85 (See Figure 4.2(top)). (iii) Finally, the filtered interface gives a low-pass filtered reading of the discharge current, such as ThinkPad T61. The fuel gauge IC used in T61 battery interface employs a low-pass filter that reports discharge current values by averaging ten readings in last 16 seconds. This low-pass filter effect is illustrated in Figure 4.2(bottom) and is designed to facilitate the estimation of battery lifetime using history [40]. Unlike N85, the battery interface readings of T61 have a delay, around 16 seconds, than the real values. We note that modern mobile devices are increasingly using the instant interfaces. We further find that different mobile devices have different reading rates in their battery interfaces. That is, they try to give the average discharge current for time intervals of different lengths. The shorter the interval, the higher reading rate the interface has. Table 4.1 summarizes the battery interface reading rates of all the mobile systems we have characterized.

Consequently, our second observation is: *the rate of the smart battery interface is low*

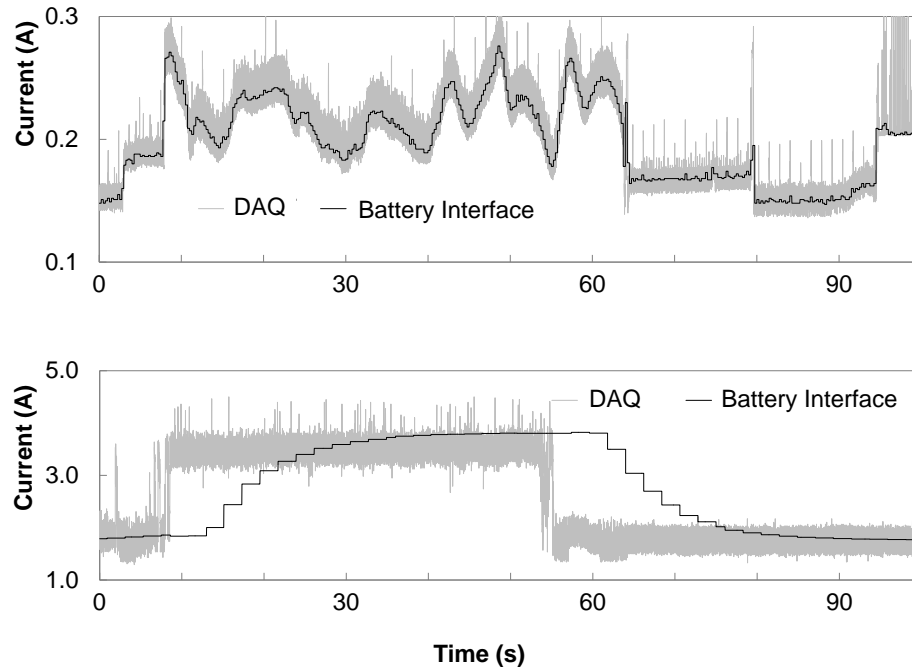


Figure 4.2 : Current readings from accurate external measurement (DAQ) and battery interfaces of Nokia N85 (top) and Lenovo ThinkPad T61 (bottom).

with 4 Hz being the highest observed. The reason is simple: high-rate measurement requires a more expensive and more power-hungry fuel gauge IC, which does not bode well for a cost-sensitive and power-sensitive battery for mobile systems. Some fuel gauge ICs, e.g., TI BQ2019 [40], employ a charge counter, which increases by one unit whenever a certain amount of charge is drawn from the battery. As a result, the higher the current, the faster the counter updates, up to 30 Hz during peak power consumption. However, because a mobile system consumes moderate or low power in most of the time, the average measurement rate is only 1.138 Hz according to the datasheet [40]. The low rate of smart battery interfaces is an important barrier toward our goal of realizing Sesame with high rate (up to 100 Hz).

Our third observation is: *the error of the instant battery interface reading is high.* We

use three representative systems, N85, T61 and N900 to demonstrate this. The battery interfaces of N85 and T61 can measure current at 4 Hz and 0.5 Hz, respectively. That of N900 does not support direct current measurement but the average current for a period can be calculated from the readings of the remaining battery capacity at the beginning and end of the period at 0.1 Hz. To evaluate the accuracy of these battery interfaces, we prepare the ground truth using the measurement from the 100 Hz DAQ. For example, to compare with the 4 Hz readings from N85, we use the average of the corresponding 25 samples collected from the 100 Hz DAQ data as the ground truth. Figure 4.3 shows the root mean square (RMS) of all the relative errors for N85 at 4 Hz (33%), for T61 at 0.5 Hz (19%), and for N900 at 0.1 Hz (45%). Such high instantaneous errors pose a great challenge to energy modeling using smart battery interface readings.

On the other hand, we also observe that *the instantaneous errors can be reduced by averaging readings to reduce the rate*. For example, we can average every four readings from the N85 4 Hz battery interface to measure the average current for each second or a 1 Hz reading. This 1 Hz current reading will have only 10% RMS error compared to 33% at 4 Hz. Figure 4.3 shows the RMS errors for N85, T61, and N900 as the reading rate is reduced down to 0.01 Hz or one reading per 100 seconds. The RMS error decreases as the reading rate decreases. There are two reasons for this error reduction. First, random noises in the smart battery interfaces contribute significantly to the instantaneous errors. By averaging the instantaneous readings to produce a low-rate reading, the random noise can be canceled out. Therefore, the model construction such as linear regression will readily address such random noises. Second, the low-pass filter employed by the battery interface of T61 introduces significant errors, e.g., Figure 4.2(bottom) for T61. By averaging the instantaneous readings to reduce the reading rate, the reduced rate will approach or even drop below that of the low-pass filters cutoff frequency, which leads to a reduced error. In

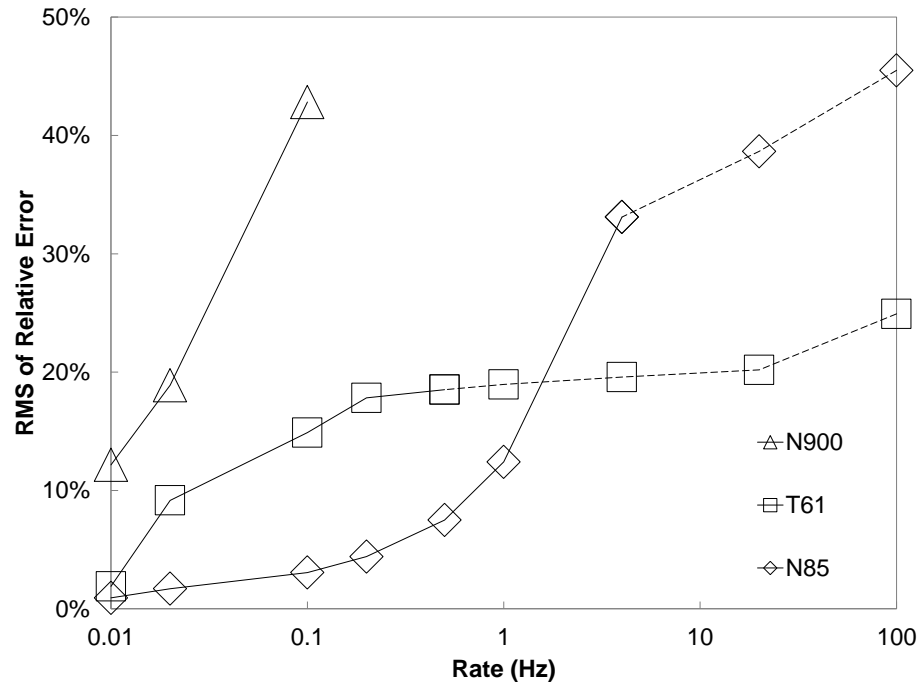


Figure 4.3 : Error vs. rate for battery interface readings. Dashed lines are calculated by comparing current readings at the highest rate against measurement.

Section 5.1.3, we will present methods to address this error.

Finally, accessing the battery interface in software contributes little to the system energy consumption by the battery interface reading. To obtain the extra energy consumed by battery interface reading, we run each of the benchmark applications twice. In the first time; we perform the experiment as described above, while in the second time, we only measure the power using the USB-2533 Data Acquisition System. Then we compare the average power consumption of the two and the results show that the contribution of battery interface reading is negligible, less than 1% for both systems.

4.4 Summary

This chapter began with identifying the system challenge in realizing Shapley value based energy accounting and OEM scheduling is to obtain $E(S) \forall S \subseteq \mathbb{N}$; and proposing a corresponding solution, i.e., to acquire system energy consumption *in situ* for very short time intervals, down to the OS scheduler period, i.e., 10 ms. Then the chapter reviewed system energy model and smart battery interface as two candidate approaches for *in situ* energy estimation and demonstrate both of them were insufficient because of inaccuracy and low rate of energy estimation.

Chapter 5

In Situ $E(\mathbb{S})$ Estimation using Existing Battery Interface

Chapter 4 has demonstrated either system energy model or smart battery interface alone is insufficient for *In Situ* $E(\mathbb{S})$ estimation due to inaccuracy and low rate. In this chapter, we propose a novel approach to address these two limitations, i.e., to enable a mobile system to construct a high-rate system energy model without external assistance. We call this approach *self-modeling*, or *Sesame*. Instead of using special circuitry for system power measurement, e.g., [24, 34] and our alternative design to be described in Section 6.1, our key idea to leverage the smart battery interface already available on mobile systems to improve the accuracy and rate of energy estimation by prior system energy modeling approach.

5.1 Design of Sesame

We next provide the design of Sesame and our key techniques to overcome the limitations of battery interfaces.

5.1.1 Design Goals and Architecture

We have the following design goals for Sesame to cater to a broad range of needs in energy models.

Sesame requires no external assistance. It will collect the data traces, identify important predictors, and subsequently build a model out of them without another computer or measurement instrument.

Sesame shall incur low overhead and complexity because it is intended to run when the system is being used. To achieve this, Sesame schedules the computation intensive tasks to run in system idle period when it is on AC power supply. Therefore, only data collection and some simple calculation are performed during system usage.

Sesame should achieve higher accuracy and rate than the battery interface. To serve a broad range of applications of energy models, Sesame should achieve high accuracy at rates as high as 100 Hz. It utilizes several techniques to overcome the limitations of battery interfaces.

Sesame must adapt to changes either in hardware or usage. Sesame monitors the accuracy of the energy model in use and adapts it accordingly when the accuracy is below a certain threshold.

Figure 5.1 presents the architecture of Sesame with three major components: data collector, model constructor, and model manager, described in detail below.

5.1.2 Data Collector

The Sesame data collector interacts with the native OS to acquire predictors for energy modeling. Only the lowest layer of the data collector is platform-specific. The data collector considers the following system behavior data as widely used in existing system energy modeling work.

First, Sesame uses system statistics, such as CPU timing statistics and memory usage, supported by modern OSes. For example, Linux uses the sys and proc file system to provide information about processor, memory, disk, and process, and the dev file system to provide information of peripherals, e.g., network interface card. Sesame further utilizes the Advanced Configuration and Power Interface (ACPI) available on modern mobile systems. ACPI provides platform-independent interfaces for the power states of hardware, including

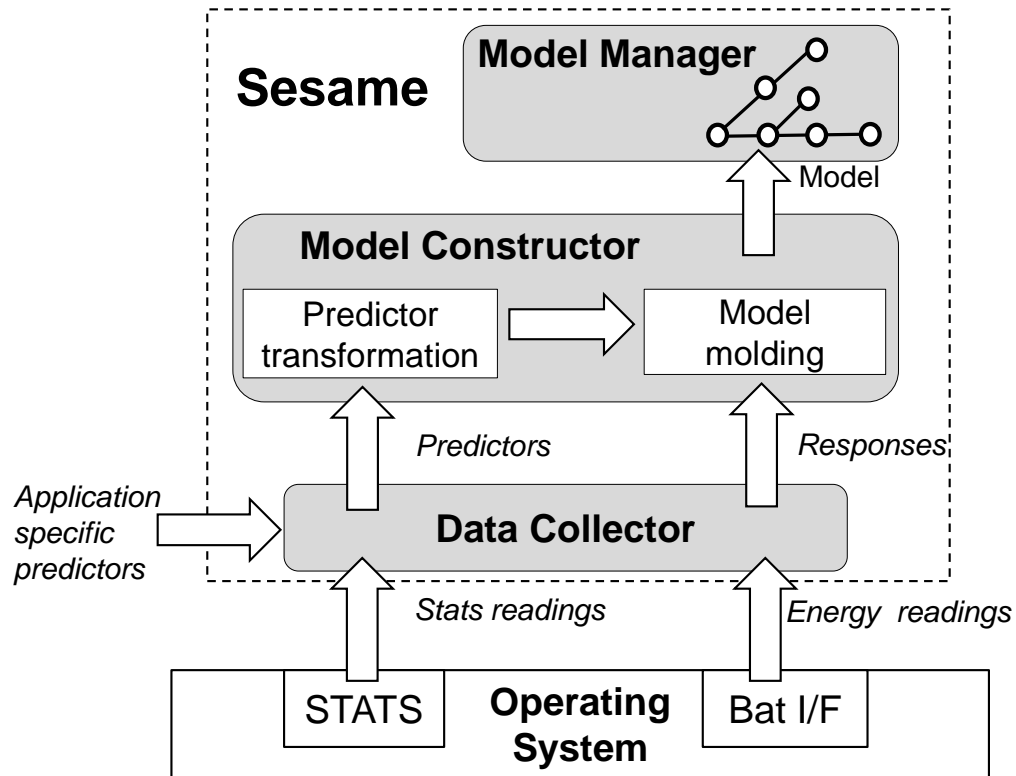


Figure 5.1 : Architecture of Sesame with three major components: data collector, model constructor and model manager.

the processor and peripherals. In Linux, the ACPI information can be found in `/proc/acpi/`. Thanks to ACPI, Sesame is able to access the power status of many system components just by reading a file. In T61, for example, the predictors include CPU P-state residency (P0-P5), CPU C-state residency (Core 0: C0-C3, Core 1: C0-C3), retired instruction number, L1/L2 cache misses, free memory percentage, disk traffic in bytes, wireless interface traffic in bytes, battery level, and LCD backlight level.

It is important to note that Sesame can incorporate any predictor that can be acquired in software. An application can supply its own internal “knob” as a predictor so that Sesame can correlate the system energy consumption with the knob.

Error in Predictors

Sesame must overcome the errors in predictor readings in order to achieve a high accuracy at a high rate. There are two major sources of errors that are more significant when reading at a higher rate.

First, some predictors are updated slowly by the OS. When the update rate is close to the reading rate, errors will occur. For example, the processor P-state residency is updated by increasing one at every kernel interrupt in Linux, 250 Hz or once every 4 ms by default. When predictors are collected at 100 Hz or once every 10 ms, there can be an error as high as 20%. Moreover, for some predictors, there exists a delay from when the predictor values are updated to when corresponding system activities are actually performed. Such predictors include the ones related to I/O operations such as disk traffic and wireless interface traffic. For example, when the OS issues a write request to the disk, the system statistics indicating sectors written to disk is immediately updated, while the write operation may be deferred by the disk driver.

Sesame employs the *total-least-squares* method [41], a variant of linear regression, to overcome the impact of predictor errors. While a normal linear regression minimizes the mean squares of residuals only in responses, the total-least-squares method minimizes the total squares of residuals in both responses and predictors. We use Figure 5.2 to illustrate this important difference. For clarity, Figure 5.2 assumes there is only one predictor x . For the j -th data point, the residual in the response is $r_j = E_j - f(x_j)$ and the shortest distance between the data point and the regression curve is d_j , which is r_j/a if the regression curve is $E = a \cdot x + b$. Regular regression seeks to identify $E = f(x)$ such that the total squares of the response residuals $\sum_j r_j^2$ is minimized. In contrast, the total-least-squares method seeks to minimize the total squares of the distances $\sum_j d_j^2$. Intuitively, the total-least-squares method can be viewed as minimizing the summation of the square of an ad-

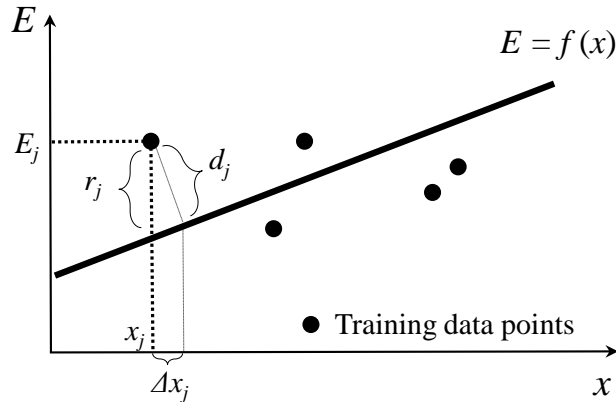


Figure 5.2 : Illustration of the total-least-squares method using linear regression with a single predictor x as example. The total-least-squares method minimizes the total squares ($\sum_j d_j^2$) instead of ($\sum_j r_j^2$) as regular regression does.

justment Δx_j made by the predictor and the square of the resulting residual $E_j - f(x_j + \Delta x_j)$. Numerical algorithms such as the QR-factorization and the Singular Value Decomposition (SVD) are well-known to be effective to solve this problem [41]. Our experiments show that the error can be reduced by about 20% by using the total-least-squares method.

Overhead Reduction

There is overhead for accessing the predictors. This is particularly true if Sesame is implemented out of the kernel space and each access will incur a few system calls, taking several hundred microseconds. Such overhead, along with the overhead of reading the battery interface, sets the limit of the accuracy and rate of the energy model generated by Sesame. To reduce this overhead, Sesame employs two special design features.

First, Sesame treats predictors differently based on their update rates. For predictors that are updated faster than or comparable to the target rate of energy model, Sesame ac-

tively polls them at the target rate. These predictors include processor timing statistics and memory statistics, etc. For predictors that are updated much slower, Sesame polls them as slow as their update rates. These predictors include I/O traffic. There are also a group of predictors that are changed by user interactions such as display brightness, speaker volume, WiFi on/off switch and so on. Sesame leverages the system events produced by their updates to read them only when there is a change.

Second, Sesame implements a new system call that is able to read multiple data structures in the OS at the same time. This bundled system call also takes an input parameter that masks the predictors of no interest. This system call is one of the very few Sesame components that are platform-specific.

We note that the overhead discussed above is also true for all existing system energy modeling methods. The only additional data Sesame collects is the response, i.e., the battery interface reading. This additional overhead is very small as we measured in Section 4.3.3.

5.1.3 Model Constructor

The model constructor is the core of Sesame. It employs two iterative techniques, namely model molding and predictor transformation, as illustrated in Figure 5.1, to overcome the limitations of battery interfaces.

Model Molding for Accuracy and Rate

Model molding involves two steps to improve the accuracy and rate, respectively. The first step, called stretching, constructs a highly accurate energy model of a much lower rate than needed. The rationale is the third finding described in Section 4.3.3 that accuracy is higher for the average battery interface reading of a longer period of time and, therefore,

an energy model of a low rate based on the battery interface is inherently more accurate when there exist systematic errors in battery interface readings such as the low-pass filter effect in T61. For example, Sesame will construct an energy model of 0.01 Hz in the first step of model molding even if the targeted rate is 100 Hz. As shown in Figure 5.3(left), to construct the low-rate energy model, Sesame calculates the energy consumption for a long interval by adding all the readings of the battery interface over the same interval. For example, when the rate is 0.01 Hz (one reading per 100 seconds) and the battery interface is 0.5 Hz (one reading per two seconds), 50 consecutive readings of the battery interface are aggregated to derive the energy consumption for the interval of 100 seconds. The key design issue is choosing a low rate. A very low rate will lead to low error but it will also lead to poor generalization of the produced model, due to reduced coverage by the training data. According to our experience with many systems, we consider 0.02-0.01 Hz a reasonably good range. We should note that we only need to perform model stretching when there exist systematic errors in the battery interface readings such as low-pass filter effect in T61. This is because random errors in the values reported by battery interface will be handled by linear regression.

Since the first step builds a highly accurate but low-rate model, the second step of model molding compresses the low-rate model to construct a high-rate one. Let $E = (1, \mathbf{x}^T) \cdot \beta$ denote the energy model for time intervals of T_L , where $\mathbf{x} = (x_1, x_2, \dots, x_p)^T$ is the predictor vector and $\beta = (\beta_0, \beta_1, \dots, \beta_p)^T$ is the vector of coefficients derived from linear regression. To estimate the energy consumption of a time interval of $T_H \ll T_L$, we simply collect the vector of predictors for T_H , or $\mathbf{x}^T(T_H)$, and use the same β to calculate the energy consumption for the interval as $E(T_H) = (1, \mathbf{x}^T(T_H)) \cdot \beta$.

The rationale for this heuristic is twofold. (i) Ideally, the linear energy model, $E = (1, \mathbf{x}^T) \cdot \beta$, holds for any length of the time interval, T . That is, β should remain the

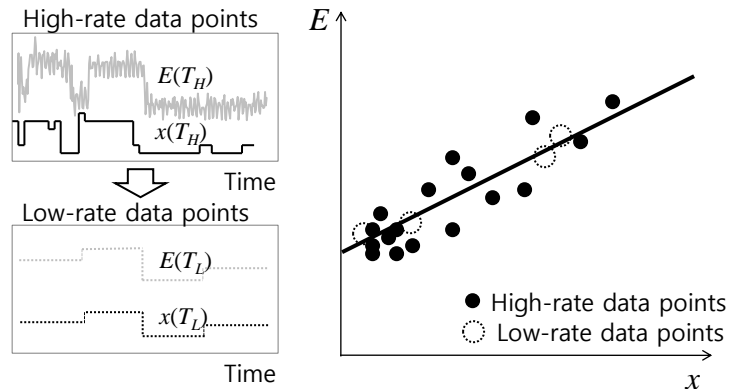


Figure 5.3 : Illustration of model molding. Low-rate data points are generated by averaging high-rate data points (left); linear models generated by two sets of data points are identical (right).

same for time interval of all lengths, as shown in Figure 5.3(right). (ii) $\mathbf{x}^T(T_H)$ can be easily obtained. Many predictors are updated much faster than the response, or the battery interface reading. For example, Linux updates CPU statistics at from 250 to 1000 Hz. For predictors that are not updated in t , their values are regarded as unchanged.

To show the effectiveness of model molding, we perform an experiment with a T61 laptop and the same setup as that used in Section 5.3. We build a linear model of 0.01 Hz, feed it with predictors at various rates, and compare the resulting energy estimation with accurate external measurement. As shown in Figure 5.4, at all rates from 0.01 Hz to 100 Hz, the accuracy of the “molded” model (Molded Model w/o PCA) is always better than the battery interface (Bat I/F). The most improvement is observed for rate of 0.1 Hz to 10 Hz.

Next, we will show that model molding will further improve the accuracy when combined with the second technique: predictor transformation.

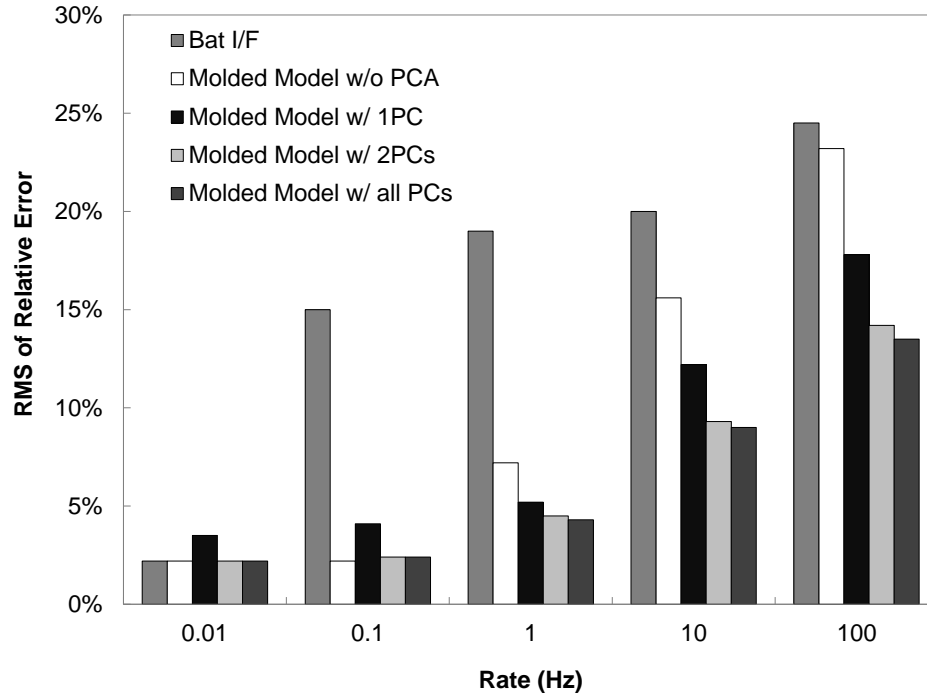


Figure 5.4 : Estimation errors for T61 at different rates and with different selections of predictors through PCA-based predictor transformation.

Predictor Transformation

We find that principal component analysis (PCA) helps to improve the accuracy of a molded model by transforming the original predictors. PCA is not new itself but has never been explored in system energy modeling. It seeks to find a set of linear combinations of the original predictors as independent dominating factors in the predictors for the response. We explain it below.

Given m training data points, each with a response and a predictor vector of p predictors, we can represent the predictor vectors in the training data as an m by p measurement matrix \mathbf{X} . The i -th column of \mathbf{X} is a measurement vector containing m measurements of predictor

x_i . PCA transforms the p predictors as follows. First, a Singular Value Decomposition (SVD) is performed on the transposed matrix of \mathbf{X}^T , i.e., $\mathbf{X}^T = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^*$, where \mathbf{U} is an p by p unitary matrix, $\mathbf{\Sigma}$ is p by m with non-negative numbers on the diagonal and zeros off the diagonal, and \mathbf{V}^* is the conjugate transpose of \mathbf{V} , an m by m unitary matrix. The diagonal entries of $\mathbf{\Sigma}$ are known as the singular values of \mathbf{X} that are ranked based on their values with the highest value on the top. The rows of \mathbf{U}^* , the conjugate transpose of \mathbf{U} and an p by p unitary matrix, are called the singular vectors. The higher a singular value is, the more important the corresponding singular vector is. Each singular vector defines a unique linear function that transforms the original p predictors into a new predictor.

Let \mathbf{U}_l^* denote the first l rows of \mathbf{U}^* , which are the l most important singular vectors. By multiplying \mathbf{U}_l^* and \mathbf{X}^T , one can derive a m by l measurement matrix $\mathbf{Z} = (\mathbf{U}_l^* \cdot \mathbf{X}^T)^T$ for the l transformed predictors with the i -th column being the m measurements of the i -th new predictor. \mathbf{Z} is then used to construct the energy model based on the l transformed predictors, instead of the original ones.

Although there are still debates in the statistic community about whether PCA will help to improve linear regression in general, our results indicate that the l new predictors produced by the PCA do a better job than the original p predictors. We also study the use of different l for energy models of different accuracy. A small l potentially requires fewer original predictors and, therefore, reduces the overhead of data collection, but at the cost of reduced accuracy. In practice, predictor transformation and model molding can be performed iteratively to reduce l while maintaining a required accuracy. Figure 5.4 shows that the molded model using all transformed predictors (Molded Model w/all PCs) ($l = p$) is always better than the one using the original predictors (Molded Model w/o PCA). The impact of l on the model accuracy is also apparent. As shown in Figure 5.4, when $l = 2$, or only top two principal components are used, the accuracy of the model is close to that when

$l = p$. When $l = 1$, the accuracy will be highly reduced, sometimes even worse than the one without PCA. Therefore, Sesame adopts $l = 2$ in the rest of the chapter unless otherwise indicated.

In the T61, the first principal component contains CPU C3 residency, CPU P0 residency, and disk write traffic; the second principal component contains free memory percentage, disk read traffic, and battery level. Interestingly, predictors of wireless interface traffic did not appear in the two principle components. We suspect this is due to the relatively low power consumption of wireless interface compared to the other components of the T61 system.

5.1.4 Model Manager for Adaptability

Because the model constructor may generate multiple models for different system configurations, the model manager organizes multiple energy models in a Hash table whose key is a list of system configurations. And each entry of the Hash table represents the energy model associated with a specific system configuration. Here, system configuration includes three categories: (i) hardware information such as component manufacturer and model; (ii) software settings such as DVS enabled/disabled; and (iii) predictors updated by user interactions such as screen brightness and speaker volume. The model manager keeps the model in the entry of current configuration as the active model. An individual model is comprised of several coefficients and the size is typically less than 1 KB.

The model manager also periodically compares the energy number calculated by the model and the response value to check the accuracy. Since the battery interface is inaccurate at a high rate, the model manager checks the error of a stretched version of the active model. That is, the manager compares a very low-rate energy readings from the active energy model and from the battery interface (~ 0.01 Hz in our implementation). The low-

rate readings are obtained by aggregating the readings over ~ 100 seconds. If the error of the active model is higher than a given threshold, the model manager will start the model constructor to build a new one by including new predictors. Using this procedure, Sesame is able to adapt the model for usage change.

5.2 Implementation of Sesame

We have implemented Sesame for Linux-based mobile systems. The implementation works for both laptops and smartphones using a Linux kernel. The implementation follows the architecture illustrated in Figure 5.1.

Our implementation is developed in C with around 1600 lines of codes and compiled to a 320 KB binary file. The same code works in both platforms without any modification. The mathematical computation such as SVD is based on the GNU Scientific Library, which is about 800 KB in binary. Sesame should work on any Linux-like system that supports GSL. Only the lowest layer of the data collector needs to be adapted for a non-Linux platform.

Sesame can be launched with user specified parameters. In particular, the user can specify rate and preferred predictors to leverage the users prior knowledge. Sesame also provides library routines that can be used by applications in energy management and optimization. To communicate with Sesame, a program should first start Sesame while specifying the rate and the paths of the predictors. Then the program should be able to get the energy estimation of a time interval in the past by providing the start and end points of the interval.

5.2.1 Data Collector

We implement the data collector as three subroutines that are in charge of initial predictor list building, data collection and configuration monitoring, respectively. A file is used to

specify the paths of predictor candidates and configuration files. By extending this library file, one can easily make Sesame work with a new platform. On T61, the data collector uses the battery interface provided by ACPI. The rate of the ACPI battery interface of T61 is 0.5 Hz. On N900, the data collector talks to the HAL layer through D-Bus to access the battery interface at 0.1 Hz. Table 3 provides the initial predictors used for each system.

All the data involved in the computation are stored in a single copy and all the routines use the pointer to gain access to the data. Sesame keeps all the data traces only during predictor transformation and saves them to storage, disk or flash, when it has collected 1000 data entries. Since Sesame collects data once every 10 ms during model construction, the data rate is quite low. Typically, there are about 25 predictors, all 16-bit integers, to collect at each sample, the data rate is only 5 KBps and the maximal size of the data trace in memory at one time is 50 KB.

5.2.2 Model Constructor

We implement the model constructor as a subroutine that is called by the model manager. One technical challenge of implementing the model constructor is to organize data to be used for model construction in an efficient way so that the response can be quickly indexed given a specific set of predictors. Therefore, we choose to use Hash maps to organize all the data. In a Hash map, the key of each entry is a vector comprised of all the predictor being used, and the value of each entry is a pointer of a structure that has two members, i.e., the number and average of the responses corresponding to the predictors. When a new data sample is passed from the data collector, the model constructor uses the predictors to calculate an index of the Hash map, finds the structure linked to the corresponding map entry, and updates the two members of the structure.

The most time consuming computation in constructing a model includes PCA and linear

regression. In our implementation, the total execution time to finish the whole process takes less than one minute on T61 and about ten minutes on N900. Because neither PCA nor linear regression is needed real-time, they can easily be offloaded to the cloud. In our currently realization, they are performed when the system is wall-powered and idle.

5.3 Evaluation

In this section, we report an extensive set of evaluations of Sesame in three aspects: overhead, rate/accuracy, and the capability of adaptation using two Linux-based mobile platforms. We also report the results of two field studies to validate whether Sesame is able to generate energy models without interrupting users. For evaluation, we use the same DAQ system described in Section 4.2 to measure the system power consumption at 100 Hz.

5.3.1 Systems and Benchmarks

We have evaluated Sesame using two mobile systems with representative form factors, laptop (Lenovo ThinkPad T61) and smartphone (Nokia N900). Table 5.1 provides their specifications. As we showed in Section 4.3.3, the T61 battery interface supports current measurement at 0.5 Hz; that of N900 does not support current measurement but current can be inferred at 0.1 Hz. Both battery interfaces are extremely challenging to Sesame due to their low rate, especially that of N900.

We choose the following three sets of benchmarks to cover diverse scenarios in mobile usage: compute-intensive (CPU2006), interactive (BLTK), and network-related (NET). CPU2006 is from SPEC. It stresses a system’s processor and memory subsystem. The Linux Battery Life Toolkit (BLTK) from Less Watts [36] is designed for measuring the power performance of battery-powered laptop systems. Only three of the six BLTK benchmarks (Idle, Reader, and Developer) can run on N900. Finally, we choose three repre-

Table 5.1 : Specifications of platforms used in Sesame evaluation

	T61	N900
Processor	Intel Core 2 Duo T7100 (2 GHz)	TI OMAP 3430 (600 MHz)
Chipset	Intel GM965	N/A
Memory	DDR 2 GB	DDR 512 MB
Storage	Disk 250 GB	Flash 2 GB
Network	WiFi	WiFi/3G
Display	14.1" LCD	3.5" LCD
OS	Linux 2.6.26	Maemo 5

sentative network applications that run on both T61 and N900 as NET. Table 5.2 provides detailed information for BLTK and NET.

In experiments reported in Section 5.3.2-Section5.3.4, we train and evaluate the models generated by Sesame using the same benchmarks. Therefore, the errors we will present are fitting errors as in regression terms. This evaluation is reasonable and also insightful because Sesame is intended to build models based on regular usages that are usually repetitive. Importantly, we complement this evaluation with two five-day filed studies of four laptop users and four smartphone users in which Sesame generates models using real traces in everyday use.

Table 5.2 : Benchmarks used in Sesame evaluation

BLTK	Description
Idle	System is idle while no program is running
Reader	Human actions are mimicked to read local webpages in Firefox
Office	Human actions are mimicked to operate Open Office
Player	Mplayer plays a DVD movie
Developer	Human actions are mimicked to edit program files in VI and then Linux kernel is compiled
Gamer	Human actions are mimicked to play a part of 3D game
NET	Description
Web Browser	Human actions are mimicked to read webpages on internet in Firefox
Downloader	Files of 1GB size are downloaded through SSH
Video Streamer	Videos on YouTube are played in Firefox

5.3.2 Overhead

We next show that the overhead of Sesame is negligible. Figure 5.5 shows the overhead breakdown of self-modeling in terms of time taken for each critical stage, including response collection, predictor collection and model calculation. Since complicated computation such as PCA and linear regression is scheduled when the system is idle and being

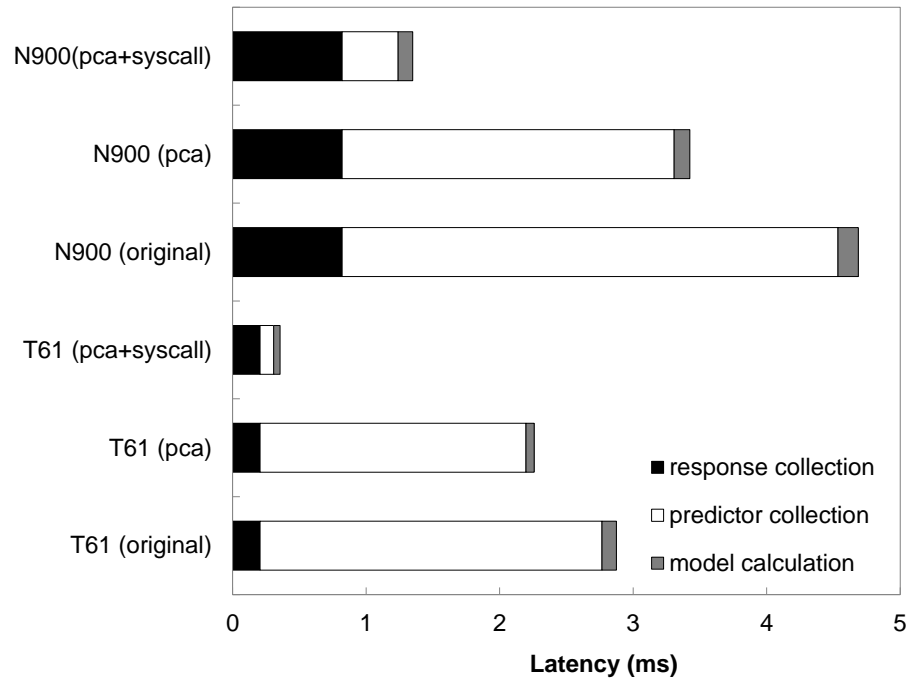


Figure 5.5 : Overhead of Sesame in terms of execution time. Predictor collection is most expensive among the three steps. PCA and the bundled system call (syscall) help reduce the overhead of predictor collection significantly.

wall-powered, the model calculation only includes simple operation such as linear transformation of predictors into principal components. As shown in the figure, the total computation time of self-modeling is about $3000 \mu s$ and $5000 \mu s$ for T61 and N900, respectively, if no optimization technique is used. When the data is collected at 100 Hz, the overhead would account for about 30% and 50% of the period on T61 and N900, respectively, which is too high for the system.

We further show how the design of Sesame helps keep the overhead low. While predictor collection dominates the overall overhead, it could have been much worse without Sesame. First, predictor transformation will reduce the number of predictors such that over-

head will be reduced. Second, most importantly, a bundled system call that obtains all the predictor values at one time will significantly reduce the time in predictor collection. As shown in the figure, the optimized overhead of computation time of self-modeling is about $300 \mu\text{s}$ and $1200 \mu\text{s}$ for T61 and N900, or about 3% and 12% of the period on T61 and N900, respectively. We note that the numbers above are in the worst case when response collection is performed. In actuality, response collection is scheduled to operate every 2 seconds and 10 seconds on T61 and N900, respectively. Therefore, the typical overhead of Sesame is only 1% and 5%, respectively.

The contributions of the delay to the energy consumption can be estimated as roughly the same or slightly larger percentage-wise. Such low overhead explains the effectiveness of Sesame. As shown in Figure 5.5, the overhead of Sesame is higher for N900 than for T61. The main reason is the rate of accessing various system statistics, for both predictors and responses, is much lower on N900 than on T61. Because the system consumes extra energy for Sesame data collection, the larger overhead of N900 contributes its higher error, as will be shown later.

5.3.3 Accuracy and Rate

We next show how Sesame improves the accuracy and rate with model molding and predictor transformation.

Figure 5.6 shows the tradeoff curves of accuracy (in terms of the RMS of relative errors) and rate (in terms of the reciprocal of the length of the time intervals for energy estimation) for the model generated by Sesame for T61 and N900. For comparison, we also put in the tradeoff curve for the model generated by feeding Sesame with accurate external measurement at a sampling rate of 100 Hz. This can be also viewed as the best tradeoff curve that Sesame can possibly achieve. The relative error of each sample is calculated by comparing

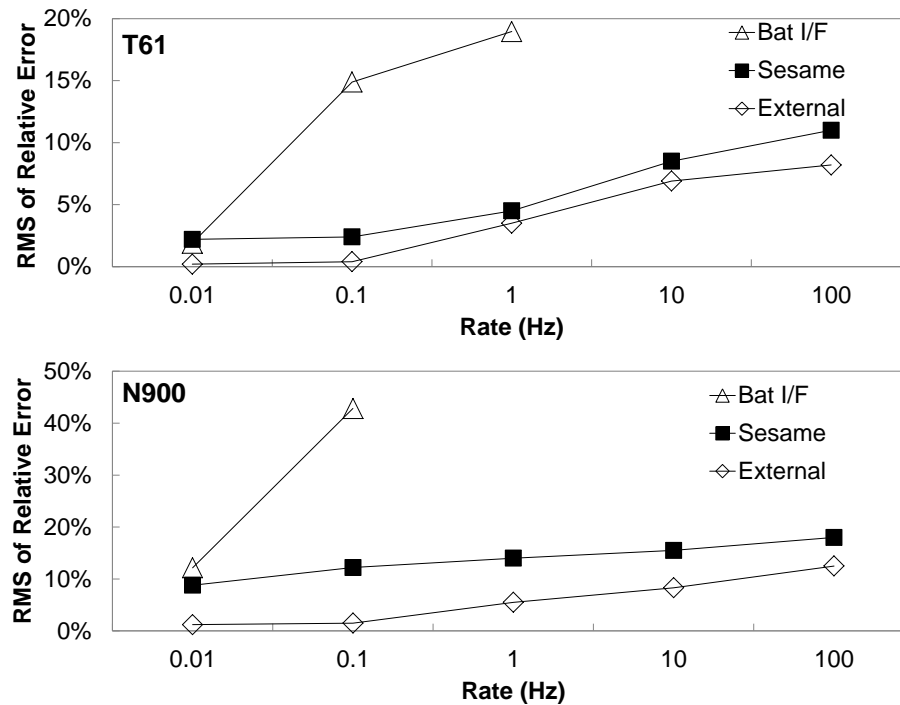


Figure 5.6 : Model molding improves accuracy and rate. Sesame is able to increase the model rate to 100 Hz.

the estimation by an energy model against the accurate external measurement.

As shown in Figure 5.6, the accuracy of the model generated by Sesame on T61 is 95% at 1 Hz, which is much better than the battery interface reading at the same rate. Also, Sesame is able to achieve an accuracy of 88% at 100 Hz using the battery interface readings at a rate of only 0.5 Hz. On N900, Sesame achieves accuracy of 86% and 82% at 1 Hz and 100 Hz, respectively, using the battery interface with an accuracy of only 55% at 0.1 Hz. Such results highlight the effectiveness of model molding, predictor transformation, and overhead reduction by Sesame.

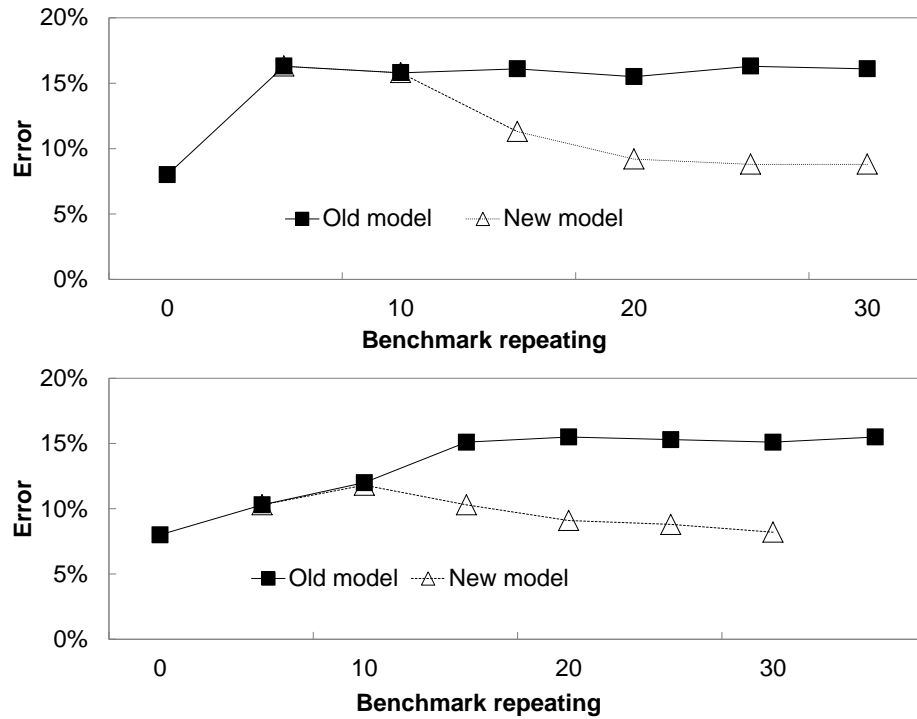


Figure 5.7 : Model adaptation to (top) system configuration change and (bottom) usage change. Error threshold is 10%.

5.3.4 Model Convergence and Adaptation

Sesame is able to adapt the energy model to the change of configuration or usage of the computing system. We devise two experiments to test how Sesame adapts to the two types of changes with the error threshold set to be 10%. The error reported in this section is monitored by Sesame and calculated based on a rate of 0.001 Hz.

In the first experiment, the T61 is running the Developer benchmark repeatedly. At the beginning, the DVS is disabled on T61 and an energy model is with accuracy 92% at 1 Hz, as shown in Figure 5.7 (top), has been established that does not include predictors of P-state residency. Then we enable DVS, which increases the error to 17% and triggers Sesame

to build a new model. In the next ten executions of the benchmarks, Sesame performs predictor transformation and includes P1 residency as an additional predictor. Finally the new energy model is able to reduce the error below the threshold (10%).

In the second experiment, T61 first runs the Developer benchmark and an energy model with accuracy 92% at 1 Hz, as in experiment one. Then we switch the benchmark to the Office and the error monitored by Sesame gradually becomes worse, as in Figure 5.7 (bottom). When the error exceeds the error threshold (10%), Sesame starts building a new energy model, which also eventually leads to an error below the threshold.

We also validate the adaptability of Sesame on N900 and observe similar results.

5.3.5 Field Study

We performed two field studies to evaluate if Sesame is able to build an energy model without affecting normal usage and with dynamics of real field usage. The first field study involved four participants using Linux laptops regularly. The four laptops were owned by the participants and had diversified hardware specifications, as listed in Table 5.3. The laptops, however, all had a battery interface similar to that of T61 described in Section 4.3, i.e., with a rate of 0.5 Hz and the low-pass filter effect. The second study involved another four participants who are smartphone users. Each participant was given a Nokia N900 and was asked to use the N900 as his/her primary phone for a week with his/her own SIM card in. All eight participants were ECE and CS graduate students from Rice University. The results clearly show that Sesame is able to (i) build energy models for different computing systems based on platform and usage; (ii) converge in 16 hours of usage on laptops and 12 hours of usage on N900; (iii) provide energy models with error of at most 12% at 1 Hz and 18% at 100 Hz for laptops and 11% at 1 Hz and 22% at 100 Hz for N900, which is comparable with state-of-the-art system energy modeling methods based on high-speed

external measurement. Most importantly, Sesame achieves so without interrupting users.

Table 5.3 : Specifications of laptops used in field study

	User1	User2	User3	User4
System	Lenovo Thinkpad T60	Lenovo Thinkpad T61	Lenovo Thinkpad Z60m	Lenovo Thinkpad X200s
Processor	Intel Core 2 Duo T5600 (1.8 GHz)	Intel Core 2 Duo T7300 (2.0 GHz)	Intel Pentium M (1.6 GHz)	Intel Core 2 Duo L9400 (1.8 GHz)
Memory	DDR2 2 GB	DDR2 4 GB	DDR2 1 GB	DDR3 4 GB
Storage	Disk 80 GB 5400 rpm	Disk 250 GB 5400 rpm	Disk 60 GB 5400 rpm	Disk 160 GB 7200 rpm
WIC	Intel 3945ABG	Intel 4965AGN	Intel 4965AGN	Intel 5300AGN
OS	Linux 2.6.31	Linux 2.6.26	Linux 2.6.31	Linux 2.6.31

Procedures

Before the study, we informed the participants that the purpose of the field study was to test the properties of their batteries and that they should use the mobile systems, their laptops or N900s, as usual. We installed Sesame on their mobile systems. To evaluate whether Sesame will affect the user experience, we set Sesame work in the first four days and stop

in the last day without telling the participants. Thus, Sesame generated two energy models, with rates of 1 Hz and 100 Hz for the mobile system during the first four days. On the sixth day, we asked each participant to bring his/her mobile system to the lab and set up a high-speed power measurement with it. Then we asked the participants use their mobile systems for 30 minutes during which Sesame was estimating energy based on the model it generated in the past five days.

We asked the participants to give a score from 1 to 5 to their experience of each day in the five-day study regarding delay in responses. 5 means most satisfied for each of the five days in the field.

Results

Figure 5.8 shows the model evolution during the modeling process. The error is calculated based on the average power readings from the battery interface in 100 seconds for T61 and 1000s for N900. As shown in the figure, the models of all four laptop users converge in 16 hours and the convergence time for all N900 users is 12 hours. We compare the error of models generated by Sesame. As shown in Figure 5.9, Sesame provides energy estimations with error of at most 12% at 1 Hz and 18% at 100 Hz for laptops and 12% at 1 Hz and 22% at 100 Hz for N900.

And the results show that all the N900 users and three of the laptop users give a straight 5 for all the five days in the field study, indicating that they did not notice any interruption in user experience. The only one participant (laptop User 2) gives two 4s for the first two days in the field study, i.e., predictor transformation phase. The reason for the delay is he was running computation intensive applications while the initial predictor set included ones related with performance counters which are expensive to access.

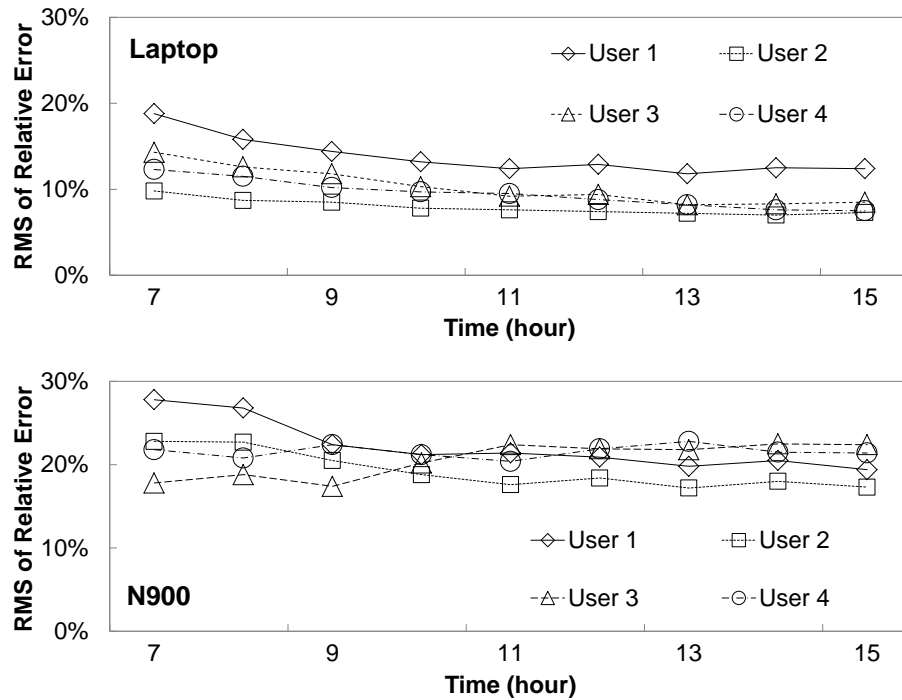


Figure 5.8 : Errors of the models generated in the field study evolve in the first five days. Errors are calculated relative to the averaged readings from battery interfaces in every 100 seconds in laptops and in every 1000 seconds in N900s.

5.4 Summary

In this chapter, we demonstrated the feasibility of self energy modeling of mobile systems, or Sesame, with which a mobile system constructs an energy model of itself without any external assistance. We reported a Linux-based prototype of Sesame and showed it achieved 95% accuracy for laptop (T61) and 86% for smartphone (N900) at a rate of 1 Hz, which was comparable to existing system energy models built in lab with the help of a second computer. Moreover, Sesame was able to achieve 88% and 82% accuracy for T61 and N900, respectively, at 100 Hz, which was at least five times faster than existing

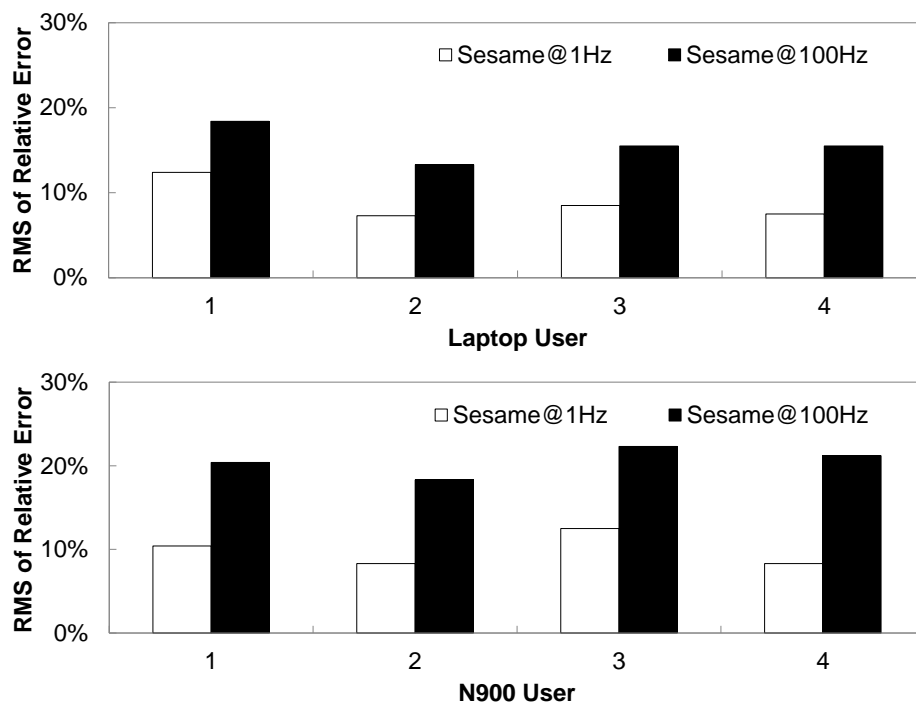


Figure 5.9 : Comparison with readings from battery interfaces. Errors are calculated relative to accurate external measurement.

system energy models. Our field studies with four laptop users and four smartphone users further attested to Sesame's effectiveness and non-invasiveness. The high accuracy and high rate of system energy models built by Sesame allows for innovative applications of system energy models, e.g., energy-aware scheduling, rogue application detection, and incentive mechanisms for participatory sensing, on a multitasking mobile system.

Finally, although Sesame was originally intended for mobile systems, the self-modeling approach is readily applicable to other systems with power measurement capability, e.g., servers with a power meter inside the power supply unit. The overhead reduction and accuracy/rate improvement techniques of Sesame will be instrumental.

Chapter 6

***In Situ* $E(\mathbb{S})$ Estimation using Improved Battery Interface**

In this chapter, we provide a system solution that is capable of *in situ* power measurement with 200 samples per second, or 200 Hz, and with an accuracy over 95%. We further provide two algorithms to estimate all the $E(\mathbb{S}, \sigma)$ based on partial information. Building on this foundation, we are able to implement both Shapley value based energy accounting (Section 6.4) and OEM based scheduling (Section 6.4.3) and evaluating them experimentally. Using this prototype and smartphone workload, we experimentally demonstrate how erroneous existing energy accounting policies can be, how little difference in energy management can they make, and show that OEM outperforms BEM in scheduling by 20%.

6.1 Improved Battery Interface

6.1.1 Architecture

Our design includes both software in the mobile system and an improved battery interface as illustrated by Figure 6.1. The battery interface is improved by adding an ultra low-power 16-bit micro-controller (MCU) and a local memory, which is often available inside the MCU. The MCU periodically measures the voltage of and the discharge current from the battery using two ADCs, calculates the power consumption, and writes it to the local memory along with a timestamp.

The driver running in the mobile system reads the local memory in the improved battery interface via a serial bus in the standard smart battery interface upon a request from the OS.

The driver also keeps a timestamped record for the Android UID of applications that are running in the application processor concurrently at each OS scheduling period as well as the hardware state of the system. Then the OS combines the timestamped power data from the battery interface and the local timestamped UID data to obtain the system energy consumption for each scheduling period.

Note that we use *application* instead of *process* as the minimum unit in correlating power measurement data with software activities because the number of processes are much higher than the number of applications. According to a recent study [42], the top ten most frequently used applications account for the total usage time by as much as 90%. Therefore, we choose to only consider the top ten applications for each user and treat other applications as a single application “other”. Combining the Android system that includes all the system processes, there are at most 12 applications to be accounted in our implementation.

6.1.2 Measurement Requirement

The key component of the improved battery interface in Figure 6.1 is the ADC. The most important properties of an ADC are *sampling rate* and *resolution*. Insufficient sampling rate and resolution will cause error in power measurement while excessive high sampling rate and resolution will lead to significant power waste.

In order to determine the sufficient sampling rate and resolution, we characterize the power consumption traces of three commercial smartphones (Samsung Galaxy S, S II, and S III) and ten popular applications with diverse properties. Table 6.1 and Table 6.1 list the specifications of the smartphones and the descriptions of the benchmark applications, respectively. We employ one external power supply for constant 4.2 V input and measure the current drawn. We employ a high-speed Oscilloscope to collect the ground truth power traces, using 500 KHz sampling rate, for each of the device and application.

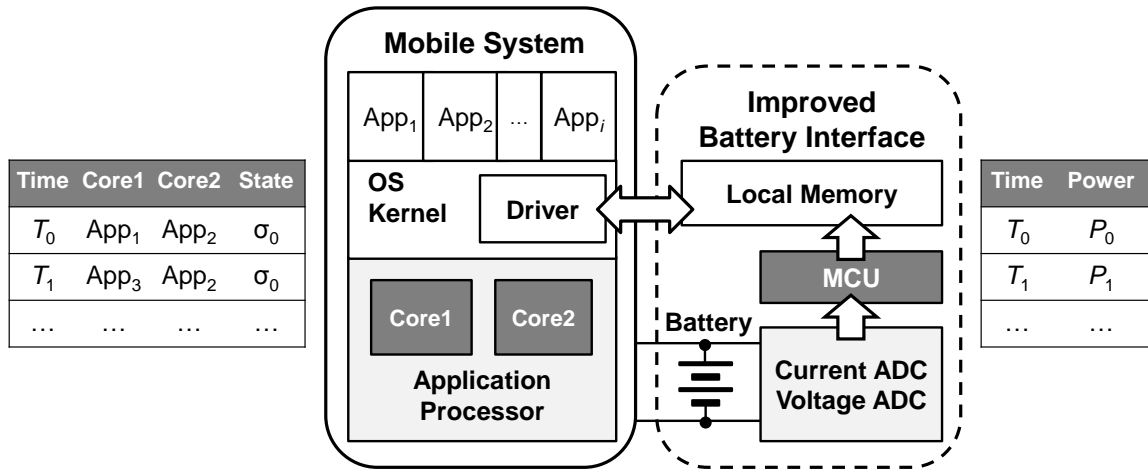


Figure 6.1 : Architecture of the *in situ* high-speed, high-accuracy power measurement solution: The battery interface measures power periodically and stores the measurement data with timestamps in its local memory; the mobile system also maintains a timestamped data structure of application combination executed in the OS kernel, utilizes a driver to read the energy measurement data structure from the local memory in the battery interface, and combines the two data structures to get all the $E(\mathbb{S})$.

Sampling Rate: By examining the spectrum of the power traces, we are able to identify the sufficient sampling rate for each application to include most of power density in spectrum, e.g., 99%.

As an example, Figure 6.2 (left) shows the power spectrum of Galaxy S III running the Phone application. We can see most of energy locates in the low-frequency range, while the high-frequency spectrum is flat. As shown in the black curve in the figure, 200 Hz sampling rate includes over 99% of the signal energy. The power traces of other applications show similar characteristics. Note that our goal is to measure the energy consumption in each scheduling period, or 10 ms. We next use the 500 KHz data traces as the ground truth and

Table 6.1 : Smartphone Specifications

Model	Galaxy S	Galaxy S II	Galaxy S III
Processor	1 GHz 1-core	1.2 GHz 2-core	1.4 GHz 2-core
Memory	512 MB	1 GB	2 GB
Network	ATT HSPA	Tmobile HSPA+	Verizon LTE
Android	2.3.6	2.3.6	4.0.4

Table 6.2 : Sufficient sampling rate for various smartphone applications(Unit:Hz)

Application	Galaxy S	Galaxy S II	Galaxy S III
Phone	197	185	200
SMS	97.1	88.5	93.3
Web Browser	102	105	95.4
Facebook	107	95.5	102
Google Hangout	87.2	85.3	80.8
YouTube	97.4	91.6	102
CamCorder	18.9	24.1	15.1
GPS Navigation	32.5	46.1	30.7
Quake III	67.2	85.6	90.8
Angry Bird	28.4	45.5	57.2

investigate the average error in each 10 ms introduced by using a sampling rate less than 500 KHz. Figure 6.2 (right) shows the three applications with highest errors. As shown in the figure, a 200 Hz sampling rate introduces a less than 1% error at most. Therefore, we conclude that 200 Hz sampling rate is sufficient for all the characterized applications on all

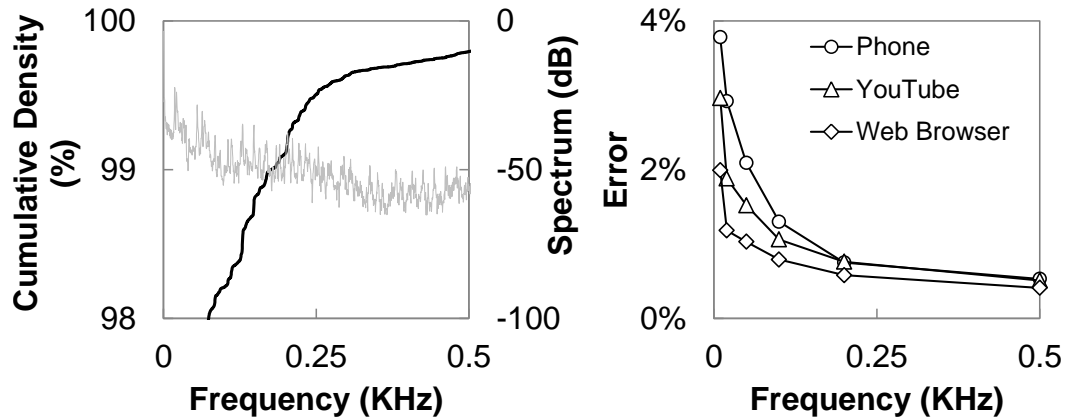


Figure 6.2 : Smartphone workload power characterization for sampling rate: 200 Hz sampling rate is able to cover 99% of the signal information of the power trace (left); 200 Hz sampling rate introduces 1% error in energy measurement of each 10 ms (right).

the smartphones in our experiment.

Resolution: Resolution of an ADC is determined by the dynamic range of the input signal. We characterize the dynamic range by reducing quantization noise below the background noise. As shown in Figure 6.3 (left), background noise is around -100 dB in normalized spectrum, which implies the required effective number of bit $ENOB$ is not less than 16 by referring to $SNR = 1.76 + 6.02 \times ENOB$. Thus, a 16-bit ADC is sufficient to record transient power trace. We next study how much error will be introduced by using a lower resolution ADC in the energy consumption in each 10 ms. As shown in Figure 6.3 (right), the quantization error introduced by using a 8-bit ADC only introduces 0.1% error in each 10 ms.

To summarize, an 8-bit ADC with 200 Hz sampling rate is sufficient for the power measurement needed by energy accounting and OEM. Thanks to fast development in low-power CMOS technology, a 15-bit audio ADC, with sampling rate higher than 20 KHz,

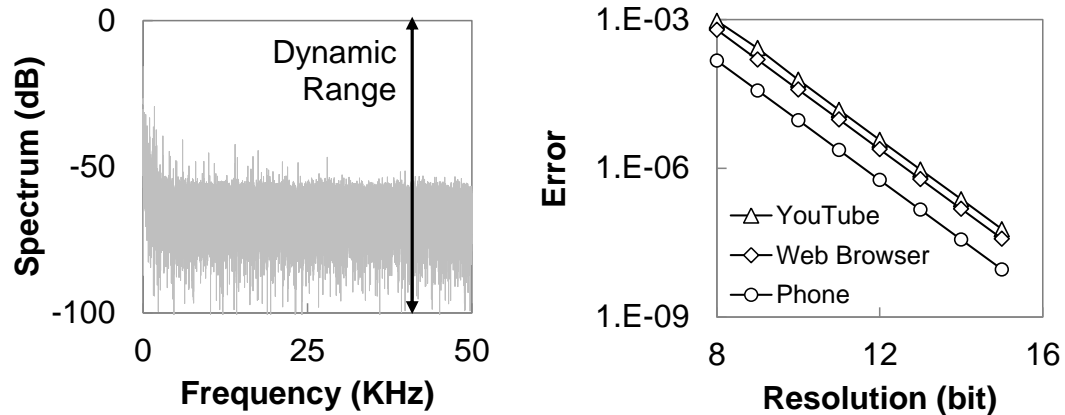


Figure 6.3 : Smartphone workload power characterization for resolution: dynamic range of power trace is around 100 dB and requires a 16-bit ADC for measurement (left); using a 8-bit ADC introduces 0.1% error in energy measurement of each 10 ms (right).

only consumes less than $200 \mu\text{W}$ [43]. The power consumption of a 8-bit ADC with 200 Hz sampling rate should be negligible.

6.1.3 Synchronization

Another key design of our solution is the synchronization of the process data collected by the driver and the power data collected by the battery interface. The process data collected by the mobile system is timestamped with a timer based on a high accuracy clock adjusted by the base station. The power data, however, is timestamped by the timer based on a local low-accuracy clock, usually implemented using a LC oscillator to achieve power efficiency. Inaccuracy in the power data timestamp will lead to energy measurement misalignment with the scheduling periods of the mobile system. For example, when clock skew is 5 ms, or 50% of the OS scheduling period, the energy measurement error can be as high as 15%. To tackle this challenge, we employ the following two techniques.

Timer Sharing: The driver periodically writes the timestamp from the OS to a specific memory location in the battery interface and the latter utilizes it to correct its own timer. Thus, the OS timer and the battery interface timer are only different by the delay of a data transfer from the OS to the battery interface, which can be further corrected by the next technique.

Data Transfer Delay Calibration: Data transfer delay is defined by the time between the moment when the driver sends a read/write request to the battery interface and the moment when the battery interface receives the read/write request. Figure 6.4 shows the calibration procedures. In the calibration, the driver (1) disables all the interrupts in the mobile system to minimize interference; (2) writes the current timestamp obtained from the driver to a specific memory location in the battery interface; (3) reads the timestamp from the same memory location from the battery interface; (4) calculates the difference between the current timestamp and the timestamp read from the battery interface; and calculates the data transfer delay by dividing the difference by a factor of two.

6.2 Estimation based on Partial Information

All the $E(\mathbb{S}, \sigma)$ can be represented in a two-dimension matrix, with only a part of the all the elements can be obtained in historical measurement data as described in Section 6.1. We employ two solutions to address such partial information problem. As shown in Figure 6.5, we use *recursive definition* to fill all the columns and *linear estimation* to fill all the rows. One need to choose between the two solutions based on how the matrix in Figure 6.5 has been filled by historical measurement data.

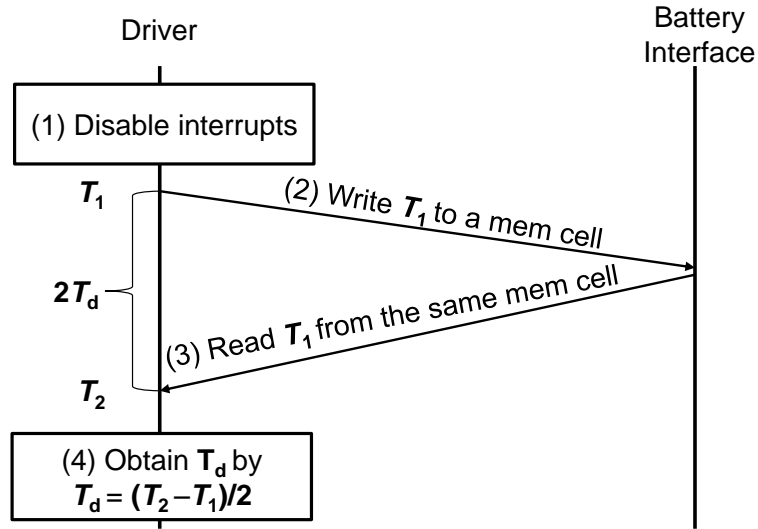


Figure 6.4 : Data transfer delay calibration.

6.2.1 Recursive Definition

When known energy costs are sparse, it becomes difficult to use the method above to estimate unknown coalitional energy costs. We propose an alternative method to recursively define Shapley value for all partially defined games. The key idea is to approximate unknown coalitional energy costs by their per-process energy cost allocations, which are presumably solvable by Shapley value. We introduce the following extension of a partially defined game E to a fully defined one:

$$\hat{E}(S) = \begin{cases} E(S) & \text{if } S \text{ is known} \\ \sum_{i \in S} \phi_i(\hat{E}) & \text{otherwise} \end{cases} \quad (6.1)$$

Here $\phi_i(\hat{E})$ is the Shapley value of the fully defined game \hat{E} , corresponding to per-process energy cost for application i . By extending E to \hat{E} , we are assuming that the energy cost of an unknown coalition is approximately the sum of those individual processes that composes the coalition. To complete the definition, $\phi_i(\hat{E})$ is the standard Shapley value for

	σ_1	σ_2	...	σ_y	...	σ_K
\mathbf{S}_1	$E(\mathbf{S}_1, \sigma_1)$	Linear Estimation				$E(\mathbf{S}_1, \sigma_K)$
\mathbf{S}_2	Recursive Definition					
...						
\mathbf{S}_x						
...						
\mathbf{S}_L	$E(\mathbf{S}_L, \sigma_1)$					

Figure 6.5 : Estimation from partial information. Columns are filled by recursive definition and rows are filled by linear estimation.

\hat{E} as defined by

$$\phi_i(\hat{E}) = \sum_{\mathbb{S} \subseteq \mathbb{N} \setminus \{i\}} \frac{\hat{E}(\mathbb{S} \cup \{i\}) - \hat{E}(\mathbb{S})}{(|\mathbb{N}| - |\mathbb{S}|) \binom{|\mathbb{N}|}{|\mathbb{S}|}}. \quad (6.2)$$

We note that the definition above is recursive since \hat{E} and $\phi_i(\hat{E})$ are defined by each other. Therefore, we propose an iterative algorithm to compute Shapley value $\phi_i(\hat{E})$. At each iteration, we compute $\phi_i(\hat{E})$ for the fully defined game \hat{E} , and whenever the computation needs to probe an unknown coalition \mathbb{S} , then $\hat{E}(\mathbb{S})$ is calculated using Shapley value $\phi_i(\hat{E})$ obtained in previous iteration.

6.2.2 Linear Estimation

We further utilize a linear model to estimate unknown coalitional energy costs for different $E(\mathbb{S}, \sigma_y)$ with the same coalition \mathbb{S} but different power states σ_y , $y = 1, 2, \dots, K$. Suppose that an unknown energy cost $E(\mathbb{S}_x, \sigma_y)$ of coalition \mathbb{S}_x needs to be estimated for the set of resulting feasible coalitions to form a desired structure. We can employ the least-square

method to find a linear substitute for σ_y based on the energy costs of other coalition in the same states, e.g., $E(\mathbb{S}_l, \sigma_y)$, $l \in 1, 2, \dots, L$ and $E(\mathbb{S}_l, \sigma_k)$, $k = 1, 2, \dots, K$, $k \neq y$. More precisely, we derive $K + 1$ parameters $\theta_0, \theta_1, \dots, \theta_K$ by solving the following optimization problem:

$$\text{minimize} \quad \sum_{l:l \neq x} \|E(\mathbb{S}_l, \sigma_y) - \sum_{k=1; k \neq y}^K \theta_k \times E(\mathbb{S}_l, \sigma_k)\| \quad (6.3)$$

Then we can calculate $E(\mathbb{S}_x, \sigma_y)$ by

$$E(\mathbb{S}_x, \sigma_y) = \sum_{k=1; k \neq y}^K \theta_k \times E(\mathbb{S}_x, \sigma_k). \quad (6.4)$$

6.3 Implementation

We have implemented a prototype of the rapid *in situ* energy measurement solution using a Texas Instruments (TI) PandaBoard ES and a MAXIM DS2756 battery fuel gauge evaluation module (EVM), as shown in Figure 6.6. Our solution is able to measure system energy consumption at 200 Hz with only 400 μ W overhead.

In our prototype, the PandaBoard ES serves as a mobile system and is powered by an external battery pack. It comes with a TI OMAP4460 application processor, similar to that used by the Galaxy Nexus smartphone and RIM Playbook, and Android 4.0.4 running on it. We implement the improved battery interface using the DS2756 EVM customized by MAXIM and use it to measure power consumption drawn by the Pandaboard ES from the external battery pack. The DS2756 EVM comes with two 14-bit ADCs that measure the voltage and discharge current, respectively. The measurement data is transferred from the memory in DS2756 EVM to the PandaBoard ES through GPIO using the 1-wire communication protocol.

The $E(\mathbb{S}, \sigma)$ collection module running in the Pandaboard ES includes two parts, i.e., a kernel modification to align timestamp and power measurement and a user-space thread to

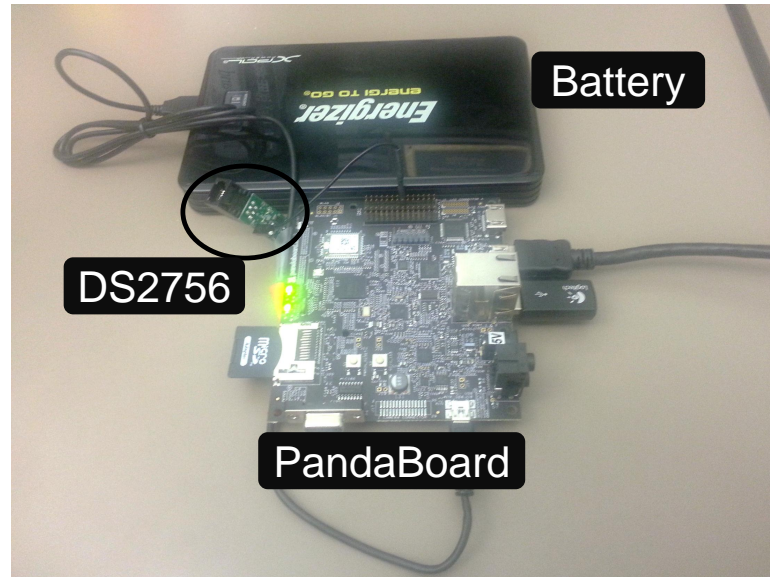


Figure 6.6 : Prototype: a TI PandaBoard ES and a MAXIM DS2756 battery fuel gauge evaluation module.

collect data. We modify the Android kernel to include the timestamped data structure (See Figure 6.1) in the kernel space. Each time when process context switch happens, the OS updates the data structure by appending a new entry to it. Such data structure is exposed to the user-space thread in the `/sys` virtual file system. The user-space thread periodically reads the timestamped data structure from the virtual file system before the data structure overflows and stores the measurement data in SD card. When the measurement data in SD card accumulates to a certain amount, the user-space thread launches a routine to estimate unknown $E(\mathbb{S}, \sigma)$ from historical measurement data, as illustrated in Figure 6.5.

6.4 Evaluation

Using the prototype described in Section 6.3, we next report our implementation of Shapley value based energy accounting and evaluation of existing energy accounting policies

against it.

In our experiment, we choose four different applications to run on our prototype:

- Download: Network intensive process that downloads a large file using HTTP protocol via WiFi.
- Web: Android built-in web browser application to visit web pages as recorded in a URL list via WiFi.
- Video: Video player based on ffmpeg.
- Game: Open source 3D video game Quake 3. The inputs are recorded in a script.

Out of the four applications, Video, Web and Game cannot run at the same time because all of them require a foreground execution to present the content on the screen. Therefore, we further define three scenarios that represent three typical use cases with multiple applications concurrently running:

- Scenario 1: Web + Download + Android.
- Scenario 2: Video + Download + Android.
- Scenario 3: Game + Download + Android.

6.4.1 Accuracy of $E(\mathcal{S})$

We first evaluate the accuracy of energy measurement by the prototype by comparing the measurement results against readings by a high-end Oscilloscope. As shown in Figure 6.7 (left), over 90% of measurement are with error less than 5%. Figure 6.7 (left) also illustrates the importance of synchronization. As shown in the figure, 90% of measurement are with error less than 30% without synchronization.

We then evaluate the accuracy of energy estimation from partial data. The prototype support 12 states in total, i.e., four states for CPU and three states for WiFi. There are

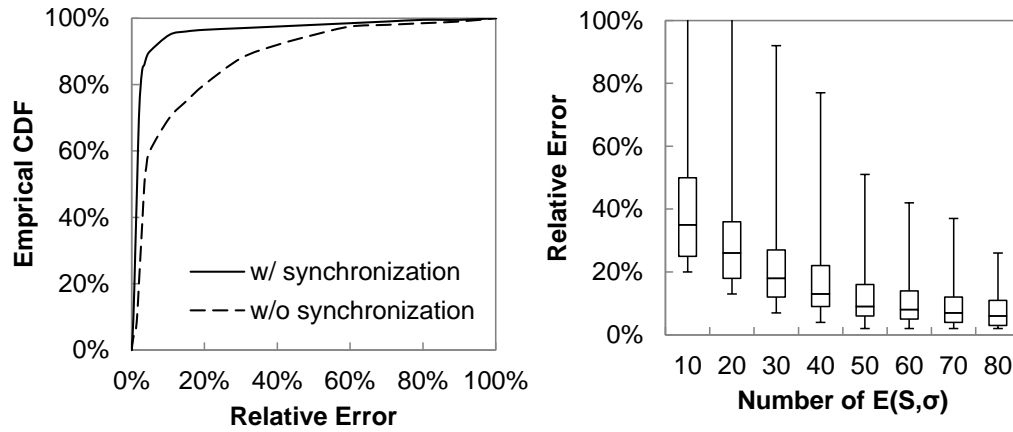


Figure 6.7 : Error in obtaining $E(\mathbb{S}, \sigma)$: Over 90% of energy measurement by the battery interface are with error less than 5% and synchronization does improve accuracy (left); estimation has a median error of 10% using one third of total $E(\mathbb{S}, \sigma)$ (right).

10 application combinations for each of the three scenarios included in our experiment. Therefore, there are 120 $E(\mathbb{S}, \sigma)$ for each scenario. In our experiment, we run each scenario for a sufficient long period and finally obtain 97, 102, and 86 $E(\mathbb{S}, \sigma)$, respectively. The footprint of the missing $E(\mathbb{S}, \sigma)$ and obtained $E(\mathbb{S}, \sigma)$ from measurement are illustrated in Figure 6.8. Each of the 10 by 12 matrix represents a scenario where each row represents each application combination \mathbb{S} and each column represents each state σ . A grey cell means the corresponding $E(\mathbb{S}, \sigma)$ is missing while a white cell means the corresponding $E(\mathbb{S}, \sigma)$ is obtained from measurement. We treat these obtained $E(\mathbb{S}, \sigma)$ (white cells) as ground truth and randomly choose part of these $E(\mathbb{S}, \sigma)$ to estimate the remaining part. Figure 6.7 (right) shows the results. As shown in the figure, our estimation algorithms can achieve a median error of 10% only using one third, or 40 out of 120, of the total number of $E(\mathbb{S}, \sigma)$.

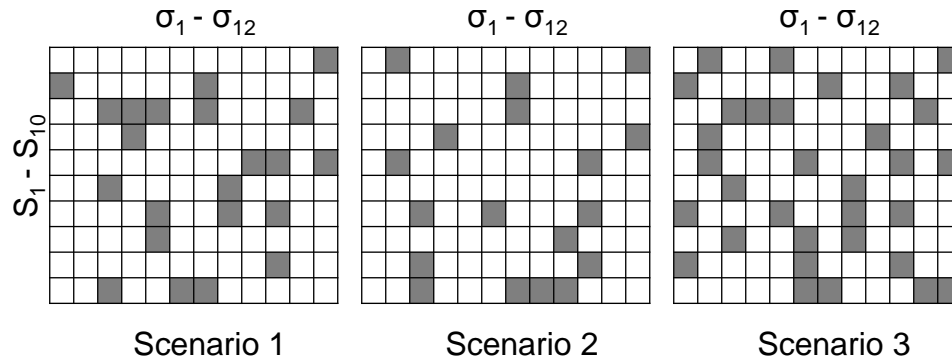


Figure 6.8 : Footprint of all the missing $E(\mathbb{S}, \sigma)$ (grey cells) and obtained $E(\mathbb{S}, \sigma)$ from measurement (white cells) in all three scenarios.

6.4.2 Evaluating Existing Policies

Using the prototype, we evaluate the five types of energy accounting policies described in Section 2.1.2 by comparing their results against those from Shapley value based energy accounting. The implementations of Policy I to IV are straightforward. We implement Policy V using the system energy model from the open source project PowerTutor [12], which is very similar to Android's own. We update the coefficients of the model following the procedures described in [5] based on offline power benchmarking and linear regression using the benchmark data. The accuracy of the updated model is higher than 90% for one reading per second.

Figure 6.9 presents the accounting results for the three scenarios. The Y axis is normalized by the total system energy consumption. If a policy meets the Axiom of Efficiency, the stacked energy contribution should be exactly 100%.

All the five types of energy accounting policies can deviate from the Shapley value significantly. Policy II and Policy III obviously violate the Axiom of Efficiency, corroborating our analysis summarized in Table 2.2. Policy IV is inaccurate because it only considers the

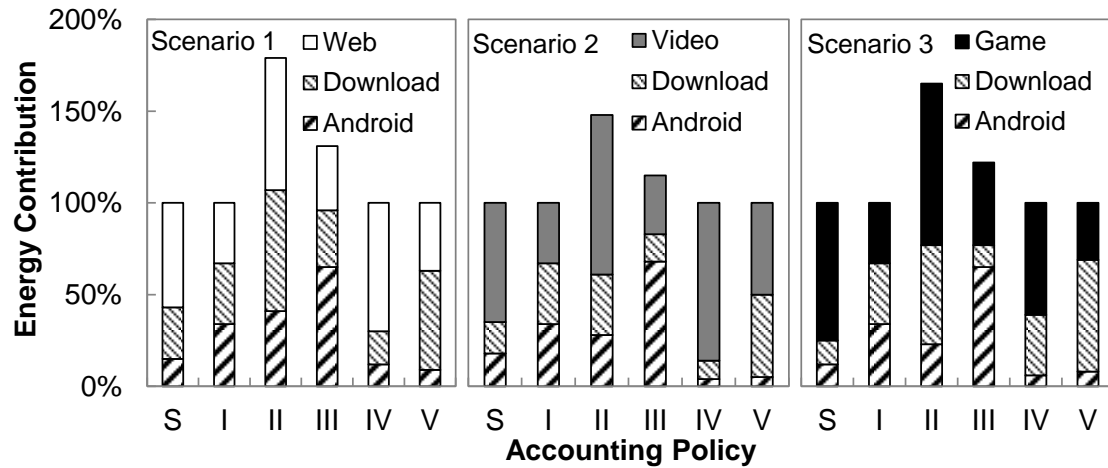


Figure 6.9 : Energy accounting results for Shapley value (S) and Policy I-V. The energy consumption (Y axis) is normalized by the total system energy consumption.

usage of CPU. Therefore, Policy IV tends to overestimate for applications that require high CPU activities such as Video in Scenario 2. Policy V is inaccurate because its energy model does not account the usage of GPU. This is obvious in Scenario 3 where Game extensively uses GPU and Policy V significantly underestimate its energy use.

6.4.3 BEM vs. OEM

We implement OEM and BEM with various energy accounting policies as described in Section 3.1 on top of Android's own kernel scheduler. The OEM/BEM scheduler works are coarse-grained and are triggered by the launch of a new application or every 100 seconds, whichever comes first. After the scheduler runs, the kernel scheduler takes care of fine-grained scheduling for each 10 ms intervals. Our implementations uses *application* instead of process as the schedule entity for the same reason described in Section 6.1

To solve the OEM optimization problem described in Section 3.1, we uses the GNU Scientific Library to implement the standard Newton Method [22] . The OEM optimiza-

tion is triggered when a new application is launched or every 100 seconds, depending on which comes first. The OEM optimization can take as long as several milliseconds in our experiment, which is comparable to the length of process scheduling period itself.

Our measurements show the OEM scheduler is able to achieve superior utility function values than BEM scheduler with various energy accounting policies. In our experiment, we combine BEM with energy accounting based on each of Policy I to V and Shapley value, as shown in Figure 6.10. To guarantee fairness, we set the utility function as the total execution time of the two applications in three scenarios using the proportional fairness utility function, the weights for the foreground application and background application is 80% and 20%, respectively. Figure 6.10 shows the corresponding optimal utility values of the optimization for the six combinations. Among all the combinations, OEM has the highest utility value and we make it as the baseline to normalize the utility values of other combinations. As shown in the figure, OEM can achieve around 20% higher utility value than the ones for BEM in all three scenarios. This confirms the results of our theoretical analysis.

Our results also corroborate the theoretical result that the energy accounting policy does not matter to BEM if it satisfies the Efficiency property. As shown in the figure, all the energy accounting policies achieve almost the same results in BEM, except Policy II and Policy III that violate Axiom of Efficiency. It is interesting that the simplest policy, Policy I, achieves similar utility value as the most sophisticated policy, Policy V. To summarize, an energy accounting policy has a very little impact on the utility value of BEM. Therefore, one should use the most efficient energy accounting policy, e.g., Policy I, to reduce implementation overhead if one insists using BEM in favor of its simplicity.

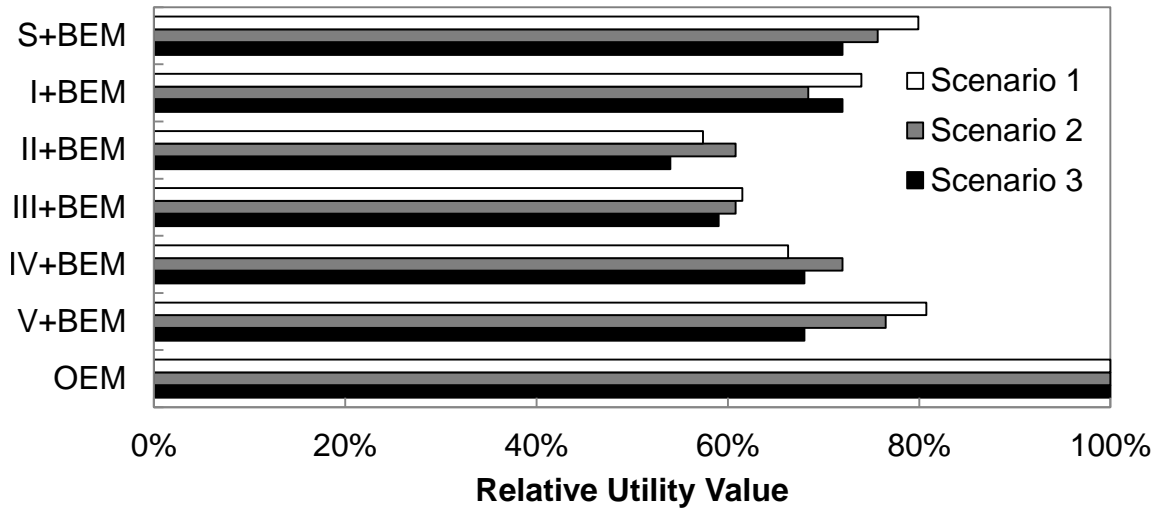


Figure 6.10 : Optimal utility values achieved by OEM and BEM with various energy accounting policies. All numbers are first normalized by that achieved by OEM for the same scenario.

6.4.4 Practical Limitations of OEM

We next investigate the overhead of OEM scheduler. The most significant portion of overhead in time is to solve the OEM optimization problem. The OEM scheduler calculates a time budget for each combination of processes, which does not scale well as number of processors increases. In our experiment, there are only three applications and the overhead is only several milliseconds. In practice, however, the application number can be as high as 12, as discussed in Section 6.1. Previous subsection provides complexity analysis of OEM, we next offer some experimental results.

Figure 6.11 shows the execution time of optimization solver with different number of variables. The time is measured in three different smartphones, Galaxy S and Galaxy S II as described in previous subsection, and an international version Galaxy S III with a quad-core processor. As shown in the figure, the execution time of the OEM optimization solver

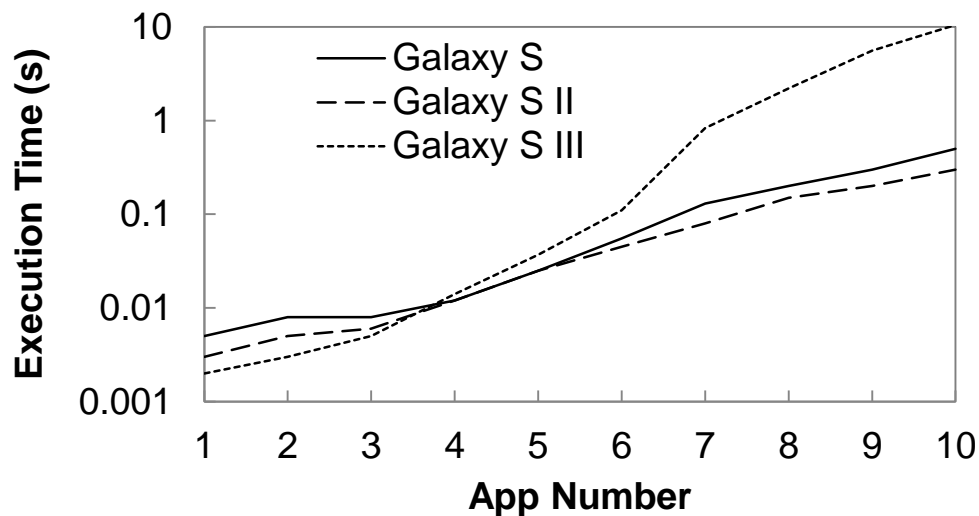


Figure 6.11 : Execution time of the optimization solver increases along with the number of variables increases. A large number of CPU cores make the scalability even more challenging.

running on Galaxy S and Galaxy S II is less than 500 ms when there are 10 applications in utility function. When the optimization solver is scheduled at every 100 seconds, the overhead in time would account for about 0.5% of the period, which is negligible. However, on Galaxy S III that has a quad-core, the optimization problem becomes much more complicated. As shown in the figure, the execution time is comparable to Galaxy S and Galaxy S II for application number less than 5, but increases significantly afterwards. With 10 applications in the utility function, the execution time is over 10 seconds, or 10% overhead. Therefore, one should limit the number of applications in utility function when using OEM. Fortunately, mobile systems rarely see more than several applications running at the same time; OEM would incur negligible overhead in such cases.

6.5 Summary

This chapter began with a novel system design of an improved battery interface that is able to conduct energy measurement at 100 Hz. The chapter then proposed two algorithms of estimating all the $E(\mathbb{S})$ using partial information. Finally, using a prototype implementation based on the improved battery interface and the estimation algorithms, the chapter showed Shapley value based energy accounting and OEM based scheduling can be realized and demonstrated their superiority experimentally.

Chapter 7

Power Modeling and Optimization for OLED Displays

Emerging organic light-emitting diode (OLED)-based displays obviate external lighting; and consume drastically different power when displaying different colors, due to their emissive nature. This creates a pressing need for OLED display power models for system energy management, optimization as well as energy-efficient GUI design, given the display content or even the graphical user interface (GUI) code. In this chapter, we study this opportunity using commercial QVGA OLED displays and user studies. We first present a comprehensive treatment of power modeling of OLED displays, providing models that estimate power consumption based on pixel, image, and code, respectively. These models feature various tradeoff between computation efficiency and accuracy so that they can be employed in different layers of a mobile system. We validate the proposed models using a commercial QVGA OLED module and a mobile device with a QVGA OLED display. Then, based on the models, we propose techniques that adapt GUIs based on existing mechanisms as well as arbitrarily under usability constraints. Our measurement and user studies show that more than 75% display power reduction can be achieved with user acceptance.

7.1 Overview

Energy consumption is an important design concern for mobile embedded systems that are battery powered and thermally constrained. Displays have been known as one of the major power consumers in mobile systems [44, 45, 46, 47]. Conventional liquid crystal

display (LCD) systems provide very little flexibility for power saving because the LCD panel consumes almost constant power regardless of the display content while the external lighting dominates the system power consumption. In contrast, the power consumption by emerging organic light-emitting diode (OLED)-based displays [48] is highly dependent on the display content because their pixels are emissive. For example, our measurement shows that a commercial QVGA OLED display consumes 1.0 and 0.2 Watts showing black text on a white background and white text on a black background, respectively. Such dependence on display content leads to new challenges to the modeling and optimization of display power consumption. First, it makes it much more difficult to account display energy consumption in the operating system for optimized decisions. Second, GUI designers will have a huge influence on the energy cost of applications, which is not their conventional concern. As a result, there is a great need of OLED display power models for use at different layers of a computing system and different stages of system design, and tools to automatically transform GUIs for power reduction.

In this chapter, we provide a comprehensive treatment of OLED display power modeling and automatically color transformation. In particular, we make four contributions by addressing the following research questions.

First, given the complete bitmap of the display content, how to estimate its power consumption on an OLED display with the best accuracy? An accurate pixel-level model is the foundation for modeling OLED display power. We base our pixel-level model on thorough measurements of a commercial QVGA OLED module. In contrast to the linear model assumed by previous work [49], we show that the power consumption is nonlinear to the intensity levels, or digital values, of the color components. Our nonlinear power model achieves 99% average accuracy against measurement of the commercial OLED display module. This is presented in Section 7.4.

Second, given the complete bitmap of the display content, how to estimate the power consumption with as few pixels as possible? This image-level model is important because accessing information of a large number of pixels can be costly due to memory and processing activities. We formulate the tradeoff as a sampling problem and provide a statistical optimal solution that outperforms both random and periodical sampling methods. Our solution achieves 90% accuracy with 1600 times reduction in sampling numbers. This is presented in Section 7.5.

Third, given the code specification of a GUI, how to estimate its power consumption on an OLED display? This code-level model is important to GUI designers as well as application and system based energy management. We use the code specification to count pixels of various colors and calculate the power consumption of the OLED display. Our model guarantees over 95% accuracy for 10 benchmark GUIs. This is presented in Section 7.6.

Fourth, given certain perceptual constraints, how to transform the colors of a given GUI to minimize the power. We first demonstrate that power-saving color transformation can be structurally applied to GUI themes and background/foreground colors. We then present a method for unstructured transformation, which applies to GUIs given in either image or code specification. We evaluate the proposed transformations with both user studies and measurement based on a commercial-off-the-shelf QVGA OLED module. We show that the automatic transformation achieves over 75% power reduction with user acceptance. This is addressed in Section 7.7 and Section 7.8.

The modeling methods and color transformation algorithms presented here provide powerful mechanisms for operating systems and applications to construct energy conserving policies for OLED displays. They will also enable GUI designers of mobile systems to build adaptable and energy-efficient GUIs; and empower end users to make informed tradeoffs between battery lifetime and usability. While many have investigated the power

optimization of conventional LCDs, these works do not leverage the extraordinary flexibility provided by OLED-based displays, i.e., one is able to change the intensity levels of every single pixel on an OLED display; for a LCD, one can only change the backlight of the whole screen at the same time. A recent few works did leverage such flexibility [49, 50, 51] but their exploitation was limited to display darkening [49, 51] and color inversion [50].

It is important to note that the proposed color transformation only applies to GUIs where usability, instead of fidelity, matters to the end users. Therefore, it does not apply to display of video or images. From this perspective, the proposed power-saving color transformation is complementary to existing LCD-based solutions that transform screen content under fidelity constraints [52, 53, 54].

The rest of the chapter is organized as follows. We provide background and address related work in Section 7.2. We describe the experimental setup used in this study in Section 7.3. From Sections 7.4 to Section 7.6, we present the power models and their experimental validations. We discuss structured color transformation based on existing color mechanisms in modern mobile platforms in Section 7.7. We propose a methodology for unstructured color transformation in Section 7.8. We address the limitation of this work and conclude in Section 7.9.

7.2 Background and Related Work

7.2.1 Display System

Figure 7.1 illustrates the relationships between the display and the rest of the system. The main processor, or application processor (AP), runs the operating system (OS) and application software with GUIs. Note the windowing system can be either part of the OS, e.g. Windows, or a standalone process, e.g. X Window in Linux. The graphics processing unit

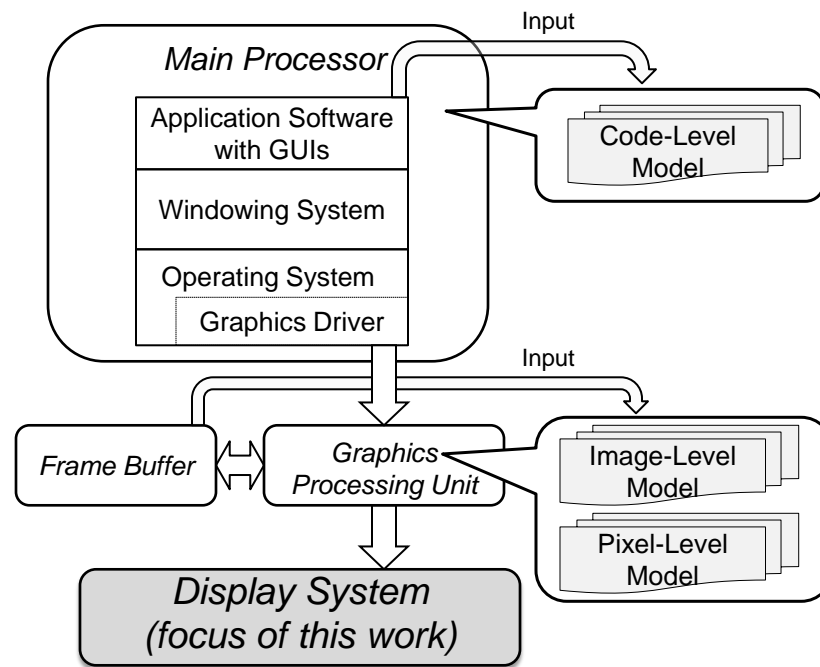


Figure 7.1 : Display system in a typical mobile system. The tree power models proposed require input of different abstractions and provide different accuracy-efficiency tradeoffs.

(GPU), often including a graphics accelerator and a LCD controller in a system-on-a-chip (SoC) for mobile devices, generates the bitmap of the display content and stores it in a memory called frame buffer; the bitmap is sent to the display for displaying. Each unit of this bitmap is described using sRGB, or standard RGB, color space, in which a color is specified by (R, G, B) , the intensity level of red, green and blue component. Because sRGB uses gamma correction to cope with the nonlinearity introduced by cathode ray tube (CRT) displays, the intensity level of each component and its corresponding luminance follow a nonlinear relation [55].

7.2.2 OLED Display

Organic light-emitting diode or OLED [48, 56] is an emerging display technology that provides much wider view angle and higher image quality than conventional LCDs. The key difference in power characteristics between an OLED display and a LCD is that an OLED display does not require external lighting because its pixels are emissive. Each pixel of an OLED display consists of three types of devices, corresponding to red, green and blue components, respectively. Moreover, the red, green, and blue components of a pixel have different luminance efficacies. As a result, the color of a pixel directly impacts its power consumption and GUI has a significant impact on the display power. In contrast, color only has negligible power impact on LCDs and illumination of external lighting dominates. OLED displays and LCDs have a very similar organization, including a panel of addressable pixels, LCD or OLED, control circuitry that generates the control and data signals for the panel based on display content, and interface to the graphics processing unit. In this chapter, we address the power consumption of the display and focus on the variance in power consumption introduced by the OLED panel when showing different content. Our power models take input from different places of the system and can be implemented either as a software tool, an operating system module, or extra circuitry.

We focus on the power consumption by a constant screen because of the following two reasons. First, a display spends most time displaying a constant screen, even for high-definition video that requires 30 updates per second. Second, our measurement showed that the power consumption by an OLED display during updating is close to the average of those by the constant screens before and after the updating. Therefore, the energy contribution by OLED display updating is very small and can be readily estimated from the power models of a constant screen. However, we note that screen updating may incur considerable energy overhead in graphics processing unit, frame buffer, and data buses, which are out of the

scope of this work.

7.2.3 Color Space

A color sensation by human can be described with three parameters because the human retina has three types of cone cells that are most sensitive to light of short (S), middle (M), and long (L) wavelengths, respectively. In optics, a color is represented by the tristimulus values, which are defined as the power per unit area per unit wavelength (W/m^3) of three primary colors, i.e., red, green, and blue, that when combined additively produce a match for the color being considered. A color space is a method for describing color with three parameters. Most used color spaces include linear RGB, standard RGB (sRGB) and CIELAB.

The linear RGB and sRGB spaces are designed to represent physical measures of light. In the linear RGB color space, a color is specified by (R, G, B) the intensity levels of the primary colors: red, green and blue, which will create the corresponding color sensation when combined. In the sRGB color space, a color is also specified by (R, G, B) but the intensity levels are transformed by a power-law compression, or *gamma correction* with $\gamma = 2.2$, to compensate the nonlinearity introduced by conventional CRT displays. Although CRT displays are no longer common nowadays, the sRGB color space is still widely used in electronic devices and computer displays. By default, almost all mainstream operating systems and applications specify a color using the sRGB color space. When a mobile display renders a color specified in the sRGB color space, it first decodes the color with its own γ value. Since most mobile displays have γ calibrated to be close to 2.2, the gamma decoding result will be the colors linear RGB values.

The CIELAB color space is designed to mimic the human vision. In the CIELAB color space, a color is specified by (L^*, a^*, b^*) , where L^* represents the lightness, human

subjective brightness perception of a color, while a^* and b^* determine the chromaticity, the quality of a color. The lightness of a color can be calculated from its relative luminance to a standard white point [57]. The CIELAB color space is so designed that uniform changes of (L^*, a^*, b^*) values aim to correspond to uniform changes in human perception of color. As a result, the Euclidean distance in the CIELAB color space is usually used to measure color difference perceived by human [57].

7.2.4 Related Work

HP Labs pioneered energy reduction for OLED-based mobile displays [49, 51, 58]. The authors proposed to selectively transform part of the screen by dimming or changing into grayscale or greenscale while maintaining good usability. Yet no real commercial OLED displays were reported in the work. The power model employed was pixel-level, thus expensive to use, and incorrectly assumed a linear relationship between intensity levels of color components and power consumption. As we will show, the relationship is indeed nonlinear. The IBM Linux Wrist Watch is one of the earliest users of OLED-based displays. The researchers sought to use low-power colors for more pixels and designed GUI objects, such as fonts and icons to minimize the need for high-power colors [59]. The work, however, only studied the GUIs with two colors, i.e., background and foreground, without addressing colorful designs.

Chuang et al. proposes two energy-aware color transformation algorithms based on a screen space variant energy model [60]. Although their algorithms can be applied to OLED displays, our work on color transformation for power optimization is different in three ways. First, they target graphic data representation, while our work addresses user interfaces on mobile systems. As a result, Chuangs work does not include any user evaluation as ours does. Moreover, they assumed an energy model that is not validated by

any real display devices, while we build and validate models based on commercial OLED devices. Finally, their algorithm constrain the transformation to colors with the same luminance, while our algorithms do not. Therefore, our color transformation algorithms provide more flexibility.

There is a large body of work on energy optimization of conventional LCD systems. It reduces external lighting and compensates the change by transforming the displayed content [52, 53, 61, 62, 63]. In particular, the characteristics of human visual perception were leveraged in [54]. Their focus on preserving the content fidelity limits their effectiveness, although it makes them applicable to all display content. In contrast, we focus on GUIs for which usability, instead of fidelity, is the concern. For example, changing the background color of a GUI will drastically reduce the fidelity but may not necessarily change the usability. Related to GUI power optimization, GUI optimization for conventional LCDs was addressed in [64]. However, because colors make very little power difference in conventional LCDs, the most effective techniques are indeed concerned with improving user speed or productivity.

7.3 Experimental Setup

7.3.1 Subjects

We recruited twenty subjects from Rice University through our own social networks for user evaluation reported in this work. Their ages range between 22 and 27. All but four were males. None of the subjects had used an OLED-based display before. Because of the number of subjects and the way that they were recruited, we do not claim that they represent any particular demography.

7.3.2 Benchmarks

To evaluate the proposed power models, we collect 300 GUI images from three Windows Mobile-based cell phones, HTC Touch, HTC Mogul, and HTC Wizard, all with a resolution of QVGA (240x320). On each phone, we exhaust all the varying GUI screens, representative of everyday use of a smart phone (e-mail, web browser, games, etc). We also capture screens with different color themes available.

7.3.3 OLED Display

For our experimental validation, we use a standalone 2.8" QVGA OLED display module with an integrated driver circuit, μ OLED-32028-PMD3T from 4D Systems [65], and a Nokia N85 smartphone with 2.4" QVGA OLED display. For the standalone μ OLED display module, we connect it to a PC using a micro USB interface, which also supplies power to it. Through the USB, we can send commands to the OLED module to display images. The display module employs a standard 16-bit (5,6,5) RGB setting. That is, there are five, six, and five bits to represent the intensity of red, green, and blue component, respectively. In this work, we call their numbers the intensity level or value of the three color components.

7.3.4 Power Measurement

We obtain the power consumption of the μ OLED module by measuring the current it draws from the USB interface and its input voltage. Figure 7.2(left) shows the measurement setup with a data acquisition (DAQ) board from Measurement Computing and the μ OLED module. To eliminate the impact of contact resistance in the breadboard, we calibrated the experiment setup before measurement. We first fed a current of fixed value from a current source to the breadboard to obtain the "real" resistance including both of the sensing resis-

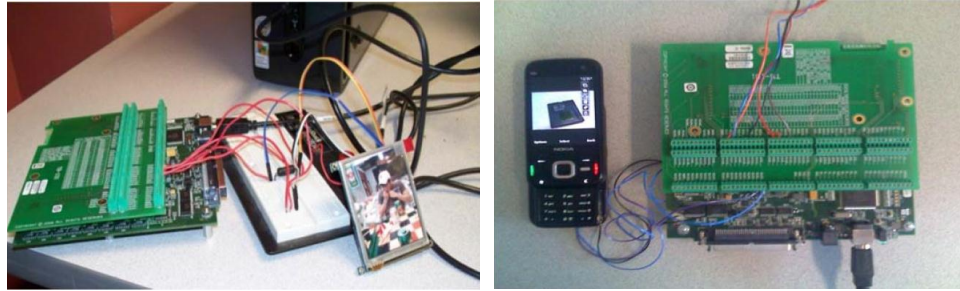


Figure 7.2 : Measurement setup of the QVGA μ OLED module (left) and the Nokia N85 (right) used in our experimental validation.

tor and contact resistor. Then we use the “real” resistance to calculate the current drawn by the OLED display in the following measurement. To overcome the variance among different measurements of the same image, we take an average of 1000 measurements for each image. We use the μ OLED module to generate a pixel power model, measure the power consumption of color transformed GUIs and display them for user evaluation.

For the Nokia N85, we first hack the battery and put a current sensing resistor at the power supply terminal; then we measure its total power consumption by measuring the current it draws from the battery and the battery voltage. Figure 7.2(right) shows the measurement setup with a DAQ board from Measurement Computing and the N85. During the measurement, we try to keep the status of components other than the display unchanged such that the difference of two measured power only comes from the change of the display.

7.4 Pixel-Level Power Model

We first present a pixel-level power model that estimates the power consumption of OLED modules based on the RGB specification of each pixel. It is intended to be the most accurate and constitutes the baseline for models based on more abstract descriptions of the display

content.

7.4.1 Power Model of OLED Module

We model the power consumed by a single pixel i , specified in (R_i, G_i, B_i) as

$$P_{pixel}(R_i, G_i, B_i) = f(R_i) + h(G_i) + k(B_i),$$

where $f(R_i)$, $h(G_i)$, and $k(B_i)$ are power consumption of red, green, and blue devices of the pixel, respectively. And the power consumption of an OLED display with N pixels is

$$P = C + \sum_{i=1}^N (f(R_i) + h(G_i) + k(B_i)).$$

Note that the model includes a constant, C , to account for static power contribution made by non-pixel part of the display, which is independent with the pixel values. This pixel model has a generic form that applies to all the colorful OLED display modules.

7.4.2 Power Model of RGB Component

We obtain C by measuring the power consumption of a completely black screen. To obtain $f(R)$, we fill the screen with colors in which the green and blue components are kept zero and the red component, R , varies from 0 to 31, enumerating every possible intensity level. For each measurement, we subtract out C to get just the power contribution by the red pixel component, or $f(R)$. We obtain $h(G)$ and $k(B)$ similarly. Figure 7.3(a) presents the measured data for all three components. Apparently, the power contribution by pixel components is a nonlinear function, instead of a linear function as assumed in [49], of the intensity level. The nonlinearity is due to the gamma correction in sRGB standard. After transforming the intensity level into linear RGB format, which is the indication of luminance, we obtain a linear relation between pixel power consumption and intensity, as

shown in Figure 7.3(b). We also perform the same procedures to obtain the power model for the N85 mobile phone and the data are presented in Figure 7.3(c)(d). As we see in the figures, the models for μ OLED and N85 are slightly different, i.e., the red devices in the display of N85 have relatively higher power consumption, although they are similar in general. This is due to the difference in OLED devices used in the displays. The measured data can be directly used to estimate the power consumption of displaying an image on the OLED display through simple table lookup. One can also apply curve fitting to obtain close-form functions for $f(R)$, $h(G)$, and $k(B)$. For example, when we use a cubic function to fit the data presented in Figure 7.3, the fitting statistics R^2 of the models of μ OLED and N85 are 0.998 and 0.952, respectively.

7.4.3 Estimation vs. Measurement

Figure 7.4 shows the histograms of the error of the estimation against the measurement for the 300 benchmark images. For μ OLED, 63% of the samples have no more than 1% error, 93% have no more than 3% errors, and the average absolute error is only 1%. For N85, 57% of the samples have no more than 1% error, 84% have no more than 3% errors, and the average absolute error is only 2%. The error is higher for N85 than the one for μ OLED is due to the experimental setup. Since we are not able to obtain the power consumption of the pixels directly, instead, we measure the total power of the whole module, μ OLED or N85, and use a constant C to account the power of other components. In μ OLED, other components are mainly driver circuits that are operating in a smooth manner. In N85, however, various system activities such as process switches will lead to power spikes such that C is suffering from more variation.

It is important to note that although we derived the pixel-level power models from two specific OLED displays, the methodology can be largely extended to other OLED displays

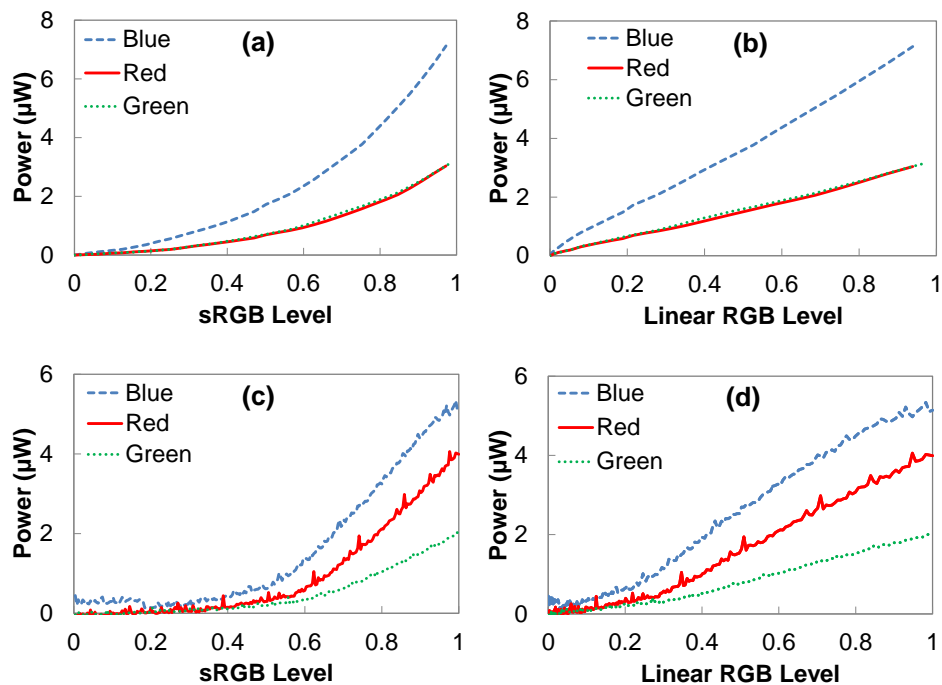


Figure 7.3 : Intensity Level vs. Power Consumption for the R, G, and B components of an OLED pixel: (a) μ OLED, sRGB; (b) μ OLED, linear RGB; (c) N85, sRGB; (d) N85, linear RGB.

with a similar RGB pattern.

7.5 Image-Level Power Model

We next present models that estimate power consumption given the image to display. They are important for a system to assess the display power cost when the pixel information is known, e.g. from the frame buffer. A straightforward image-level model can be a simple application of the pixel-level model to all pixels. Such a method, unfortunately, is exceedingly expensive. Modern displays can have hundreds of thousands or millions of pixels.

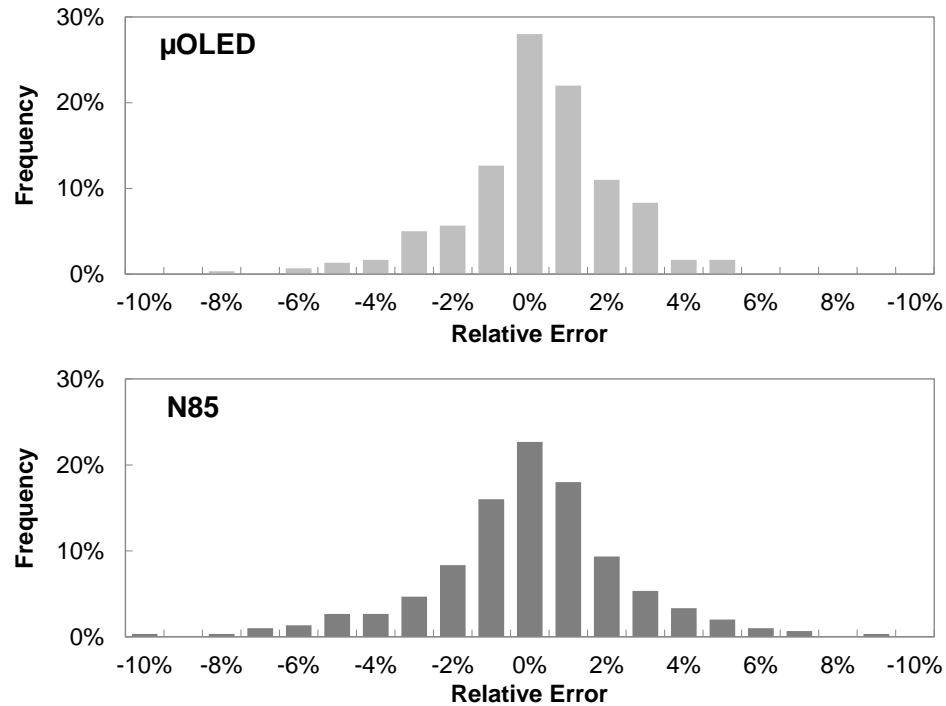


Figure 7.4 : Histogram of percent errors observed for the 300 benchmark images using our pixel-level power model.

The cost of a large number of pixel power calculations and the overhead of accessing the frame buffer can be prohibitively high for the system. To tackle this challenge, we propose a color histogram based method in which we count the number of pixels with the same color and aggregate power consumption of all the colors. Moreover, we propose a more aggressive approach, i.e., to estimate the power based on a small subset of pixels, or *sampling*.

7.5.1 Color Histogram-based Method

A QVGA image contains 76,800 pixels and many of them share the same color. Therefore, it is unnecessary to recalculate the power contribution of the pixels that are of the same

color. To leverage the repetitiveness in colors, we propose a color histogram-based method to estimate the power consumption of an OLED display showing a specific image. In this method, we first traverse all the pixels of the image and obtain a color histogram of the image, i.e., the pixel numbers of each color contained in the image. Depending on the color depth to use, the histogram may have different resolution. For example, when we consider each of the R , G , and B has four levels of depth, the histogram will have 64 colors in total. Based on this color histogram, we can obtain the power contribution of the image by calculating the weighted sum of the pixel number of all the colors. The weight of each color is its power consumption that can be pre-calculated using the pixel level power model described in Section 7.4.

Figure 7.5 shows the tradeoff between computation time and error of the color histogram-based method. The numbers presented in the figure are measured in the N85 mobile phone. It clearly indicates that we can achieve a lower error by utilizing a higher resolution in creating the color histogram and, correspondingly, the computation time is longer. When each of the R , G , and B is represented in eight levels of depth, i.e., 512 colors in total, power estimation of an image of 90% accuracy can be completed in 100 ms.

7.5.2 Sampling-based Method

We next discuss a more aggressive approach, i.e., to estimate the power based on a small subset of pixels, or *sampling*.

Problem Formulation

The power consumption by an image on an OLED display can be described by a vector of N elements, i.e., $\mathbf{y} = (y_1, y_2, \dots, y_N)^T$, in which y_i denote the power consumption of the i -th pixel, $i = 1, 2, \dots, N$. The total display power can be calculated as

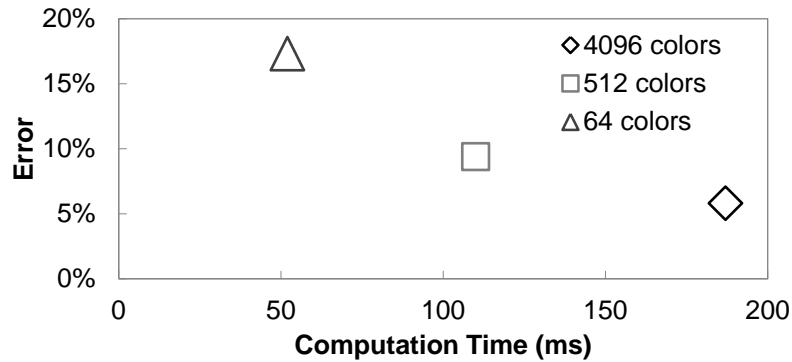


Figure 7.5 : Tradeoff between computation time and error of the color histogram-based method.

$$P_{image} = \mathbf{1}^T \mathbf{y} = \sum_{i=1}^N y_i.$$

Sampling is to select K pixels out of N and use the average power of the K samples to approximate the average power of all N pixels. For each pixel, we use a random variable to indicate whether the pixel is sampled or not, i.e., $X_i = 1$ when the i -th pixel is sampled; $X_i = 0$ otherwise. Thus, the sampling of an image can be represented by a vector $\mathbf{X} = (X_1, X_2, \dots, X_N)^T$. And denote the joint probabilistic distribution of them as $f_{\mathbf{X}}(x) = f_{(X_1, X_2, \dots, X_N)}(x_1, x_2, \dots, x_N)$. Given an image, \mathbf{y} , the root-mean-square estimation error can be calculated as

$$\begin{aligned} \varepsilon &= E_{\mathbf{X}} \left[\left(\frac{\mathbf{X}^T \mathbf{y}}{K} - \frac{\mathbf{1}^T \mathbf{y}}{N} \right)^2 \right] = E_{X_1, X_2, \dots, X_N} \left[\left(\frac{\sum_{i=1}^N X_i y_i}{K} - \frac{\sum_{i=1}^N y_i}{N} \right)^2 \right] \\ &= \int \left(\frac{\sum_{i=1}^N X_i y_i}{K} - \frac{\sum_{i=1}^N y_i}{N} \right)^2 f_{X_1, X_2, \dots, X_N}(x_1, x_2, \dots, x_N) dx. \end{aligned}$$

Therefore, an optimal sampling given the image can be obtained by finding $f_{\mathbf{X}}(x)$ that leads to the minimal ε , or

$$f_{\mathbf{X}}^*(x) = \arg \min_{\forall f_{\mathbf{X}}(x)} \varepsilon$$

Now, we consider all possible images by treating the power consumption of an image as a random vector, $\mathbf{Y} = (Y_1, Y_2, \dots, Y_N)^T$, and denote the joint probability distribution as $g_{\mathbf{Y}}(y) = g(Y_1, Y_2, \dots, Y_N)(y_1, y_2, \dots, y_N)$. Then the optimal sampling for all images can be found as

$$f_{\mathbf{X}}^*(x) = \arg \min_{\forall f_{\mathbf{X}}(x)} E_{\mathbf{Y}}[\varepsilon].$$

That is, finding a probability simplex $p = (p_1, p_2, \dots, p_{\binom{N}{K}})^T$ such that $E_{\mathbf{Y}}[\varepsilon]$ is minimal, where $p_j = \text{Prob}(X = x^{(j)})$ and $x^{(j)}$ is the j -th ($j = 1, 2, \dots, \binom{N}{K}$) possible combination of \mathbf{X} . This is a linear programming problem, i.e.,

$$\begin{aligned} & \text{minimize} \int \sum_{j=1}^{\binom{N}{K}} p_j \left(\frac{\sum_{i=1}^N X_i y_i}{K} - \frac{\sum_{i=1}^N y_i}{N} \right)^2 g_{Y_1, Y_2, \dots, Y_N}(y_1, y_2, \dots, y_N) dy \\ & \text{subject to} \sum_{j=1}^{\binom{N}{K}} p_j = 1; p_j \geq 0. \end{aligned}$$

It is, however, impractical to obtain the joint distribution $g_{\mathbf{Y}}(y)$ because the dimension is prohibitively high. Therefore, we transform the objective function to eliminate the joint distribution as below.

$$\begin{aligned} & \int \sum_{j=1}^{\binom{N}{K}} p_j \left(\frac{\sum_{i=1}^N X_i y_i}{K} - \frac{\sum_{i=1}^N y_i}{N} \right)^2 g_{Y_1, Y_2, \dots, Y_N}(y_1, y_2, \dots, y_N) dy \\ & = E_{\mathbf{X}} \left[\left(\frac{\mathbf{X}}{K} - \frac{\mathbf{1}}{N} \right)^T E_{\mathbf{Y}}[\mathbf{Y}\mathbf{Y}^T] \left(\frac{\mathbf{X}}{K} - \frac{\mathbf{1}}{N} \right) \right] \\ & = \sum_{j=1}^{\binom{N}{K}} p_j \left(\frac{\mathbf{X}}{K} - \frac{\mathbf{1}}{N} \right)^T E_{\mathbf{Y}}[\mathbf{Y}\mathbf{Y}^T] \left(\frac{\mathbf{X}}{K} - \frac{\mathbf{1}}{N} \right). \end{aligned}$$

Therefore, the key to an accurate estimation is to obtain a good knowledge of the expectation of outer products of all possible images, which can be statistically learned from a set of training images by using the mean of the outer products of the training images to approximate the expectation $E_{\mathbf{Y}}[\mathbf{Y}\mathbf{Y}^T]$.

Practical Learning-based Sampling

There are two critical barriers for the practical implementation of the statistically optimal sampling. First, the linear programming problem described above is very difficult, if possible, to solve because of the extremely high dimension, i.e. N-choose-K. To address this issue, we divide the image into much smaller windows, and then solve the linear programming problem for each window.

By preparing a training set for the window at every position within an image, we are able to obtain an optimal sampling solution for every window. Second, the solution requires generating a large number of random numbers, which is expensive for mobile embedded platforms. Therefore, instead of random sampling, we decide to use deterministic sampling method in which we seek to solve a combinatorial problem to find the samples for each window such that the objective function is minimal. That is

$$\begin{aligned} & \text{minimize} \quad \left(\frac{\mathbf{X}}{K} - \frac{\mathbf{1}}{N}\right)^T E_{\mathbf{Y}}[\mathbf{Y}\mathbf{Y}^T] \left(\frac{\mathbf{X}}{K} - \frac{\mathbf{1}}{N}\right) \\ & \text{subject to} \quad \sum_{i=1}^N x_i = K; x_i = 1 \text{ or } x_i = 0. \end{aligned}$$

Another practical issue is the number of samples to take from each window, or K in the formulation above. Given a fixed sampling rate, or the total number of samples, there are two strategies. One can increase the window size, therefore reduce the number of windows, but allow more samples per window. Or one can reduce the window size, therefore

increase the number of windows, but allow fewer samples per window. The second strategy obviously is more efficient because the computational load increases much faster with the window size than with the number of samples per window. We also experimentally compared the accuracy of these two strategies. We found that there is no difference in accuracy between them. Therefore, we take the second strategy in our image-level mode, i.e. a single sample is taken from each window.

Experimental Results

To evaluate our proposed learning-based sampling method (Learning-based), we compare its performance against three other sampling methods, described below.

- **Periodical sampling:** an image is divided into non-overlapping windows of the same size, exactly the same as Learning-based sampling but the bottom right pixel of each window is selected.
- **Local Random sampling:** one pixel is randomly selected from each window.
- **Global Random sampling:** pixels are randomly selected from the whole image. The number of pixels is kept the same as the number of windows in the first three methods.

Collectively these benchmark sampling methods will highlight the effectiveness of the design decisions made in Section 7.5.2. It is important to note that for learning-based sampling, we employ a bootstrapping method to improve the reliability of accuracy evaluation because training is involved. That is, we divide the 300 benchmark images evenly into 10 groups, i.e. 30 in each. Then we use 9 groups as training set and test the 10th group. We repeat the process 10 times using different groups as the test group and the report the average accuracy.

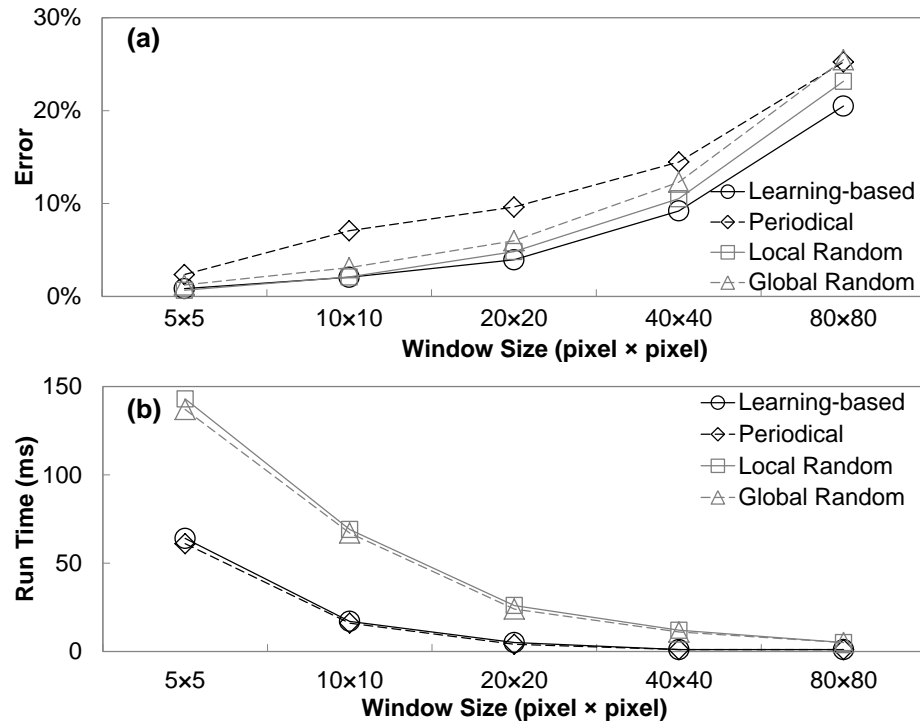


Figure 7.6 : Comparison of four different sampling methods in image-based power models.

We investigate the tradeoff between sampling rate and accuracy by varying the window size from 55, 1010, 2020, 4040, to 8080. This leads to a sampling rate between $1/25$ and $1/6400$.

Figure 7.6(a) presents the tradeoffs between estimation error and window size over the 300 benchmark GUIs described in Section 7.3. It clearly shows that as window sizes increases, or sample rates decreases, the estimation error increases, for all four sampling methods. Figure 7.6(a) clearly shows that our learning-based method achieves the best tradeoff between accuracy and sampling rate. When window size is 4040 and a 1600 times reduction in pixels needed for power estimation, learning based sampling method achieves accuracy of 90%.

Figure 7.6(b) presents the run-time of all four sampling methods on the N85 mobile phone. It clearly demonstrates the advantage in efficiency of deterministic methods, i.e. Learning-based and Periodical. With the comparable accuracy, Learning-based sampling is at least 1.5X faster than the random methods, both Global and Local. This will lead to significant efficiency improvement when the power model is employed in mobile embedded systems for energy management and optimization.

In summary, our learning-based sampling method achieves the best tradeoff between accuracy and efficiency. We note that the learning-based method achieves this through training, which can be compute-intensive. However, training can be carried off-line and therefore does not consume any resource at run-time.

7.6 Code-Level Power Model

Both pixel and image-level models require that the RGB information is available for all pixels of the display content. In this section, we present a power model that is based on the code specification of the display content. This is possible because graphical user interfaces (GUIs) are usually described in high-level programming languages, e.g. C# and Java, and are highly structured and modular. The code-level model can be even more efficient than the sampling-based image-level power model because they do not need to access the frame buffer. Therefore, it can be readily employed directly by the application or the operating system. In addition, they also provide a tool for GUI designers to evaluate their designs at an early stage.

We take an object-oriented approach because modern GUIs are composed of multiple objects, each with specified properties, e.g. size, location, and color. Moreover, GUI programming is also object-oriented. Developers rarely specify the graphics details; instead they extensively reuse a “library” of customizable common objects, e.g. buttons, menus,

and lists. They customize these objects by specifying their properties and their relationship with each other within a GUI. Our methodology for code-based power modeling is first estimating power contribution by individual GUI objects based on their properties and then estimating the total power based on the composition of these objects.

7.6.1 Power by GUI Object

We can view a GUI object as a group of pixels with designated colors. By accounting the number of pixels with each color that appear in an object, we can obtain its power consumption using the pixel-level power model. To obtain the estimated power for the object, we enhance the object class with a pixel list property that records the (R, G, B) values and pixel number of each color. Therefore, by utilizing our pixel-level power model, we can calculate the power consumption of an object with n colors as

$$P_{object} = \sum_{i=1}^n num_i \cdot P_{pixel}(color_i).$$

In most cases, a GUI object only has three colors, i.e., border color, background color and foreground (text) color. All three can be obtained from the properties of the GUI object. Knowing the geometric specification of the object, usually rectangular, we can calculate the number of pixels for both background and border colors. Then we create a pixel list of two entries for this object, which includes the (R, G, B) values of colors and their corresponding pixel numbers. By extending the pixel list, we are able to handle more complicated objects with arbitrary shapes and more colors.

Text is a special object property and needs a special treatment. Unlike the color information available from the object properties, the pixel number of text is not easy to obtain. Thus, we build a library for all the ASCII characters; the library contains the number of pixels of each character based on its font type and size. Thus, we are able to account the to-

tal number of pixels of a whole text string, which we should subtract from the background pixel number. For our experiments, we have constructed the library for Times New Roman and Tahoma of font sizes 12 and 9, which are the most common on Windows Mobile devices.

7.6.2 Power by Composition of Objects

A GUI usually consists of multiple objects. Simply aggregating the power of all of the objects is not accurate enough because they may overlap with each other. To consider the effect of overlapping, we maintain a pixel list pl_{GUI} for the whole GUI based on the pixel list of each objects. We first sort the objects from back to front in the GUI. Denote the sorted object list includes n objects O_1, O_2, \dots, O_n , with the sequence from the back to the front, and the pixel list of O_i is pl_i . As shown in Algorithm 1, we go through all the objects one by one while checking their overlapping situation. If two objects overlap with each other, we update the pixel list of the object in the back by subtracting the number of pixels being covered. Finally, we merge all the pixel lists by combining the pixels with the same color together. We describe how we deal with two special effects of GUIs below.

Dynamic Objects: Some GUI objects can have multiple states. For example, a radio button has two states, checked and non-checked; a menu has even more states when different items are activated. For these objects, we maintain a pixel list for each state and treat each state as an individual object in the procedure described above.

Themes: Many operating systems support color themes that contain pre-defined graphical details, such as colors and fonts, so that GUIs of different applications will follow a coherent style. For instance, in Windows Mobile and C#, BackColor and ForeColor can be either specified using (R, G, B) or selected from a set of pre-defined colors, such as Window and ControlText. These system colors are determined by the theme of Windows Mobile.

Algorithm 1 Power estimation for a composition of GUI objects

```

for  $i = 2$  to  $n$  do
    for  $j = 1$  to  $i-1$  do
        if  $O_i \cap O_j \neq \emptyset$  then
            UPDATE( $pl_j$ )
        end if
    end for
end for

for  $k = 1$  to  $n$  do
     $pl_{GUI} = \text{MERGE}(pl_{GUI}, pl_k)$ 
end for

```

Fortunately, the color and font information can be obtained in the GUI code in the run-time for power estimation.

7.6.3 Experiment Results

We have implemented the object-based power estimation on .NET Compact Framework and C#, for Windows Mobile based mobile embedded systems. In C#, most GUI objects belong to namespace System.Windows.Controls. All objects provide height, width, and background color, which are enough for power estimation. We use eight sample programs with 10 GUIs from the Windows Mobile 5.0 SDK R2 to evaluate the implementation. Using code-level model, we estimate the power consumption of the 10 GUIs, and compare the results with the estimation from the pixel-level model, image-level model (window size: 4040), and measurement. Figure 7.7(a) presents the results and shows that our code-level model with text achieves higher than 95% accuracy for all the benchmark GUIs. As shown

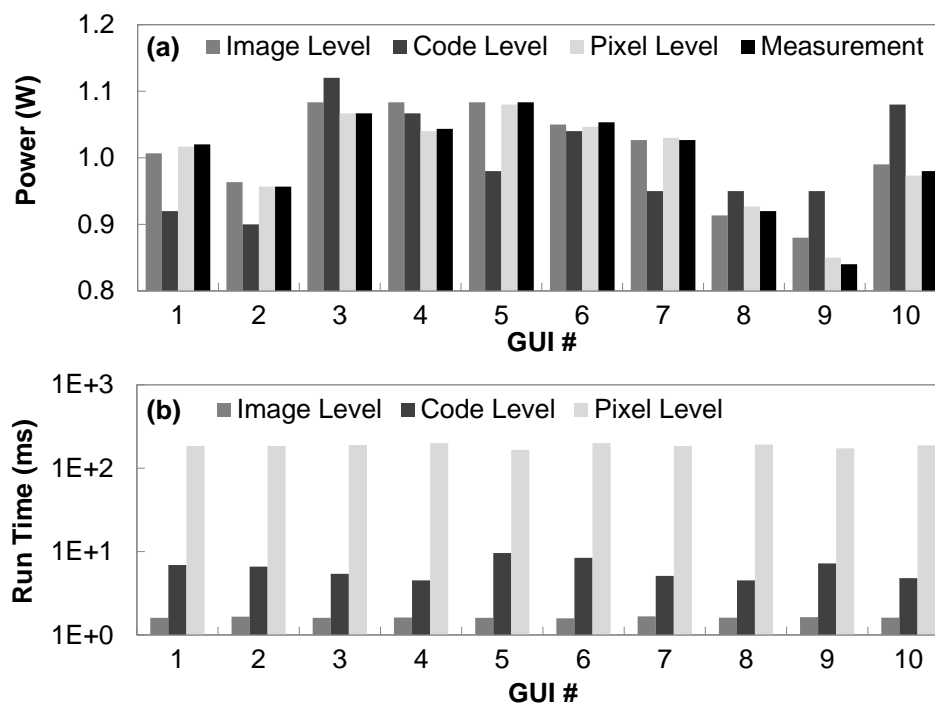


Figure 7.7 : Comparison of three power models.

in Figure 7.7(b), the average time cost of code-level model is approximately 6 ms, or 3X higher than the average time cost of image-level model whose accuracy is about 90%.

7.7 Structured Transformation

Modern GUIs are usually structured. A GUI is composed of multiple objects, each with properties, e.g. size, location, and color, specified in software. In most cases, most pixels of a GUI object have one of three colors, i.e. border color, background color and foreground (usually text) color. In addition, most platforms support color themes in which GUI objects enjoy structurally consistent color patterns. Such structured provide an opportunity for color transformation without jeopardizing usability.

7.7.1 Background/Foreground Transformation

The background color usually claims most of the pixels of a GUI, e.g., the 300 GUI images described in Section 7.3, on average, have 73% of the pixels as background. Unfortunately, white, the most power-hungry color, is commonly used as the background color. Therefore, the most straightforward power-saving color transformation to employ a very low-power color for background but colors of high contrast for border and foreground.

The research question is whether doing so will jeopardize user satisfaction of the GUIs. To answer this question, we design 15 very simple background/foreground schemes using black, grey, and white as background colors, and use black, white, red, green, and blue as foreground colors. We apply these schemes to a GUI that only includes texts, load them to the OLED module described in Section 7.3, and show them to our subjects one by one.

We ask our subjects to choose their top 3 favorite from the 15 background/foreground color schemes without knowing their power consumption. The three most popular schemes are grey/white, white/, and black/green, with 18, 16, 15 votes, respectively. With similar user satisfaction, however, the power consumption of the three schemes are dramatically different, i.e., 1.2 W, 3.0 W, 0.7 W, respectively. This demonstrates that *it is possible to achieve both low-power and high user satisfaction in background/foreground transformation.*

7.7.2 Theme-based Transformation

Modern mobile platforms usually support color themes that apply consistent colors to platform-specific features, e.g. title bar and borders. The themes therefore provide another structure to transform GUI colors for power saving.

To study the power impact of themes, we choose eight different themes from the Windows Mobile Themes website [66], each with over 10 review scores from the website. We

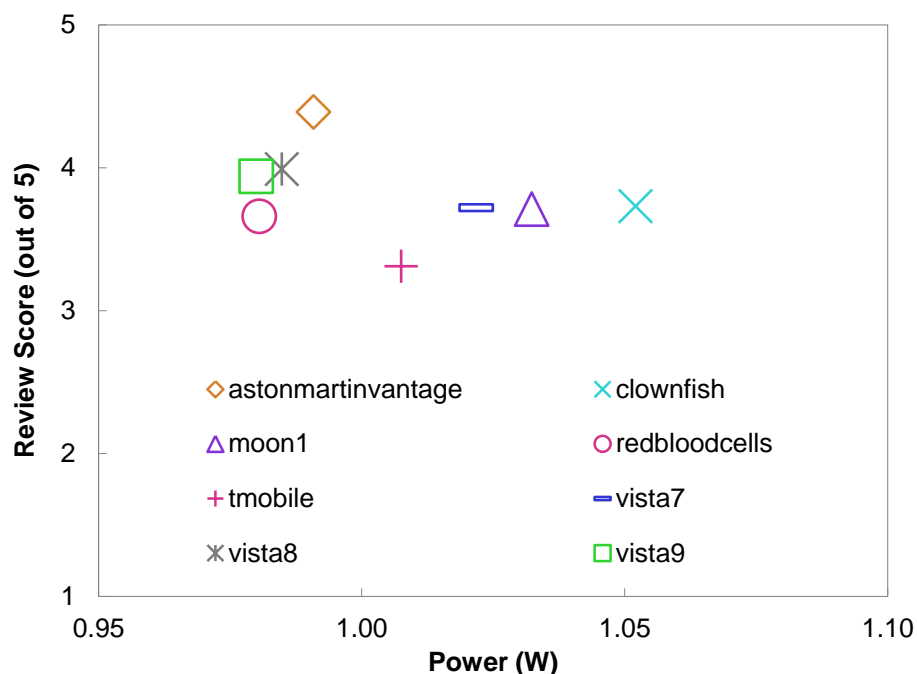


Figure 7.8 : Power consumption and user review scores of GUIs with different themes.

apply them to eight sample programs with 10 GUIs from the Windows Mobile 5.0 SDK R2. We measure the power consumption of the 10 GUIs with each theme. Figure 7.8 shows the power consumption and average user review scores of the eight themes. As shown in the figure, there is no obvious relation between power consumption and review scores. This indicates that *proper theme designs can achieve both low-power and high user acceptance*.

7.8 Unstructured Transformation

We present a much more flexible unstructured transformation method that can be applied to any GUIs. We first count the number of pixels for each color in a GUI to get a color histogram. Then, we map each of the original colors to a new one based for lower power

consumption.

7.8.1 Color Counting

Using the power models described in Section 3.3, we are able to calculate the power consumption of a single pixel of any color. To get the power consumption of the whole display, we must count how many pixels for each color in the GUI, or a pixel number histogram of all colors. For a GUI specified by image, we can obtain the histogram by simply enumerating all the pixels. We can also apply sampling method to tradeoff between accuracy and computing cost. For a GUI specified by code, we use the following algorithm to obtain the color histogram. Suppose the list *Obj* and *ColorList* record all the objects and colors in the GUI, respectively. We go through all the objects in *Obj* and check whether its colors, including background, foreground and border colors, is in the *ColorList*, and add in the colors that are not yet in the list.

7.8.2 Color Mapping

After a color histogram of a GUI is obtained, we seek to reduce the power consumption of the OLED-based display by replacing each color of the histogram with a new color.

Problem Formulation

This process can be formulated as a minimization problem. A GUI employs multiple colors, $color_1, color_2, \dots, color_n$, each of which is specified by a three component vector, either in sRGB space or CIELAB space, i.e., $color_i = (R_i, G_i, B_i) = (L_i^*, a_i^*, b_i^*), i = 1, 2, \dots, n$.

We can count how many pixels have color $color_i$ and denote the number as num_i , which we obtain from color accounting. Then the power consumption of the GUI can be calculated as

$$P_{GUI} = \sum_{i=1}^K num_i \cdot P_{pixel}(color_i),$$

where $P_{pixel}(color_i)$ is the pixel-level power model described in Section 7.4. The object of color mapping is to find n colors, $color'_1, color'_2, \dots, color'_n$, such that power consumption $\sum_{i=1}^n num_i \cdot P_{pixel}(color'_i)$ is minimized, while satisfying certain human perception constraints.

Color difference, defined as the Euclidean distance in CIELAB color space (L^*, a^*, b^*) , is a widely used metric to measure human perceived difference [57] between colors. Two colors are considered perceptually identical when the difference between the two is less than certain threshold. Note previous work attempted to transform the screen while preserving its fidelity, i.e., there should be little human-perceptible difference between the transformed screen and the original [54]. In our case, however, we do not preserve the fidelity of GUI images. Instead, we only keep the usability, i.e., human users still perceive different colors in the transformed GUI as different as their originals. Particularly, we study two heuristics to quantitatively measure usability for mathematical optimization. (i) The color difference between any two colors in the transformed GUI should not deviate from that between their original colors by a user-specified factor. This heuristics guarantees that the contrast between colors in the original GUI will be preserved to a user-specified degree. (ii) If in the original GUI, the difference between color x and color y is larger than that between color u and color v , the same relation should be preserved for the colors x, y, u , and v are transformed into. This heuristics ensures that a salient feature in the original GUI will remain so in the transformed GUI.

Thus, we formulate the problem as

$$\begin{aligned}
& \text{minimize } \sum_{i=1}^n \text{num}_i \cdot P_{\text{pixel}}(\text{color}_i) \\
& \text{subject to } \|\text{color}'_i - \text{color}'_j\|_2 = \lambda \cdot \|\text{color}_i - \text{color}_j\|_2, \forall i, j \in \{1, 2, \dots, n\}.
\end{aligned}$$

As shown in the formulation, we seek to minimize the power consumption of a GUI while guaranteeing that the color difference between two colors in transformed GUI is approximately the same as that between their original colors. Since it is likely that no solution exists that satisfies all the constraints, we employ a scalar $\lambda \in [0, 1]$ to indicate how strictly we want to enforce the perception constraints. Because potentially the operating system can transform the GUI at run time for energy management, it is important that a solution to the problem can be identified rapidly. Therefore, we will focus on the complexity in describing solutions below.

Rank-based Mapping

The most straightforward solution is brute-force search, in which we divide the each of the R , G , and B component into discrete intervals and traverse all the possible combinations to find the one with minimal power consumption. Obviously, brute-force search guarantees to find the optimal solution, but its computation time is too long when n is large. Denote the intensity level of R , G , and B component is m_R , m_G , and m_B , respectively. Brute-force search requires $(m_R m_G m_B)^n$ iterations. In each of them, there are $n(n-1)/2$ constraints to check. Therefore, its complexity is $O(n^2(m_R m_G m_B)^n)$.

We use a greedy algorithm to speed up the search process which determines the colors one by one and seeks to achieve minimal power consumption at each step. It first chooses the color with zero power, i.e., black, to transform the color with the highest pixel number. And then it determines the remaining colors based on the rank of pixel number obtained in

color counting. For each color, it seeks to achieve the minimal power consumption of the color itself while satisfying the constraints. Because the greedy solution determines the n colors one by one without stepping back, it only requires nm_{RMGB} iterations. Therefore, its complexity is $O(n^3 m_{RMGB})$.

The greedy solution has much better asymptotical computation time than the brute-force search, but it cannot guarantee the optimal solution. Yet, the greedy solution works fine in all our experiments, especially in the cases when a white background color accounts for most of the pixels in a GUI.

Monochrome Mapping

By default, the color mapping can select from all colors. We can also add constraints so that only a subset of the color space can be used. For example, by keeping a_i^{*} and b_i^{*} constant, we are able to perform a monochrome mapping, in which all the new colors share the same chromatic property but are with different luminance. Then the problem becomes

$$\begin{aligned} & \text{minimize } \sum_{i=1}^n \text{num}_i \cdot P_{\text{pixel}}(\text{color}_i) \\ & \text{subject to } |L_i^{*} - L_j^{*}| = \lambda \cdot \|\text{color}_i - \text{color}_j\|_2, \forall i, j \in \{1, 2, \dots, n\}. \end{aligned}$$

Apparently, we can still use rank-based mapping to solve the problem. In this case, however, the constraints become linear and the objective function regarding L^* is convex, which guarantees that we can find solutions in polynomial time, e.g. the Newton method or Interior-Point method [22]. Moreover, as we see later, monochrome mapping provides higher user acceptance than polychrome mapping.

7.8.3 Evaluation

We next provide both measurements and user studies to evaluate the proposed unstructured color transformations.

Power Reduction

We implement the unstructured color transformation on .NET Compact Framework and C#, for Windows Mobile-based mobile embedded systems. We use the same 10 GUIs described in Section 7.3 to evaluate the implementation. As originals, we set the 10 GUIs in the vista9 (green) theme which has the lowest power consumption (See Figure 7.8). For each GUI, we perform transformations with three different settings.

- **Green (75%):** We perform a monochrome mapping only with color green as used in the theme, and we set λ as 75% since there is no solution when λ is 100%.
- **Green (50%):** To evaluate the impact of λ , we perform another monochrome mapping with λ as 50%.
- **Color64 (100%):** We also perform a polychrome mapping, in which each of the R , G , and B components has four intensity levels resulting 64 different colors in total. In this case, we set λ as 100%.

One original GUI and three automatically generated GUIs are shown in Figure 7.9. Figure 7.10 shows the average power reduction of the 10 GUIs generated using the three transformations. As shown in the figure, each transformation achieves over 75% power reduction. Optimization effort λ does not make much difference in power reduction although it results huge difference in their appearances.

Computation Time

As described in Section 7.8.2, intensity level has a huge impact on the computation time of the rank-based mapping algorithm. We perform experiments on the N85 mobile phone to study the tradeoff between computation time and power reduction by changing the number of intensity level in color space (See Figure 7.11). We investigate four different cases with the number of the intensity level of each color equals to 16, 8, 4, and 2. As shown in the figure, using a coarse granularity, or small intensity level, will dramatically reduce computation time while achieving comparable power reduction. The reason is that every benchmark in the experiment has a white background that contains most pixels of the GUI and the majority of power reduction is achieved by mapping the white background into black. Also, we notice that even when intensity level is 4, or 64 different colors in total, the computation time of rank-based mapping is in the order of one hundred milliseconds, which is not fast enough to be integrated into the operating system and for run-time GUI transformation. The monochrome mapping has a much longer computation time of approximately 10 seconds. This is because there is no efficient linear programming solver for the N85 mobile phone. Therefore, color transformation cannot be done in real time on a mobile phone. But we can still use it as a CAD tool for GUI designers.

User Evaluation

We used the same apparatus setup and procedure to perform user evaluation for the five color schemes. We apply these schemes to the 10 GUIs described in Section 7.3, load them to the OLED module described in Section 7.3, show them to our subjects one by one, and ask them to score them according to their fondness in 1 to 5 scale (5 means “like the most”). During the study, the subjects were not aware the power of the schemes. The study was conducted in a room with typical lighting condition. Since our color transformation

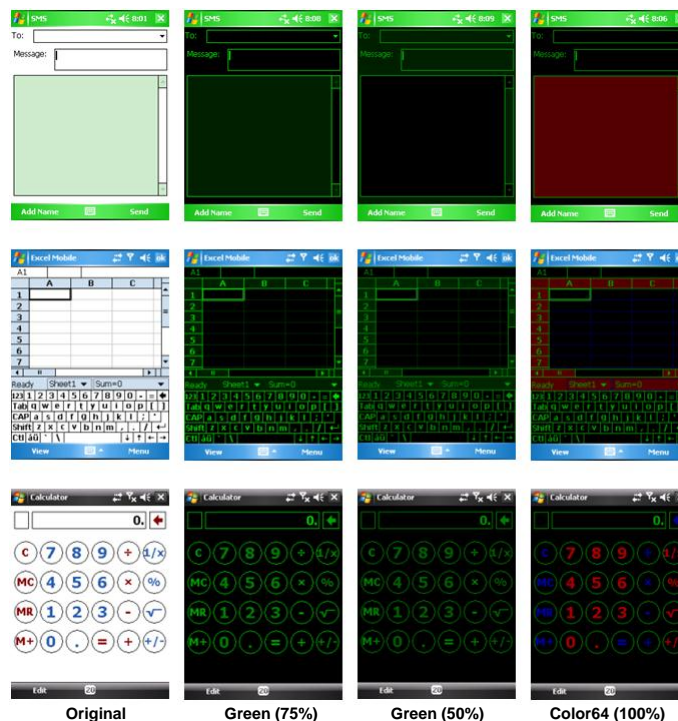


Figure 7.9 : Unstructured transformed GUIs with different settings.

guarantees that the color difference of any two colors in transformed GUIs is approximately the same as the color difference of the two corresponding original colors, the readability of the transformed GUIs would not be much worse than that of the original GUIs even under the sun.

Since the original theme gets the highest score by all the 20 subjects, we use it as the baseline to normalize the review scores of other themes. Fig. 12 presents the evaluation results. As shown in the figure, monochrome mapping is more acceptable to users than polychrome mapping. Between the two monochrome transformed GUI color schemes, the one that enforces tighter perception constraints gets higher scores. Based on the results, we have the following findings.

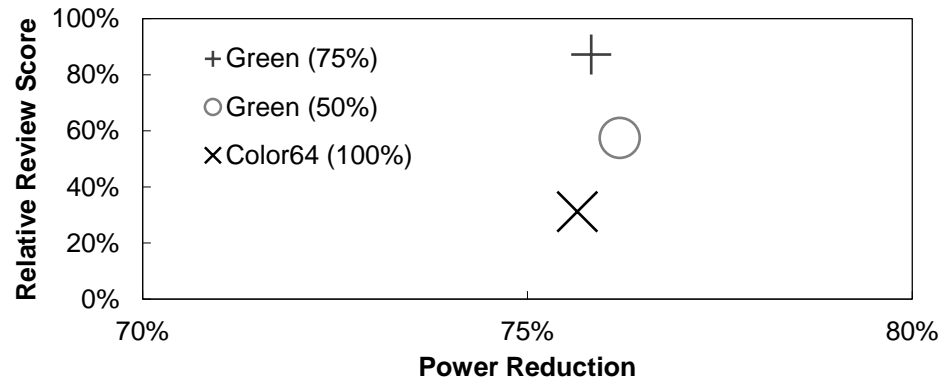


Figure 7.10 : Power reduction vs. relative review score.

- *People prefer to use monochrome mapping rather than polychrome mapping;*
- *Tighter perception constraints may lead to higher user acceptance with comparable power reduction.*

Furthermore, after the study, we asked the subjects whether they prefer the low-power schemes over the originals, given the 75% power reduction in the display. 18 out of 20 subjects said Yes.

7.8.4 Discussion

We next discuss two practical issues related to color transformation for power saving.

Large Number of Colors: The unstructured color transformation identifies an individual mapping for each color and the computation time is highly correlated to the number of colors. When there are a large number of colors in a GUI, the computation time can be very high. A reasonable solution is to identify one single mapping for all colors in a GUI. We examine a linear transformation, i.e.,

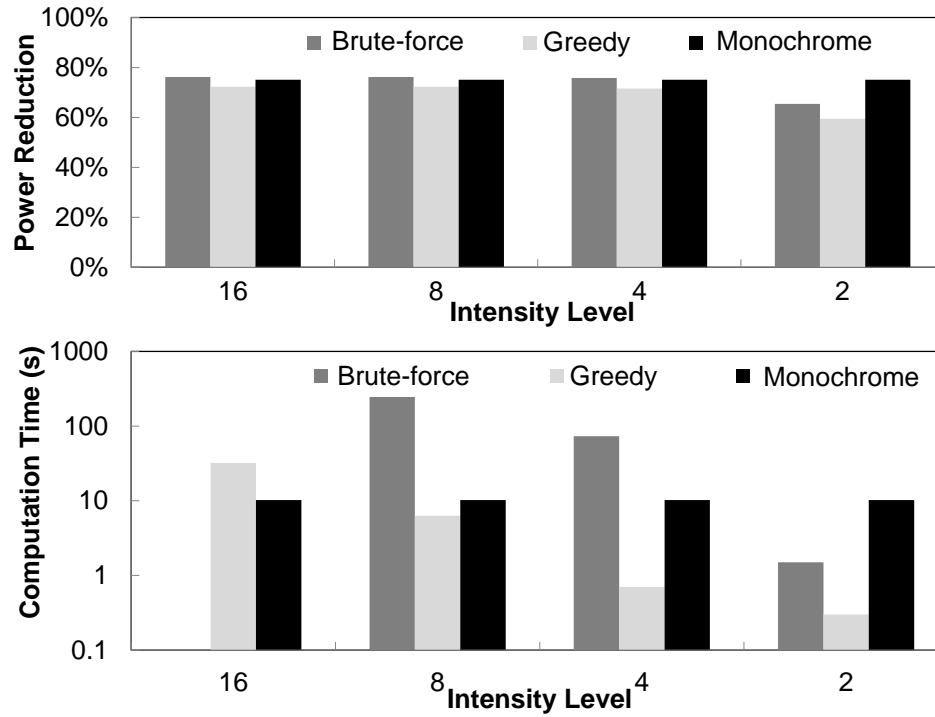


Figure 7.11 : Power reduction and computation time with various color level.

$$(L_i^{*'}, a_i^{*'}, b_i^{*'})^T = \mathbf{A}(L_i^*, a_i^*, b_i^*)^T + \mathbf{b}.$$

In this case, 12 variables, 9 in \mathbf{A} and 3 in \mathbf{b} , determine the linear transformation, which is favorable when the number of colors is huge. The challenge of this approach is to find proper \mathbf{A} and \mathbf{b} so that all the constraints in the minimization problem are satisfied. When the smallest singular value of \mathbf{A} is larger than or equal to λ , all the color distance constraints are guaranteed. However, whether the transformed colors are valid, e.g., $0 \leq R_i' \leq 1$, still needs to be checked individually.

Multiple GUIs in One Application: We presented the proposed color transformation in the context of a single GUI. However, it is critical to maintain a consistent color scheme

for all the GUIs in the same application. We have extended the proposed solutions to address this need. In particular, we choose to count the pixels of each color in all the GUIs, and minimize the expected power consumption based on the expected duration of the appearance of each GUI. The statistics of the duration can be either estimated by the designer at the design time or collected by the operating system or application itself from real usage.

Offline Color Mapping: As described in Section 7.8.3, the computation time of color mapping is too long to be executed in real time on a mobile phone. On the contrary, the time cost of color counting and color repainting is relatively small, i.e., in the order of tens of milliseconds. Therefore, OS should complete the color transformation process asynchronously. First, OS conducts color counting while the original GUIs are being used and obtains a historical color histogram. Second, using such a historical color histogram, OS executes the color mapping algorithms offline to obtain the optimized color scheme. Finally, OS is able to apply the computed new color scheme to regenerate the GUIs in the future.

7.9 Summary

In this chapter, we provided models for efficient and accurate power estimation for OLED displays, at pixel, image, and code levels, respectively. The pixel-level model built from measurements achieves 99% accuracy in power estimation for 300 benchmark images. By aggregating the power of a small group of pixels instead of all the pixels in an image, our image-level power model reduces the computation cost by 1600 times, while achieving 90% accuracy in power estimation. Our code-level model utilizes specification of the GUI objects to calculate the power consumption, which guarantees 95% accuracy.

The three power models we presented require different inputs and provide different

tradeoff between accuracy and efficiency. Therefore, we intend them for implementation and use at different system components. The pixel and image-level models are best for implementation in hardware or the operating system because they need to access the frame buffer. In contrast, the code-level model is best for implementation in application software or GUI development kits due to its dependence on knowing the composition of GUI object. All three models can provide power estimation for the system to better manage and optimize energy consumption, for the end user to make better tradeoff between usability and battery lifetime, and for GUI designers to design energy-efficient GUIs.

Based on the power models, we present a methodology of GUI optimization for saving power on OLED-based displays color transformation. We have the following findings.

- Using appropriate background/foreground color schemes may save a lot of power without sacrificing user satisfaction. For example, black/green is a low-power scheme with high user acceptance.
- Color transformation is able to reduce power by as much as 75% with little degradation in user satisfaction. And our user study shows that most of participants are indeed willing to sacrifice their satisfaction to save power.
- In unstructured color transformation, color schemes generated by monochrome scheme have better user satisfaction than the ones generated by the polychrome scheme.

The proposed methods can be employed at different stages during the lifetime of a mobile GUI. Faster and more efficient methods can be used at run-time to transform GUIs real-timely and can be carried out by either the operating system or the application. Slower but more flexible methods can be used by GUI designers to identify designs that are both energy-efficient and aesthetically attractive.

Chapter 8

A Color-Adaptive Web Browser for Mobile OLED Displays

Displays based on organic light-emitting diode (OLED) technology are appearing on many mobile devices. Unlike liquid crystal displays (LCD), OLED displays consume dramatically different power for showing different colors. In particular, OLED displays are inefficient for showing bright colors. This has made them undesirable for mobile devices because much of the web content is of bright colors.

To tackle this problem, we present the motivational studies, design, and realization of Chameleon, a color adaptive web browser that renders web pages with power-optimized color schemes under user-supplied constraints. Driven by the findings from our motivational studies, Chameleon provides end users with important options, offloads tasks that are not absolutely needed in real-time, and accomplishes real-time tasks by carefully enhancing the code base of a browser engine. According to measurements with OLED smartphones, Chameleon is able to reduce average system power consumption for web browsing by 41% and is able to reduce display power consumption by 64% without introducing any noticeable delay.

8.1 Overview

Displays are known to be among the largest power-consuming components on a modern mobile device [46, 67, 28]. OLED displays are appearing on an increasing number of mo-

bile devices, e.g., Google Nexus One, Nokia N85, and Samsung Galaxy S. Unlike LCDs where the backlight dominates the power consumption, an OLED display does not require backlight because its pixels are emissive. Each pixel consists of several OLEDs of different colors (commonly red, green and blue), which have very different luminance efficiencies. As a result, the color of an OLED pixel directly impacts its power consumption. While OLED displays consume close to zero power when presenting a black screen, they are much less efficient than LCDs in presenting certain colors, in particular white. For example, when displaying a white screen of the same luminance, the OLED display on Nexus One consumes more than twice of that by the LCD on iPhone 3GS [68]. Because the display content usually has a white or bright background, OLED displays are considered less efficient than LCDs overall. For example, Samsung reportedly dropped OLED for its Galaxy Tablet due to the same concern [69].

Our goal is to make web browsing more energy-efficient on mobile devices with OLED displays. Not only is web browsing among the most used smartphone applications according to recent studies [32, 42], but also most of today's web content is white (80% according to [70]). For example, about 60% of the average power consumption by Nexus One of browsing CNN mobile is contributed by the OLED display according to our measurement. Our algorithmic foundation is discussed in Chapter 7 that has demonstrated the potential of great efficient improvement in OLED displays by changing the color of display content, or color transformation. In order to effectively apply color transformation to web pages for energy efficiency we must answer the following questions.

First, *will web content providers solve the problem by providing web pages in energy-efficient color schemes?* Some websites allow the end users to customize the color schemes; and many provide a mobile version. However, our study of web usage by smartphone users revealed that close to 50% of web pages visited by mobile users are not optimized for

mobile devices at all [71]. Therefore, while it would be ideal if each website provides a version of content optimized for OLED displays, it is unlikely to happen at least in the near future. Moreover, our measurement of OLED smartphones showed that different OLED displays may have different color-power characteristics. There is no single color scheme that is optimal for all OLED displays.

Second, *will the problem be solved by configuring a web browser with the most energy-efficient color scheme?* Some web browsers already allow users to customize their color schemes. A user can simply choose the most energy-efficient scheme. This solution is, however, highly limited because the customized color scheme only affects the web browser itself, not the content rendered by it. Some web browsers allow a user-defined color style to set the color for texts, backgrounds, and links in the web content. Unfortunately, our study shows that an average web page visited by smartphone users employs about four colors for texts and three for backgrounds. If a single user-defined color style is applied, many web pages will be rendered unusable.

Finally, *is it feasible to implement color transformation at the mobile client?* Color transformation requires collecting statistics of color usage by pixels in real-time, transforming each color, and applying the new color to web page rendering in real-time. Because both the number of pixels ($\sim 10^5$) and the number of colors ($\sim 10^7$) are large, a straightforward application of color transformation will be extremely compute-intensive (See Chapter 7) and, therefore, defeat the purpose of energy conservation.

By realizing Chameleon, we answer the last question affirmatively. The key to Chameleons success lies in a suite of system techniques that leverage the findings from our motivational studies. Chameleon applies color transformation to web contents with the following features. It applies consistent color transformations to web pages from the same website and applies different perceptual constraints in color transformations for web content of different

fidelity requirements. Chameleon constructs the power model of an OLED display without external assistance in order to achieve device-specific optimal color transformation. It allows end users to set their color and fidelity preferences in color transformation. Finally, Chameleon only performs the absolutely necessary tasks in real-time and aggressively optimizes the real-time tasks for efficiency. Our evaluation shows that Chameleon can reduce the average system power consumption during web browsing by 40% without introducing any user noticeable delay on OLED smartphones, and the transformed web pages provided by Chameleon are well accepted by users.

In designing and realizing Chameleon, we make the following contributions:

- Three motivational studies that lead to the design requirements of Chameleon (Section 8.3).
- The design of Chameleon that extensively leverages the findings from the motivational studies in order to meet the design requirements (Section 8.4).
- Efficient realizations of Chameleon based open-source web browsers, including Android Webkit and Fennec (Section 8.5).

8.2 Background and Related Work

We next provide background and discuss related work.

8.2.1 Web Browser

Modern web browsers render a web page through a complicated process teemed with concurrency and data dependency. Figure 8.1 shows the workflow of a web browser. To open a web page, the browser first loads the main HTML document and parses it. When parsing the HTML document, more resources such as other HTML documents, Cascading Style

Sheets (CSS), JavaScripts, and images, may be discovered and then loaded. These two iterative stages are *Resource Loading* and *Parsing* in Figure 8.1. In *Parsing*, the browser manipulates objects specified by HTML tags in the web page using a programming interface called Document Object Model (DOM). These objects are therefore known as DOM elements and are stored in a data structure called the *DOM tree*.

In *Style Formatting*, the browser processes CSS and JavaScripts to obtain the *style* information, e.g., color and size, of each DOM element and constructs a *render tree*, the visual representation of the document. Each node in the render tree is generated from a DOM element to represent a rectangular area on the screen that shows the element. The style information of each DOM element is stored as *properties* of the corresponding node in the render tree.

Then, in *Layout Calculation*, the browser computes the layout and updates the position property of each node in the render tree based on the order of these nodes. Chameleon utilizes position and size properties of image nodes to identify which pixels in the current screen belong to images. This is discussed in Section 8.4.4.

Finally, in *Painting*, the browser calls a series of paint functions to draw the nodes of the render tree with layout onto a bitmap in the *frame buffer* to represent the screen and each paint function covers a group of adjacent nodes on the screen. Chameleon catches these paint functions to identify which regions of the current screen are updated. Nodes of different types are painted using different libraries. Images are painted using various image libraries depending on the image format. GUI objects, e.g., links, forms, and tables, etc, are painted using a graphics library that handles basic geometry elements such as points, lines and rectangles. Chameleon realizes color transformation by modifying the interfaces of images and graphics libraries.

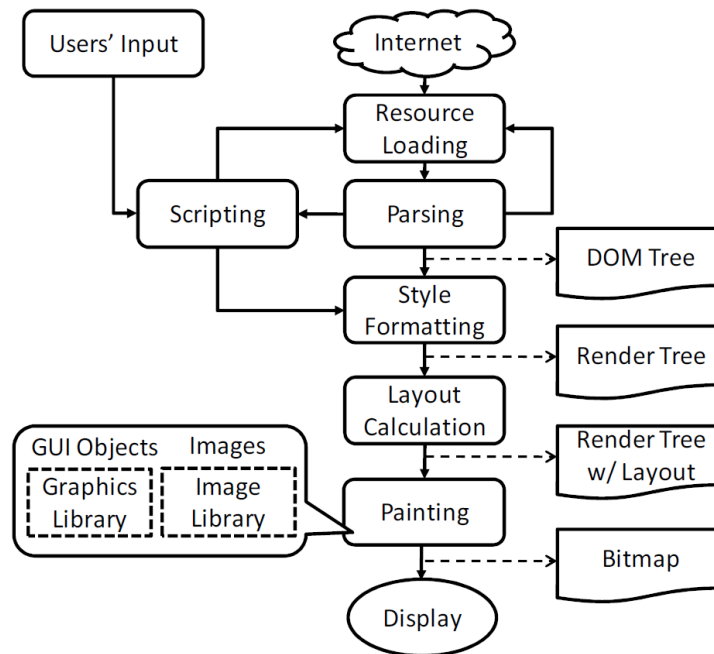


Figure 8.1 : A simplified workflow of a web browser.

8.2.2 Related Work

Chameleon is motivated by prior work in display power management and leverages its algorithmic solutions. Most of the related work has been discussed in Chapter 7 already. There also exists a lot of research effort that adapts web pages for mobile displays either manually [72] or automatically [73]. Its goal to better fit web content into the small mobile display is very different from ours in conserving energy. Moreover, its solutions usually involve the modification of layout, instead of color.

8.3 Motivational Studies

We next report three studies that directly motivated the design of Chameleon: the OLED displays, web usage by smartphone users, and user preferences in web page color transformation.

8.3.1 OLED Display Power

Using the measurement procedure described in Chapter 7, we derived the pixel power consumption of three OLED smartphones, i.e., Nexus One, Galaxy S, and Nokia N85. Figure 8.2 shows the power consumption by each color component of an OLED pixel at various linear RGB values. Strictly speaking, this OLED pixel is a logic pixel and refers to the corresponding pixel in the displayed bitmap. The displays of Nexus One and Galaxy S utilize the PenTile RGBG pattern in which a physical pixel has either red and green components or blue and green components. A mapping algorithm has to be used to map a bitmap image to this PenTile RGBG pattern. Moreover, the pixel of N85 consumes much higher power because it is approximately twice as large as a pixel in Nexus One or Galaxy S. Importantly, we make the following observations as related to the design of Chameleon.

First, the power consumption of an OLED pixel is a linear function of its linear RGB values, or, more accurately, the gamma decoding result from its sRGB values. The linear regression fitting statistics R^2 of all the three devices are over 0.95. The reason is that the power consumption of an OLED pixel is a linear function of the current passing through it. And the current passing through an OLED is also a linear function of the gamma decoding result from its sRGB value [74]. The linearity simplifies the construction of the OLED power model as is necessary in Chameleon.

Second, different displays have different power characteristics. Particularly, relative power consumption by red and green colors varies significantly from device to device. In

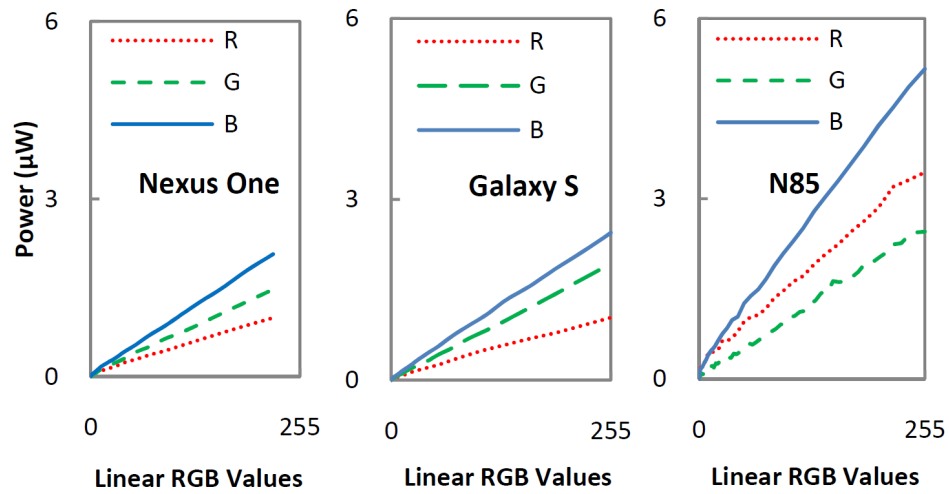


Figure 8.2 : Pixel power models of three OLED displays. The power consumption by each color component of a pixel is a linear function of its corresponding linear RGB value.

N85, green is more power efficient than red; in Nexus One and Galaxy S, the opposite is true, as shown in Figure 8.2. This means that the most energy-efficient color scheme on N85 may be not most energy efficient on Nexus One or Galaxy S. This observation motivates a device-specific color transformation that employs a device-specific OLED power model.

Finally, chromaticity makes a big difference even the lightness is identical. Figure 8.3 presents the color power model of the OLED display used by Galaxy S in CIELAB color space, in which power consumption of each color is normalized by that of white. Each curved surface in the figure represents of the colors of the same lightness (L^*). Unsurprisingly, the figure shows that the power consumption of a color will increase when lightness increases given the same chromaticity (a^* and b^*). This indicates that one should use darker colors to reduce power consumption of an OLED display, which has been widely known and practiced already. More importantly, however, Figure 8.3 also shows that the power

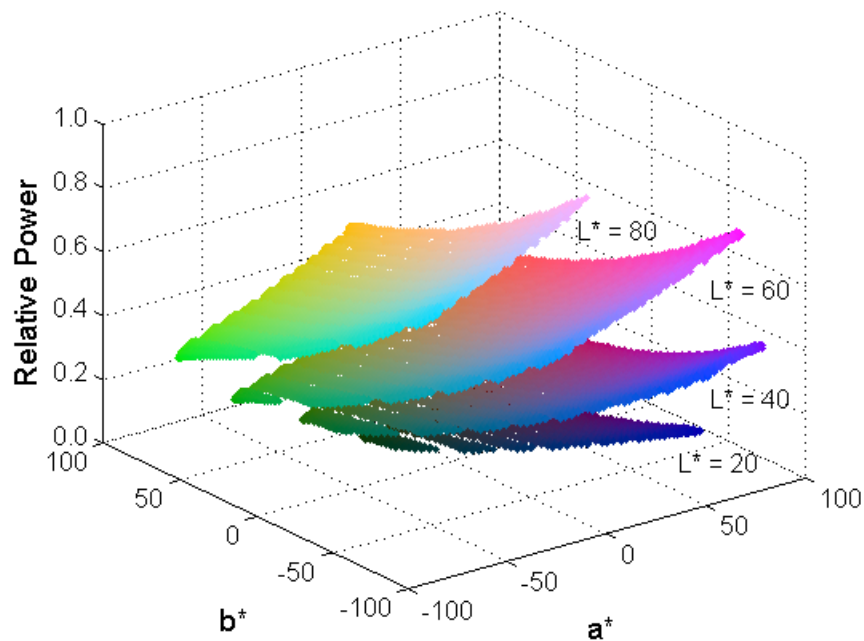


Figure 8.3 : OLED color power model in CIELAB. Given the lightness, or L^* , power consumption can differ as much as 5X between two colors with different chromaticity.

consumption difference can be as high as five times between two colors of the same lightness, or on the same curved surface. This finding indicates that changing chromaticity is another effective way to reduce OLED display power and color transformation will be more effective than display darkening.

8.3.2 Web Usage by Smartphone Users

By studying web browser traces from 25 iPhone 3GS users over three months [42] (called LiveLab traces below), we make the following observations as related to the design of Chameleon.

Browsing Behavior

First, mobile users still visit web pages that are not optimized for mobile devices. While a website can potentially provide an OLED-friendly version, e.g., with dark background, our data show that approximately 50% of web pages visited by mobile users are not optimized for mobile devices at all [71]. Therefore, one cannot count on every website to provide an OLED-friendly version to reduce display power. This directly motivates the necessity of client-based color transformation.

Second, a small number of websites account for most web usage. We find that the 20 most often visited websites of each user contribute to 80%-100% (90% on average) of the web usage by the same user, as shown in Figure 8.4(top). Therefore, it is reasonable to maintain a color transformation scheme for each of the 20 websites and to use a universal transformation scheme for, or simply not transform, the other websites. This is the key rationale behind our design decision to maintain color consistency per website in Chameleon (Section 8.4.1).

Web Content

We further analyzed the web pages visited by the 25 iPhone 3GS users with the following findings.

First, 65% of the pixels in the web pages visited by the 25 users over three months are white. This is lower but close to the claim made by [70] that white color takes as high as 80% of web content. As OLED displays are power-hungry for white, they can be less energy-efficient overall than LCDs. Color transformation is therefore very important to improve the energy efficiency of mobile devices with an OLED display.

Second, an average web page from the LiveLab traces includes very rich styles, about four colors for texts and three for backgrounds. On the contrary, a user defined Cascading

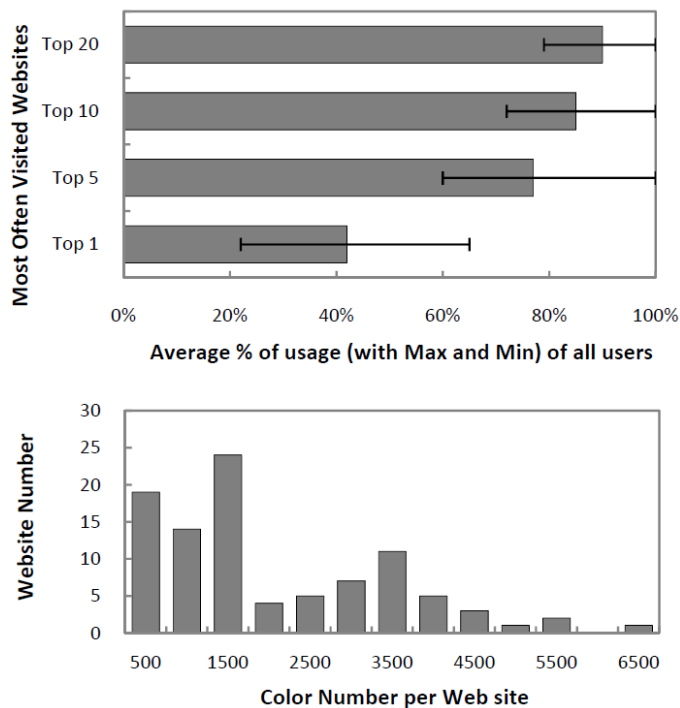


Figure 8.4 : Web usage by smartphone users extracted from the LiveLab traces. The most visited websites of each user account for a high percentage of his web usage (top); a website uses a very small number of colors, about 1500 colors on average and 6500 at most, compared to all the available colors (10^7) (bottom).

Style Sheet (CSS), or browser color profile only defines one color for all texts and one color for all backgrounds. As a result, using a user defined CSS to format an entire web page will significantly reduce color numbers of the web page and compromise aesthetics or even impact the web usability, as exemplified by Figure 8.4(bottom).

Third, the number of colors used by a website is very small ($\sim 10^3$) compared to the number of all colors ($\sim 10^7$) supported by a modern mobile device. The number of colors has a significant implication on the computational cost of color transformation not only because each color has to be transformed but also because the occurrences of each color

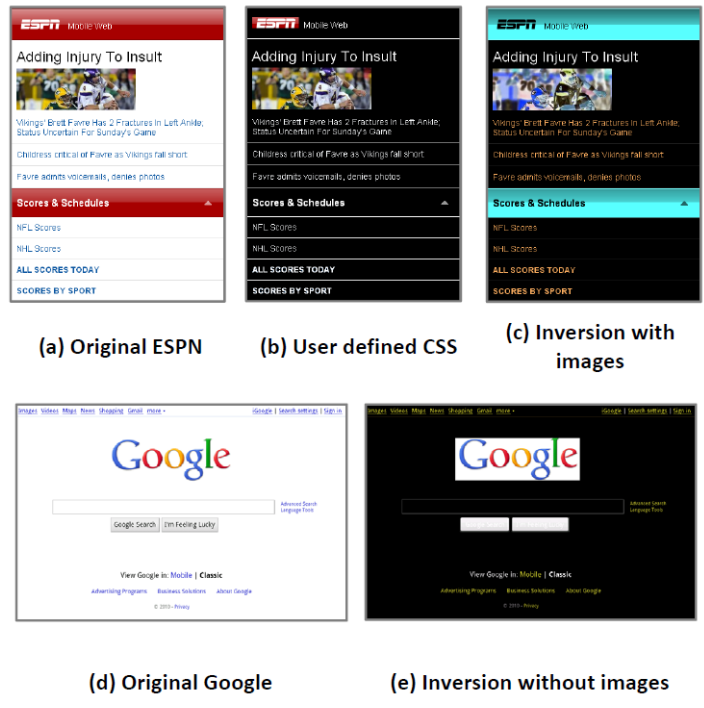


Figure 8.5 : To apply a user defined CSS will lose color information of different CSS boxes (b); Inversion with images makes them unusable (c); Inversion without the background images in two buttons makes them undistinguishable from inverted texts (e); Inversion without the foreground logo image makes it clash with the inverted background color (e).

must be counted for an optimal transformation. The number of colors is as many as 2^{24} or 16,777,216, in a modern web browser in which each of RGB components is represented using eight bits. While this number is prohibitively high, we find that websites accessed by LiveLab users only have about 1500 colors on average, with the maximum being 6500, as shown in Figure 8.4(bottom). This observation is key to Chameleons feasibility and design of color contribution collection.

Finally, a modern web page contains visual objects of different fidelity requirements in color transformation. Videos and many images require color fidelity. That is, their color

cannot be modified arbitrarily. An analysis of the LiveLab traces shows that such fidelity-bound objects are abundant in web pages accessed by mobile devices. For example, images account for about 15% of the pixels for an average web page in the trace. As a result, blindly changing the color of a pixel without considering the fidelity requirement of a visual object, e.g., inverting all pixels in the page (Figure 8.5(c)), can be unacceptable. In contrast, GUI objects such as backgrounds, texts, and forms only require usability. Their colors can usually be modified arbitrarily as long as the transformed colors are still distinguishable and aesthetically acceptable.

Similarly, images on a web page may have different fidelity requirements too. Foreground images are images specified by HTML IMG tags. For most foreground images, such as photos, fidelity matters. Significant color changes will render them useless as shown in Figure 8.5(c). However, logo images, a common, special kind of foreground images, usually have the same background color as the adjacent GUI objects by design. As a result, if a logo image is not transformed along with the GUI objects, the background color of the logo image will clash with the inverted background color, as shown in Figure 8.5(e). Therefore, Chameleon allows the user to choose if it is important to keep the fidelity of logo images. If not, Chameleon will treat logos along with the GUI objects in color transformation. Background images are images specified by the CSS BACKGROUND-IMG property to serve as the background of a CSS box. In the LiveLab traces, 23% of web pages contain background images. Because background images are usually designed with a color pattern consistent with adjacent GUI objects, color transformation without background images may make the background images undistinguishable from adjacent GUI objects with transformed colors. As shown in Figure 8.5(e), the background images of the two buttons are undistinguishable from inverted texts. Therefore, Chameleon treats background images along with the GUI objects in color transformation.

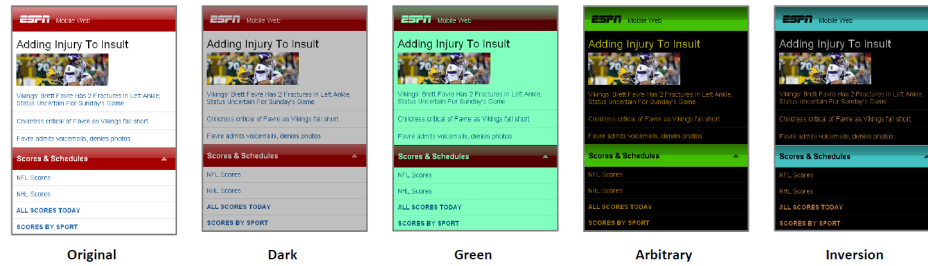


Figure 8.6 : Original ESPN homepage and its four color transformations generated by four algorithms used in the user study.

8.3.3 User Preference of Color Transformation

While color transformation can potentially reduce the power consumption by an OLED display, it must be performed with user acceptance considered. We employ a series of user studies to investigate how users accept color transformation of web pages. We recruited 20 unpaid volunteers to use Nexus One to review a series of web pages with typical office lighting. The participants were asked to score each web page by 1 to 5 with 1 being the least acceptable.

The web pages used in the experiment include four pages from the five top mobile websites, i.e., CNN, Facebook, Google, Weather, and ESPN. For each web page, we present the original and four color-transformed versions, as shown in Figure 8.6. Thus, there are 100 pages in total. The color transformations include:

- *Dark*: Lightness of all the colors is uniformly reduced, i.e., $R' = \lambda R$, $G' = \lambda G$, $B' = \lambda B$, in which $\lambda \in [0, 1]$. This is similar to what a user would experience with modern smartphones with LCDs and OLED displays.
- *Green*: Lightness of each of the (R, G, B) components is reduced separately, i.e., $R' = \lambda_R R$, $G' = \lambda_G G$, $B' = \lambda_B B$, in which $\lambda_R, \lambda_G, \lambda_B \in [0, 1]$. Green component is

reduced least because green is the most power efficient color.

- *Inversion*: All the colors are inverted by replacing each of the (R, G, B) components with its complement multiplying a scalar $R' = (1 - \lambda)R$, $G' = (1 - \lambda)G$, $B' = (1 - \lambda)B$, in which $\lambda \in [0, 1]$. The rationale of this transformation is that most pixels from web pages are white, according to Section 8.3.
- *Arbitrary*: Lightness and chrome of each color are changed to minimize display power consumption (See Chapter 7).

The transformations are only applied to GUI objects, background images and non-photo foreground images. Each of these algorithms includes one or more controllable parameters, e.g., lightness reduction ratio λ in *Dark*. Such parameters affect both perception constraints and power consumption of transformed web pages. For a fair comparison, we adjust the parameters in each transformation so that the same usability constraint discussed in Chapter 7 is used in all four algorithms.

Dark, *Green*, and *Inversion* are linear transformations as discussed in Chapter 7. As a result, *Arbitrary* promises the biggest power reduction. Not, surprisingly, the power reduction by *Dark*, *Green*, *Inversion*, and *Arbitrary* is 25%, 34%, 66%, and 72%, respectively, under the same perceptual constraint.

We have the following two findings by analyzing the scores by the participant. *First, different users prefer different transformations for a website.* For each color transformation of a web page, we count the number of users who gave the highest score out of the four transformations. Thus, we have four numbers of user “votes” for each web page. Figure 8.7(top) shows the four numbers for the homepage of each website used in our study. As shown in the figure, given a website, each transformation gets some votes. The numbers of votes for all four transformations are actually not dramatically different for most websites.

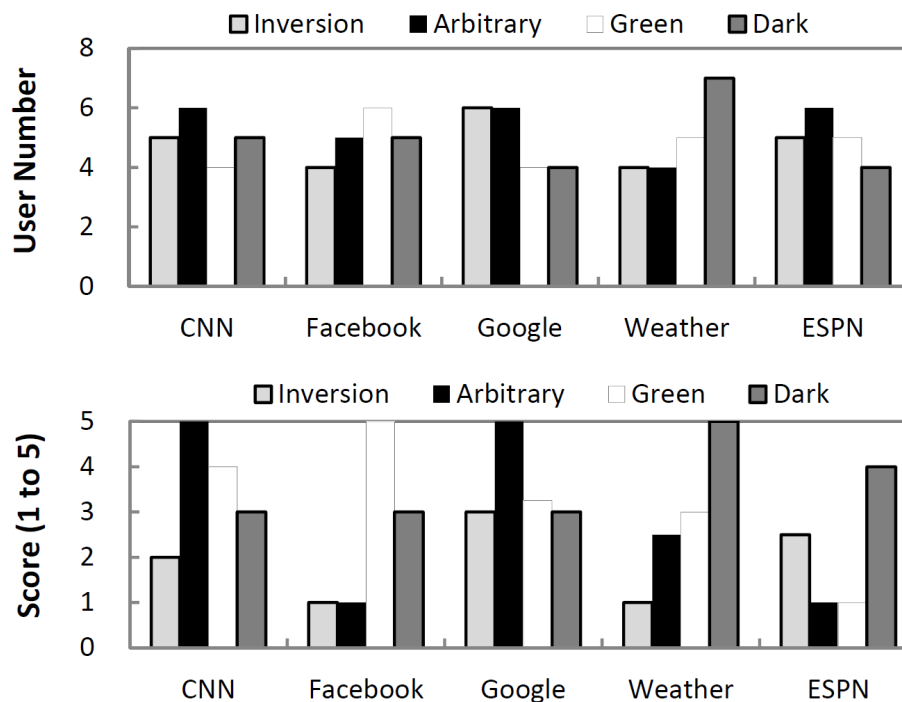


Figure 8.7 : User study results. User favorite algorithm (top); average score of each website by User 1 (bottom).

Therefore, it is important to give end users options in selecting the color transformation algorithm, instead of providing one for all users.

Second, even the same user may favor different color transformations for different websites. Figure 8.7(bottom) shows the average scores of each website by User 1 as an example. As shown in the figure, User 1 prefers Arbitrary for CNN and Google, Green algorithm for Facebook, and Dark for Weather and ESPN. Similar results are found for other users. This, again, motivates Chameleons design to give end user options in selecting the color transformation algorithm and selecting it per website.

Finally, as shown in Chapter 7, users may want different tradeoff between energy saving and perceptual constraints, depending on the available battery level. They are willing to

sacrifice more aesthetics when the need for energy saving is urgent. This again motivates that choices should be given to the end user.

8.4 Design of Chameleon

We next describe the design of Chameleon as motivated by the results from the motivational studies.

8.4.1 Key Design Decisions

The results from the motivation studies lead us to make the following major design decisions for Chameleon.

Treat GUI Objects and Images Differently: Chameleon only applies display darkening to foreground images in order to preserve fidelity. It applies color transformation only to GUI objects, background images, and possibly logo images depending on the user choice, according to findings reported in Section 8.3.2. A side benefit of only applying darkening to foreground images is that it works very well with incrementally rendered large photos.

Keep Color Consistency per Website: Web pages from many websites often employ the same color scheme to maintain a consistent user experience. Chameleon keeps this consistency by applying the same color transformation to all pages from a website and does so for the top 20 websites of the user. We opt against color consistency for multiple websites because a user may prefer different color transformations for different websites, as found in Section 8.3.3. Moreover, the top 20 websites of a user accounts for most of her/his web usage according to Section 8.3.2.

Generate Device Specific OLED Power Model: As shown in Section 8.3.1, power models of different OLED displays are different. To make sure the transformed color scheme is optimized for each device, Chameleon builds an OLED power model for the device it runs

using power readings from the battery interface.

Calculate Color Maps Offline: Chameleon finishes the compute-intensive mapping optimization offline, in the cloud in our implementation, and only perform the absolutely necessary tasks such as color contribution collection and painting in real-time.

Give User Options: For each website, Chameleon allows a user to choose from linear and arbitrary transformations described in Chapter 7, to specify the color preference and perceptual constraint for the color transformation and to choose to transform logo images either by darkening or color transformation. Chameleon will have all color maps using all possible user options ready. As a result, the user will immediately see the effect of her selections without waiting. Our user study showed this is extremely useful for users to find out their favorite transformations.

8.4.2 Architecture

Now we provide an overview of the architecture of Chameleon. Without knowing the future, Chameleon uses the web usage by the user in the past to approximate that of the future. Therefore, Chameleon collects color information of web browsing and seeks to identify the color transformation for each color so that the average display power consumption of past web usage could be minimized. Chameleon then applies the color transformation to future web browsing to reduce the display power consumption.

Suppose a user has been browsing a website for time T . Then the energy consumption by the OLED display in time T is:

$$E = \int_0^T \sum_{i=1}^n num_i(t) P_{pixel}(color_i) dt = \sum_{i=1}^n P_{pixel}(color_i) \int_0^T num_i(t) dt.$$

where $color_i = (R_i, G_i, B_i)^T$, $i \in \{1, 2, \dots, n\}$, are the n colors supported by the browser and $num_i(t)$ is the pixel number of $color_i$ in the display at time t . Notably, the integral

factor $\int_0^T num_i(t)dt$ considers both the spatial and temporal contributions by a color. It naturally gives a larger weight to a web page that is viewed for a longer time. The integral factor can also “forget” past record by including a weight that diminishes for contributions from the distant past.

As shown in Section 8.3.1, the power consumption of an OLED pixel is a linear function of its linear RGB values, i.e.,

$$P_{pixel}(color_i) = (a, b, c) \cdot color_i = a \cdot R + b \cdot G + c \cdot B.$$

Note that the constant factor in the original linear function is not included because it is independent from the color. Thus, the color-dependent energy consumption can be rewritten as

$$E = \mathbf{M} \cdot \mathbf{X}' \cdot \mathbf{D} = (a, b, c) \cdot \begin{pmatrix} R'_1 & \cdots & R'_n \\ G'_1 & \cdots & G'_n \\ B'_1 & \cdots & B'_n \end{pmatrix} \cdot \begin{pmatrix} \int_0^T num_1(t)dt \\ \vdots \\ \int_0^T num_n(t)dt \end{pmatrix}.$$

\mathbf{M} is the OLED power model; \mathbf{X}' is a matrix called the color map with the i -th column being the transformed color for $color_i$; \mathbf{D} is the color contribution vector for the website with each entry corresponding to the contribution from $color_i$, $\int_0^T num_i(t)dt$.

To minimize the energy consumption, E , Chameleon must construct the power model, \mathbf{M} , gather data to derive \mathbf{D} , calculate \mathbf{X}' with user-supplied options, and apply it to change the colors of future web pages. Therefore, Chameleon consists of four modules that interact with a browser engine, as illustrated in Figure 8.8.

- A *model construction* module generates a power model, \mathbf{M} , of the OLED display of the mobile device, using the smart battery interface (See Chapter 5.

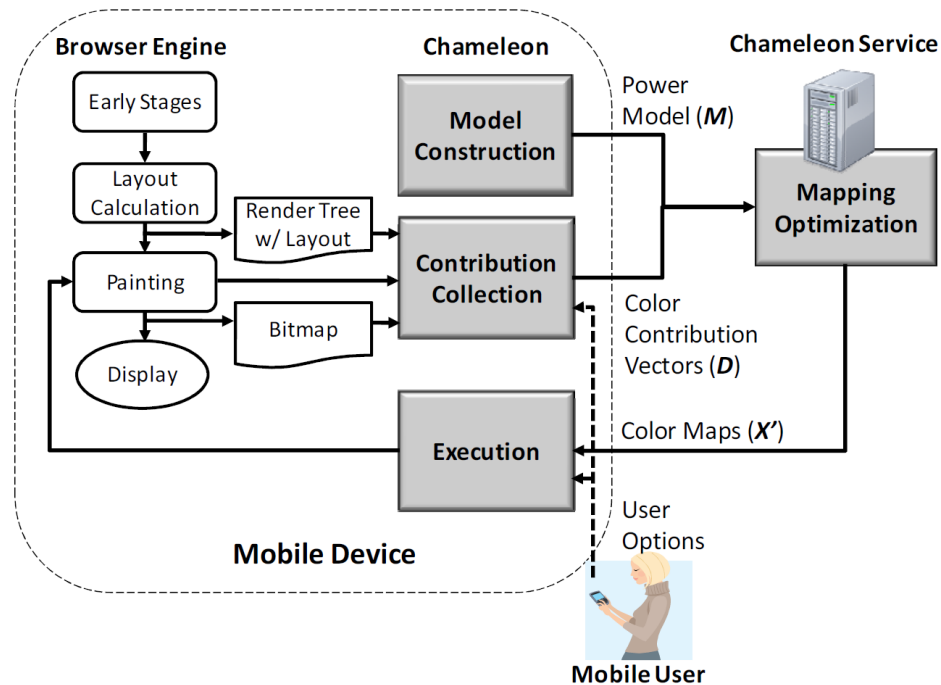


Figure 8.8 : Architecture of Chameleon and how it interacts with the browser engine.

- A *contribution collection* module gathers a color contribution vector, \mathbf{D} , for each website from the *Layout Calculation* and *Painting* stages of the browser engine in an event-driven manner.
- An offline *mapping optimization* module computes the color map \mathbf{X}' based on \mathbf{M} and \mathbf{D} . Note that Chameleon computes the color maps for all possible user options so that the user can immediately see the impact of a change in user options.
- An *execution* module applies the color map \mathbf{X}' to transform colors in a web page.

Out of the four modules, contribution collection and execution have to be executed in real-time. We next discuss each module in detail.

8.4.3 Color Power Model Construction

Chameleon constructs the model \mathbf{M} automatically without any external assistance, allowing for immediate deployment. The key idea is to employ the limited power measurement capability provided by the smart battery interface on modern mobile devices.

Chameleons model construction module shows a series of benchmark images on the OLED display and measures the corresponding power consumption of the whole system from the battery interface. A benchmark image has all pixels showing the same color. Because all pixels have the same color, the system power consumption would be

$$P_{system} = N \times (aR + bG + cB) + P_{black},$$

where N is the total number of pixels and P_{black} is the system power consumption when the display is showing a black screen. When all the benchmark images have been shown and the corresponding power consumption numbers have been collected, the module applies linear regression to obtain the values of a , b , and c .

The model can be constructed once and calibrated over the lifetime of the device. The calibration is necessary because OLED displays are known to age and therefore exhibit different color power properties over years. However, since the aging process is slow, the model only needs to be re-calibrated using the same process a few times per year, without engaging the user.

8.4.4 Color Contribution Collection

Chameleon generates the color contribution vector, \mathbf{D} , of GUI objects, background images, and possibly logo images (depending on the user choice). Recall that an element of \mathbf{D} is determined by how many pixels have the i -th color and for how long. Therefore, whenever

the display changes, contribution collection must determine how long the previous screen has remain unchanged, or time counting, and how many pixels in that screen are of the i -th color, or pixel counting. To process a large number of colors ($> 10^3$) and pixels ($> 10^5$) in real-time, Chameleon employs a suite of techniques to improve the efficiency of contribution collection.

Why must contribution collection be done in real-time? It would be much easier to use the browser history to record the URLs of visited web pages with a timestamp and examine the pages offline. The key problem with this off-line method is that it does not capture what actually appears on the display. Because often a small portion of a web page can be shown on the display and the user must scroll the page, zoom in, and zoom out during browsing, the problem will lead to significant inaccuracy in the color contribution vector, **D**.

Event-driven Time Counting

Chameleon leverages the paint functions of the browser engine to efficiently count time in an event-driven manner. The browser engine updates the screen by calling a paint functions in the *Painting* stage. The time when a paint function returns indicates the screen is updated and would be a perfect time to start contribution collection. However, it is very common that the browser engine calls a series of paint functions to paint multiple nodes in the render tree with layout, even for one screen update as perceived by the user. If contribution collection runs for every paint function, the overhead can be prohibitively high. Therefore, Chameleon seeks to identify a series of paint functions that are called in a burst and only runs contribution collection when the last of the series of paint function returns.

Chameleon employs a simple yet effective timeout heuristics to tell if a paint function is the last in a series. That is, if there is no paint function called after a timeout period, Chameleon considers the last paint function the last of a series. The choice of the timeout

value is important. If the timeout value is too small, contribution collection will be called frequently. If it is too large, many display updates will go unaccounted for.

To determine a reasonable timeout value, we performed a user study with ten mobile users to collect timing information of paint function calls. In the user study, each of the users freely browsed web for 30 minutes using an instrumented web browser on Nexus One. The instrumented browser records the time a paint function starts and completes, producing a 300-minute trace. Given a timeout value, the paint function calls in the trace can be grouped into series. We then calculated the timing statistics for the identified series. Inter-series interval is the time between the finish of the last paint function in a series and the finish of the last paint function in the next series. It is the time between two executions of contribution collection. Series duration is the time between the start of the first paint function in a series and the finish of the last paint function in the same series. It measures the time during which the screen updates will not be counted by Chameleon.

Figure 8.9 shows the box plots of inter-series interval and series duration generated using different timeout values. As shown in the figure, both inter-series interval and series duration have wide distributions. As a result, we should not only examine their averages but also their ranges. For inter-series interval, we are more interested in its lower percentile because short inter-series intervals mean frequent execution of contribution collection. When the timeout is 0.25 second, the 25th percentile of inter-series interval is approximately 0.5 second. In other words, it will be one out of four chances that the time overhead of contribution collection is over 10%. The overhead can be reduced to 2.5% with a one-second timeout. For series duration, we are more interested in its higher percentile because long series duration imply inaccuracy in the color contribution vector, **D**. When the timeout is 4 seconds, the 75th percentile of series duration is more than 18 seconds. In other words, it will be one out of four chances that the screen updating for at least 18 seconds will not be

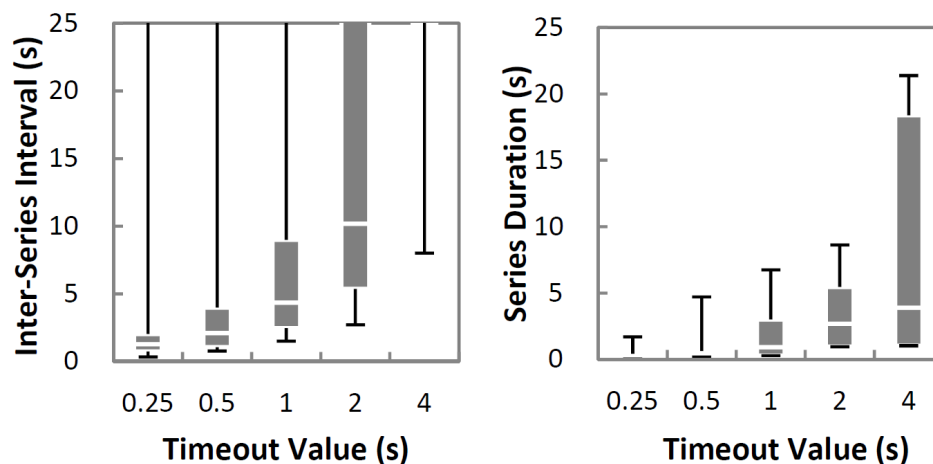


Figure 8.9 : Box plots for inter-series interval (left) and series duration (right) calculated using different timeout values. The bottom and top edges of a box are the 25th and 75th percentile; the white band in a box is the median, and the ends of the whiskers are maximum and minimum, respectively.

counted, which can introduce a considerable error in color contribution vector \mathbf{D} . The 75th percentile of series duration can be reduced to about 5 seconds with a two-second timeout.

Therefore, we consider a reasonable timeout should be between one and two seconds and we set the timeout as one second in the reported implementations.

Pixel Counting

Once contribution collection is called, it will count the number of pixels for each color. Because the numbers of both pixels and colors can be large, we design the module as follows to improve its efficiency in both computing and storage.

Chameleon obtains pixel information, or RGB values, from the frame buffer. We note that an obvious alternative is to traverse the render tree without layout to calculate the pixel

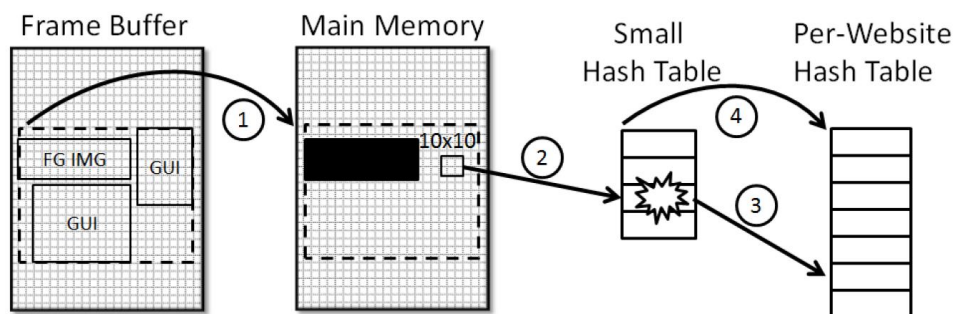


Figure 8.10 : How Chameleon counts pixels. (1) Copy the pixels from the frame buffer to the main memory; (2) Count the pixels in the main memory and save the results to a small Hash Table; (3) Merge the colliding entry of the small Hash Table to the per-website Hash Table; (4) Merge the entire small Hash table to the per-website Hash Table when all the pixels have been examined.

number of each color. However, the render tree contains no pixel information of images, which makes it impossible to count pixels in background image. Second, the overlapping of GUI objects makes it compute-intensive to count pixels from the render tree, thanks to a possibly large number of GUI objects in a web page. For example, counting the pixel number of all the colors in the web page shown in Figure 8.5(a) from the render tree with layout takes more than 200 ms, compared to less than 60 ms it takes to from the frame buffer.

Figure 8.10 shows the whole process of pixel counting. As in Step (1) of the figure, Chameleon copies the screen content from the frame buffer to the main memory in a stream and examines pixels in the main memory copy to minimize expensive frame buffer accesses. Moreover, Chameleon only reads from the frame buffer for the screen region that has been updated by the last series of paint functions. Because each paint function includes

parameters that specify a rectangle region in the screen on which it draws, Chameleon calculates the superset rectangle of these rectangles and only updates the super set to the main memory copy.

Chameleon excludes foreground images from the rectangular superset. Chameleon leverages two parameters of a paint function to identify the area of the superset rectangle that belongs to foreground images. One parameter indicates what low-level library to use and this parameter tells whether the pixels to be drawn are from images or GUI objects. The other parameter is the pointer of the image node in the render tree with layout. Using this pointer, Chameleon tells if the image is foreground or background by its position in the render tree with layout: a foreground image is an element node of the render tree while a background image is a property of an element node. In case of a foreground image, Chameleon reads its size and position and skips its pixels in pixel accounting.

Chameleon employs a Hash table to store the color contribution vector, \mathbf{D} , to reduce its storage requirement because the number of colors used by a website is orders of magnitudes smaller than that of all possible colors. To reduce the index collision and make an efficient Hash table, we choose to use a special key for the Hash Table, i.e., each of the RGB components is reversed in bit and then interleaved together. The rationale of such a choice is that, compared to lower bits, the higher bits of the RGB components are more effective to differentiate colors and should be used earlier by the Hash table. The Hash Table implementation only requires about 10 KB for a website because the number of colors used by a website is no more than a few thousands. When updating the Hash table, Chameleon can discount the existing record, instead of simply aggregating the new pixel counts into it, in order to “forget” web pages browsed in the distant history.

As in Step (2) of Figure 8.10, Chameleon counts pixels in the main memory copy by 10 pixels by 10 pixels (10×10) blocks and employs a small Hash Table with special collision

resolution to store the results for a block. The key of the small Hash table is the higher two bits of RGB components of each color; the table resolves a collision by merging the existing entry into the per website Hash Table, as in Step (3) of Figure 8.10. After all the blocks have been examined, Chameleon then merges the small Hash Table into the per-website Hash table, as in Step (4) of Figure 8.10. This design is motivated by the spatial locality of colors in web pages: the average color numbers in a 10×10 block is only 5 according to the LiveLab traces. By counting pixels block by block and using the small Hash table of a fixed size to store the block results, Chameleon reduces the number of writings into the per-website Hash Table.

Finally, Chameleon can further leverage the spatial locality of color to reduce the number of pixels to examine by sampling. That is, instead of counting every pixel, Chameleon can only count one pixel out of a block (e.g., 10×10). This pixel can be sampled from a fixed position or simply by random. This sampling technique makes a profitable tradeoff between the accuracy of D and the efficiency. As discussed in Chapter 7, a sampling rate as high as one out of 10×10 pixels can still yield very high accuracy (95%) in display power estimation, thanks to the spatial locality of color in web pages.

As will show in Section 6, the time overhead of the execution of contribution collection is at most 66 ms and 17 ms without and with sampling.

8.4.5 Color Mapping Optimization

Chameleon treats foreground images and GUI objects separately. It only darkens foreground images but transforms the color schemes of GUI objects, background images, and possibly logo foreground images (depending on the user choice), all under a perceptual constraint.

Given the power model, \mathbf{M} , the color contribution vector, \mathbf{D} , Chameleon calculates the

optimal color maps, \mathbf{X}' , offline for each website. Chameleon computes the color maps for all possible combinations of user options, including all possible perceptual constraints. With all color maps at hand, Chameleon allows the user to change the options and see the resulting color scheme immediately. There are 20 color maps for each website in our implementation.

8.4.6 Color Transformation Execution

Given a color map \mathbf{X}' , Chameleon replaces each original color with its corresponding transformed color for GUI objects, background images, and possibly logo images (depending on the user choice). Instead of transforming pixel by pixel, Chameleon modifies the parameters of paint functions to transform the entire region updated by the function at once. In the *Painting* stage, when a geometry primitive is painted with a color (R, G, B) , Chameleon uses \mathbf{X}' to find the transformed color (R', G', B') and passes (R', G', B') to the low-level graphics library to continue the painting process.

For foreground photo images, the transformation only involves darkening, or reducing brightness, i.e., each pixel (R, G, B) in the image becomes $(\lambda R, \lambda G, \lambda B)$, $\lambda \in [0, 1]$ in the transformed image. This darkening operation is integrated into image decompression such that the RGB components of each pixel are multiplied by λ right after they are calculated by the image library.

8.5 Implementation of Chameleon

We next report our implementations of the Chameleon design described above. The first choice we faced was whether to implement it as an add-on to existing web browsers or to directly modify the source code of a browser.

Many web browsers support add-ons to enhance functionality as either a plug-in or an

extension. Plug-ins are provided as libraries by third party software companies to handle new types of contents, such as Flash and QuickTime movies, and unable to impact on how a web browser renders a web page. Therefore, Chameleon cannot be implemented as a plug-in. Extensions are designed by users using XML and JavaScripts to add or change browser features they prefer. An extension can use a JavaScript to change the color of any GUI object. We opt not to implement Chameleon as an extension for three reasons. First, such an extension is unable to generate a color contribution vector as described in Section 8.4.4 because JavaScripts have no access to the render tree with layout to obtain layout information of GUI objects and images. Second, an extension cannot transform color of background images because JavaScripts do not affect how images are decompressed. Third, an extension-based implementation will be prohibitively expensive because the extension needs to traverse the whole render tree to execute color transformation. For example, to perform a color inversion on CNN mobile home page using such a script costs more than 500 ms while Chameleon only costs only 2 ms.

We choose to implement Chameleon by modifying the source code of a web browser. We have implemented Chameleon on Fennec, the mobile version of Mozilla Firefox, and the Android WebKit. WebKit is an open-source web engine which serves many modern web browsers such as Safari and Chrome. One can change the painting stage of WebKit to realize Chameleon. Because an Android system includes WebKit as part of the kernel, a modification of the WebKit codebase requires rebuilding the whole Android system. Fennec is an open source web browser project that is ported on both Android and Maemo operating systems. Unlike the Android Web browser, Fennec is a standalone application that can be installed on almost any mobile platforms with Android and Maemo systems without rebuilding the system.

Due to page limits, we only report the details of Fennec implementation because it does

not require rebuilding Android from source. The Android WebKit implementation, however, is similar. Our modification to the Fennec includes changing 15 files and adding/changing 387 lines of code. Chameleon/Fennec runs on any Android/Maemo based mobile platform. We next use Nexus One as an example target device to present the implementation details.

8.5.1 Automatic Power Model Construction

Battery interface of Nexus One updates a power reading in every 56 seconds, which is computed by averaging the last 14 samples (one sample per four seconds). Therefore, it is important to make sure Nexus One is operating with a with stable power behavior during the model building process. We enforce several methods to achieve a stable power behavior, including turning off all the wireless links by using airplane mode and shutting down all the running third party applications except Chameleon. More importantly, the calibration process itself, updating the frame buffer using different colors, should have a stable power behavior. Chameleon employs OpenGL to update the frame buffer in order to minimize energy consumption in non-display components, e.g. storage access when a gallery application is used to present images. Because the linearity of color power model, it is unnecessary to go through all the colors. We only calibrate sixteen levels for each of the red, green, and blue. Each level costs 56 seconds and the whole model building process takes approximately one hour. A user should start such model building process from the menu before using Chameleon.

We observe some peculiarities about the power consumption by Nexus Ones display. First, Nexus One has a well known issue in color representation: the data widths of red, green, and blue components are only five, six, and five, respectively in the OS [68]. As a result, the maximal sRGB values of red, green, and blue are 240, 248, and 240, respectively. This issue leads to a saturation effect in OLED power models, i.e., the power consumption

keeps constant when the sRGB values are between 240 and 255 for red and blue and between 248 and 255 for green. Second, the display of Nexus One has a poor performance in color additivity. That is, the luminance of a color showing on the display is lower than the sum of luminance of the three color components when there are more than one non-zero components. As a result, the power consumption of a color with more than one non-zero components is lower than the sum of the power consumption of the three components. We suspect the reason is that color representation in the display hardware uses an even shorter data width. Nevertheless, these two issues only require a small modification to the power model construction to consider the saturation effect for sRGB values close to 255 and to scale up sRGB values to estimate the power consumption of colors that have more than one non-zero components.

8.5.2 Color Contribution Collection

Contribution collection is implemented as a function and is called by Chameleon after a timeout because a paint function returns as described in Section 8.4.4. When contribution collection is called, Chameleon copies the updated screen region from the frame buffer to the main memory in a stream using the OpenGL function `GLReadPixels`. The time taken by this copying process depends on the region size and is at most 16 ms on Nexus One. As described in Section 8.4.4, the color contribution vector for a website, **D**, is implemented as a Hash table. Such Hash Table for each website is saved as a binary data file. When Chameleon opens a website, it loads the corresponding binary data file and updates it accordingly. A typical size of such a binary file is ~ 6 KB such that loading/saving the Hash table will not introduce much overhead. We note that color contribution collection does not need to be always on. As we will see in Section 8.6.2, two weeks of color contribution collection will lead to close to optimal power reduction for at least the next three months

for an average user.

8.5.3 Color Mapping Optimization

Out of the four algorithms described in Section 8.3.3, *Dark* and *Inversion* do not require the power model, \mathbf{M} , nor the color contribution vector, \mathbf{D} . Instead, the resulted color maps are only determined by the parameter λ . Chameleon utilizes these two algorithms with five levels of parameter λ to generate ten built-in color maps such that a Chameleon user is able to see the visual results of color transformation right after installing Chameleon.

For the other two algorithms with significantly more energy savings, *Arbitrary* and *Green*, optimized color maps must be generated for each website through color mapping optimization. Since mapping optimization runs offline and is independent from Chameleon, we choose to implement it as a service running in an Internet server. This service takes two inputs, i.e., a triplet of three float numbers representing the OLED display power model of a Chameleon users mobile device and a binary data file consisting of 20 color contribution vectors of the users top 20 most visited websites. The user can initiate the mapping optimization service from Chameleons menu. Chameleon will upload the two inputs to the Chameleon server and the service will calculate optimized color maps using all possible algorithms and options. For each website, the service generates color maps using two algorithms, i.e., *Green* and *Arbitrary*, five parameters settings for each algorithm. As a result, there are ten optimized color maps for each website. Then Chameleon automatically downloads this output data file to the mobile device. Finally, a color map will be selected for each website based on the users choice. When the user changes his/her choice, a new color map will be selected and applied immediately without needing the server.

The mapping optimization service is implemented in two parts: a front-end interface implemented by PHP to handle requests from Chameleon and a back-end computation

engine. The back-end engine is implemented in C++ and employs the GNU Scientific Library for optimization.

8.5.4 Color Transformation Execution

Chameleon implements color transformation of GUI objects by modifying the color interface of Fennec. When a paint function draws a geometry primitive, i.e., a point, a line or a rectangle, it uses a HEX string or a color name as a parameter to specify the color of the primitive. The color interface translates such a HEX string or a color name to RGB values that will be further used by the low level graphics library. We add code right after where the RGB values are calculated. The added code uses the RGB triplet as address to find a transformed color from the color map. Since we have modified the color interface to return the transformed RGB values instead of original ones, the paint function will draw the geometry primitive using the transformed color. Therefore, the overhead induced by color transformation execution only involves the time to load a RGB triplet from the color map. As will show in Section 8.6.3, the total time overhead of execution in opening a web page is less than 5 ms on average, which is negligible than the total time to open a web page (2-8 seconds according to [71]).

Chameleon implements color transformation for images by modifying the image library of each individual image format because each image library employs a unique decompression procedure. In particular, code is added to modify the RGB values of image pixels after the decompression finishes. The new RGB values are determined based on if the image should be darkened or color-transformed.

8.6 Evaluation

We evaluate through measurement, trace-based emulation, and multiple field trials. The evaluation shows that Chameleon is able to reduce OLED smartphone power consumption by 41% during web browsing.

8.6.1 Experimental Setup

Google Nexus One is used in the measurement and trace-based emulation. We run a script that automatically feeds Chameleon with web pages specified by a URL list extracted from the LiveLab traces. We measure time overhead of Chameleon by inserting time-stamps in the Chameleon source code and counting the latencies contributed by the contribution collection and execution modules. We obtain the power consumption of Nexus One by measuring the battery current through a sensing resistor with a 100 Hz sampling rate DAQ from Measurement Computing.

8.6.2 Power Reduction

As mentioned in the beginning of Section 8.4, Chameleon transforms future web pages based on color transformations optimized with past web usage. This raises two questions: (i) how long does a user need to train Chameleon; and (ii) how well does the past predicts the future? By “train,” we mean to run the contribution collection module of Chameleon to gather the color contribution vector, \mathbf{D} .

We leverage the LiveLab traces, the display power model of Nexus One, the Arbitrary color transformation used in the study reported in Section 8.3.3, to answer these two questions. We first train and test Chameleon week by week, using the same weeks trace to train and test. The resulting power reduction is not realistic but serves as a theoretical upper bound. The weekly average for all 25 users is shown in Figure 8.11 as Optimal. Then

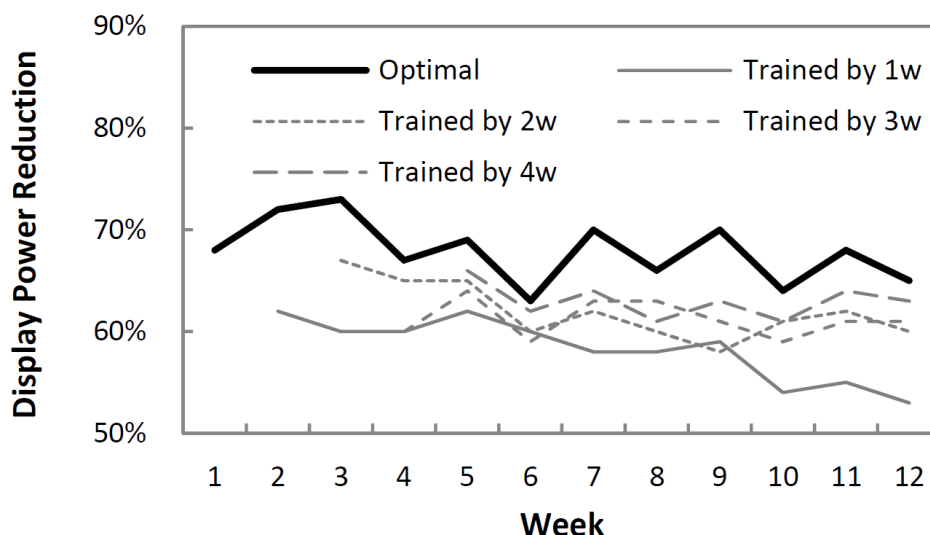


Figure 8.11 : Display power reduction by training Chameleon with different weeks of data from the LiveLab traces.

we train Chameleon using first one to four weeks of traces and test it using the rest of the traces, respectively. Again the weekly averages for all users are shown in Figure 8.11. Because the color transformation in Chameleon is per-website, the weekly average is calculated over the top 20 websites for all users. We note this emulation is an approximation only because the LiveLab traces did not actually capture what appeared on the display, as discussed in Section 8.4.4.

As shown in Figure 8.11, power reduction with two to four weeks training is close to each other but that with one week training is the obviously lowest. This indicates that two weeks of training should be enough for Chameleon on average. The average display power reduction is 64% and such reduction remains close to the optimal ($\sim 70\%$) for the rest of traces (or at least 10 weeks), as shown in Figure 11. So the past does not predict the future well. We also note that there is a not-so-obvious trend of declination in the power reduction

with two to four weeks training only over the long term. This suggests that Chameleon only needs to train (or run the contribution collection module) for two weeks a few times a year to maintain its effectiveness in power reduction. Meanwhile, Chameleon only needs to communicate with the server to calculate optimized color maps a few times a year when the device is wall-powered and idle. As a result, it is safe for us to ignore the overhead of offloading.

We also measured the system power consumption by Nexus One when browsing web pages in the LiveLab traces. The average system power consumption is 1.3 W and 2.2 W with Fennec and Chameleon, respectively. This indicates over 40% total system power reduction through Chameleon.

8.6.3 Overhead

We next examine the overhead in time and power introduced by Chameleon. For time overhead, we are only interested in contribution collection and execution because only they run in real-time during web browsing. Our measurement shows that execution takes less than 5 ms on average in opening a web page, which is negligible compared to the total time to open a web page in a mobile browser (2-8 seconds according to [71]). The time overhead from contribution collection consists of two parts: reading the frame buffer and updating the color contribution vector, **D**. Figure 8.12 shows the worst case overhead of both parts using different sampling sizes. As shown in the figure, frame buffer reading costs less than 16 ms in the worst case, i.e., reading the whole screen. The average reading time is ~ 12 ms because only part of the frame buffer is read each time. The figure also shows that cost for updating the color contribution vector is ~ 50 ms without sampling and < 1 ms using a sampling window of 10×10 . Therefore, the total time overhead of color contribution vector updating is at most 66 ms without sampling and < 17 ms using a sampling window

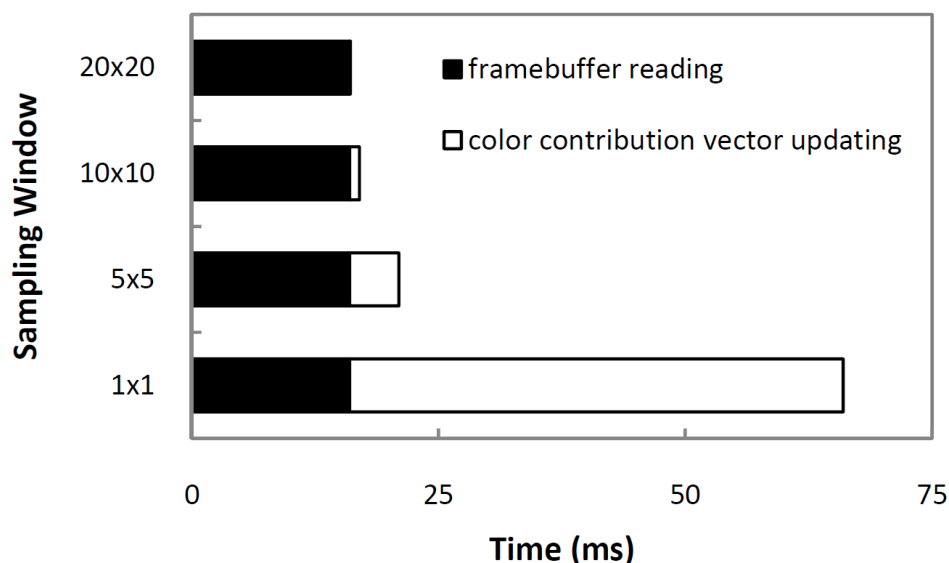


Figure 8.12 : Worst case time overhead of Chameleon.

of 10×10 .

Such overhead will be barely noticeable given the fact that it takes two to eight seconds to open a web page on a mobile browser [71], as confirmed by our own experience and the field trial to be reported in Section 8.6.4. Moreover, the overhead of Chameleon is likely to be reduced as smartphone hardware becomes better while the web page opening time will remain large because it is determined by network condition other than smartphone hardware [71]. Finally, as shown in Section 8.6.2, Chameleon only needs to run contribution collection and incur this overhead for two weeks a few times a year to be effective.

We also measured the power overhead of Chameleon by going through the LiveLab URL list twice, one with Fennec and the other with Chameleon. To eliminate the power difference introduced by the display, a color map that does not change the color at all is used in Chameleon. The results show that the system power difference between the two

trials is $< 5\%$, negligible compared to the 40% reduction from Chameleon.

8.6.4 Field Trials

Three field trials have been performed with Chameleon/Fennec during the development. The first trial was with two Nexus One users and an early version of Chameleon. The two participants used Chameleon for one week and provided valuable feedbacks. For example, the issues with background images and logo images discussed in Section 8.3.2 were reported by the two participants whose different preferences also motivated us to give the end user an option in how to treat logo images.

The second trial was with five users of Nexus One and Samsung Galaxy S and the reported implementation of Chameleon. The trial lasted seven days for each participant. After the trial, we asked each participant to use two Nexus One smartphones, one with the original Fennec and the other with Chameleon. They used the smartphones in the lab at a random order to access the same website of their favorite in order to assess if there was noticeable latency introduced by Chameleon. No participant noticed any slowdown in web browsing in Chameleon compared with using the original Fennec. From our post-trial interview of the participants, we found that the ability to see the effect of changing a user option is very important, as Chameleon currently supports. Finally, all the participants are satisfied with the aesthetics of the transformed color scheme of his/her choice.

These early results suggest Chameleon is effective in reducing power reduction while keeping the user happy.

The third trial was with 36 participants from the United States and China. The participants include 16 females and 20 males. Their ages range from 15 to 42 with a mean of 26.5 and a standard deviation of 6.6. They have very diverse academic background from arts to engineering. Every user owns an Android smartphone with an OLED display and there

are eight smartphone models in total. The participants started the trial at different time but all the users have at least one month usage. They started by downloaded and installed a special version of Chameleon with two weeks of training and with code instrumentation to understand the usage. Whenever the participant opens/closes a web page or switches color maps from the menu, the instrumented Chameleon records the following data: the timestamp, the URL of the web page, the color map choice, and the battery status. By analyzing the data collected, we make the following observations.

First, our participants accepted the transformed web pages well, especially when battery level was low. Figure 8.13 shows the average percentage of usage time of color maps generated by each algorithm by each participant. To reveal the impact of battery status on users choice of color maps, the figure presents the percentage for high ($> 50\%$) and low ($< 50\%$) battery levels separately. As shown in the figure, original (no color transformation) was used for 52% and 37% of the time when battery level was high and low, respectively. That is, the participants spent roughly half of the web browsing time with color-transformed web pages when battery level was high and the percentage increased to above 60% when battery level went low. Moreover, the fact that the color maps generated by each of the four algorithms take a considerable percentage indicates that the transformation options provided by Chameleon are valuable to end users. This confirms the observation made in Section 8.3.3.

Second, our participants changed color maps more often at the beginning. Figure 8.14 shows the average number of color map switches per website by a participant in each week. We distinguish switches between two color maps generated by different algorithms and switches between two color maps generated by the same algorithm using different parameters. As shown in the figure, both types of switches achieved their maximums in the first week due to users initial excitement and decreased from the second week. Both

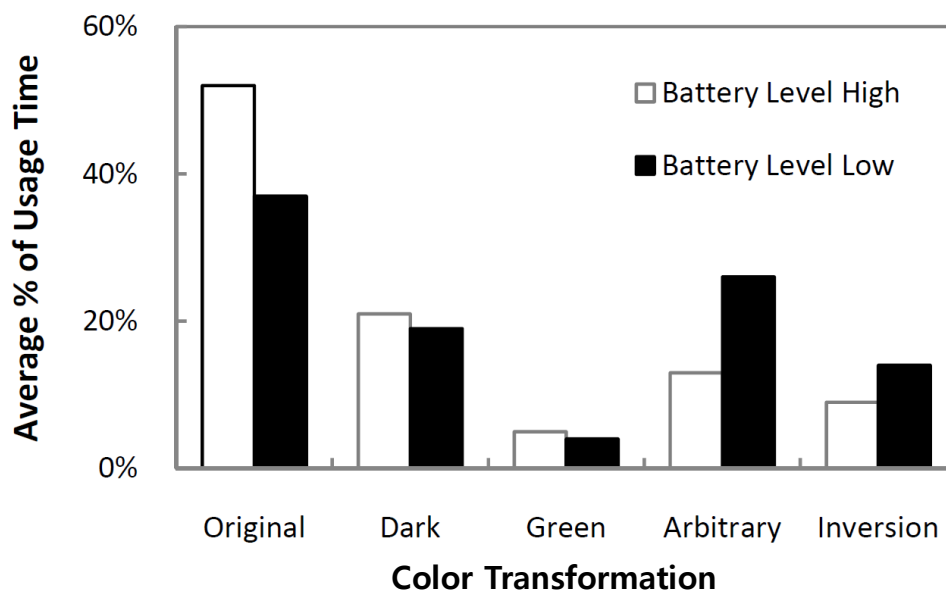


Figure 8.13 : Average percentage of usage time of color maps generated by each algorithm by a participant.

increased in the third week when the participants got optimized color maps after two-week training. Towards the end of the trial, the number of color map switches between algorithms became almost zero, which indicates that the participants had already adopted a favorite color transformation for each website. Meanwhile, on the contrary, the color map switches between parameters remained a certain level. We conjecture the participants might still need to adjust color maps slightly to accommodate changes in viewing condition such as ambient lighting. This indicates the usefulness of color maps generated with different parameters by the same algorithm for fine tuning. We also notice that there always existed color map switches between algorithms even at the end of the trial. Such cases mainly took place when different sections of the same website had very different color distribution, e.g., different sports sections of ESPN. Fortunately, Chameleon can easily address this issue by

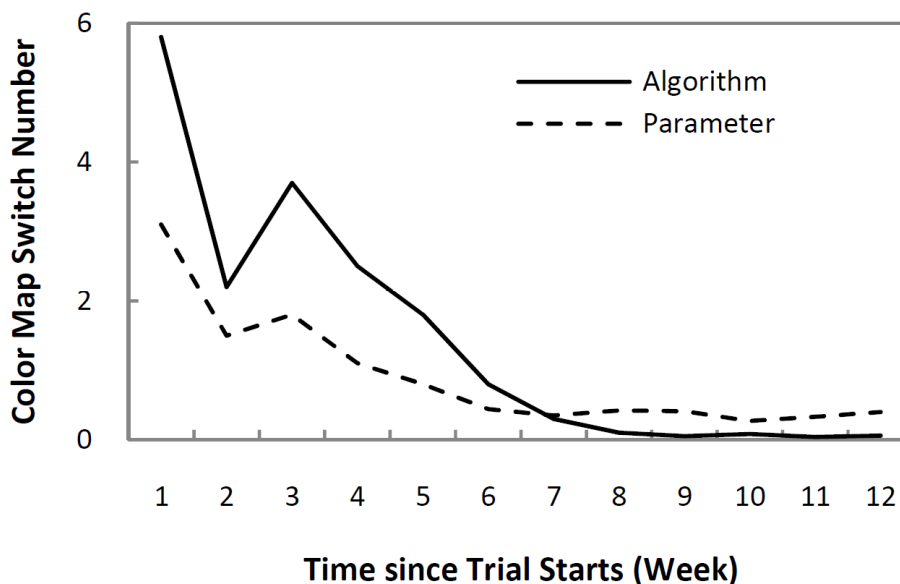


Figure 8.14 : Evolution of average number of color map switches per website by each participant along time.

considering these sections as different websites.

8.7 Discussion

Chameleon does not consider video right now because videos are not yet widely supported by mobile browsers. For example, Safari on iPhone has no support of flash, a typical video container in web pages. However, supporting video is also easy for Chameleon as videos can be considered in a way similar to foreground images. Chameleon can easily apply darkening to videos by leveraging the *opacity level* defined in CSS standard. Opacity level is a property of each object and in the range of $[0,1]$; it is used to handle the overlapping of two objects. To darken a video by a scale factor $\lambda \in [0,1]$, Chameleon can overlay a black image with the opacity level of $(1 - \lambda)$ and of the same size on the top of the video.

Chameleon can be enhanced to accommodate aesthetic factors by adding aesthetic constraints in the algorithms to calculate power efficient color maps. For example, the relative aesthetic impact of a color, or *color energy*, is mainly determined by saturation [75]. It is visually appealing to use more saturated color for foreground objects and less saturated color for background. Chameleon can easily take consider this factor by mapping the background color of a web page, usually the color corresponding to the largest component in the color distribution vector, \mathbf{D} , only to colors of low saturation.

8.8 Summary

In this chapter, we report the design and realization of Chameleon, a color-adaptive mobile web browser to reduce the energy consumption by OLED mobile systems. Chameleon is able to reduce the system power consumption of OLED smartphones by over 40% for web browsing, without introducing any user noticeable delay. Our field trial results show that the transformed web pages provided by Chameleon are well accepted by users.

Emulation using web usage traces collected from real smartphone users showed that two weeks of web usage is enough for Chameleon to derive a color transformation that performs close to the optimal for at least ten weeks. This suggests that only the execution module of Chameleon needs to be performed all the time and the color contribution collection module only need to be performed for two weeks once or twice a year.

We found studying the users and the web usage by users in the field instrumental to the design and success of Chameleon. Much of Chameleons optimization techniques were directly motivated by findings from these studies.

Chameleon also represents a good example of how the “cloud” can be leveraged for the usability and efficiency of mobile devices. By cleverly offloading compute-intensive mapping optimization to the cloud and obtaining all possible color maps, Chameleon allows

the user to see the visual impact of different user options without delay.

Chapter 9

Related Work

This chapter lists related works. We group related work into three categories: energy accounting, energy modeling, and energy management.

9.1 Energy Accounting Policies

Energy accounting is to tell how much energy a software entity, i.e., application, contributes to the system energy consumption. Denote there are n applications, i.e., $1, 2, \dots, i, \dots, n$ concurrently running in a mobile system. During a time interval T , the total energy consumption of the system is E . The energy contribution by application i can be represented by $\phi_i, i = 1, 2, \dots, n$.

WattsOn [2] utilized energy accounting to obtain the energy consumption of each mobile app to enable third-party app developers to improve energy efficiency of their products. Its accounting policy was *Policy II*, as described in Chapter 3, which states energy contributed by process i is the same as its stand-alone energy consumption $E(\{i\})$ when only process i is running in the system. [9] utilized energy accounting to get energy contribution in graphics user interfaces. It used accounting *Policy III* which states that the energy contributed by process i is equal to its marginal energy contribution, which is the difference between system energy consumption when process i is running and when it is not running, while all other conditions remain unchanged. PowerScope [10] used *Policy IV* to account energy consumption of each software running in a system, in which the energy contributed

by process i is equal to the sum of system energy consumption over all scheduling intervals when process i is scheduled in CPU.

PowerTutor [5, 12], ECOSystem [3], and Joulemeter [13] utilize *Policy V* for energy accounting for smartphones, laptop, and servers, respectively. This sophisticated policy relies on a system energy model $E = f(x_1, x_2, \dots, x_p)$, where $x_k, k = 1, 2, \dots, p$, are predictors that can be obtained in software such as CPU and memory usage. The system figures out how much process i contributes to each of the predictors, $x_{k,i}$ and determines the energy contribution by i by plugging $x_{k,i}$ back into f , i.e., $\phi_i = f(x_{1,i}, x_{2,i}, \dots, x_{k,i}, \dots, x_{p,i})$. To have the Efficiency property described in Chapter 2, however, f must be a linear function, i.e., $E = \beta_0 + \sum_{k=1}^p \beta_k \cdot x_k$ and there must be a heuristic to split the constant β_0 among the processes.

This policy fails because how it determines the energy contribution by a process severely limits how it estimates the system energy consumption. In particular, the estimation can only use resource usage that can be attributed to individual processes; and the estimation must employ a linear model. These two limitations are fatal because a mobile system consists of many hardware components including some major energy consumers that are invisible to the OS [14], e.g., graphics, and others whose usage cannot be easily attributed to a process, e.g., display and GPS. Moreover, it has been recently argued that linear models are inadequate for estimating system energy consumptions [15, 16]. As a result, existing solutions will not only significantly underestimate the energy contribution by a process and but also be unable to attribute a significant portion of system energy consumption to running processes.

9.2 Energy Modeling Methods

In Chapter 6, we described the system solution for high-accuracy *in situ* energy measurement for short intervals. Instead of direct measurement, one can also estimate $E(\mathbb{S})$ with a system energy model without requiring new hardware. There is a large body of work on energy modeling on smartphones [5, 23]. Zhang et al [5] created a linear energy model for Android-based smartphones based on CPU, LCD, WiFi and 3G usage. Pathak et al [23] utilized system call tracing to address the tail power consumption issue of WiFi and 3G devices. These energy models can provide a software-only solution for energy accounting and OEM. On the other hand, they have lower accuracy than that can be achieved with our solution reported in Chapter 6. Even if improved battery interface is unavailable, our solution described in Chapter 5 still outperforms these models in both accuracy and speed.

The low accuracy of these models is due to a major limitation that all of them are generated by a fixed set of benchmark applications. As shown by [32, 33], smartphone usage is diverse among different users. Different usage will lead to different energy models (See Section 4.2.3). Therefore, a model generated by a fixed set of benchmark applications can be less accurate when a user runs a new application. On the contrary, our solution reported in Chapter 5 utilizes real usage of every individual user to generate personalized energy models and to adapt such models for potentially higher accuracy.

In [23], the authors view a system as a finite-state-machine (FSM) and associate each state with a fixed power number. They trace system calls to determine which state the system is in and thus obtain the corresponding power number. This FSM model achieves high accuracy at a high rate of 20 Hz (80% of the estimation has an error less than 10%). However, this approach requires external hardware to measure power as well as a sophisticated calibration procedure that requires device specific hardware information. In contrast, our solution described in Chapter 5 does not require any hardware information and our solution

described in Chapter 6 does not require any calibration procedure.

There are two recent papers following our OLED power modeling work discussed in Chapter 7. WattsOn [2] presents a mobile application development tool that includes a system energy modeling module. In this module, OLED display energy consumption is estimated using a similar model as described in Chapter 7, but their new model includes a special step to compensate power non-additivity in certain type of OLED displays. [76] studies the energy consumption characteristics of OLED displays used by a series of commercial smartphones, e.g., Samsung Galaxy S, Samsung Galaxy S II, and Samsung Galaxy S III, and proposes a unified OLED energy model. Such unified model is based on energy efficiency of OLED devices to eliminate the impact of spatial resolution and physical size of the OLED display. Both pieces of work complement our OLED modeling work described in Chapter 7 and can be readily integrated in our color transformation framework presented in Chapter 7 and the color-adaptive browser presented in Chapter 8.

9.3 Energy Management Approaches

Operating-system-based energy management has widely utilized energy accounting of various sophistication. It seeks to control the energy use by each process and schedule their executions to maximize aggregate system utility under energy constraints. There is an extensive body of literature on system energy management. We next discuss two major pieces of work that utilize BEM, as described in Chapter 3.

ECOSystem [3] presents "currency", an abstraction of energy to unify a systems device power states. ECOSystem incorporates "currency" for explicit energy management from system's perspective. Energy consumption by each hardware component is modeled as charge in "currency" that attributed to each running process. This is exactly how BEM works as described in Chapter 3. In its implementation, ECOSystem uses a resource

container to realize "currency" and groups logical related processes to share the same container. This approach is limited because it must either share its container with a child or put it in a new container that competes for resources. Therefore, it is impossible for a process to delegate.

Like ECOSystem, Cinder [4] also utilizes BEM, but advances ECOSystem by providing more sophisticated mechanisms for processes to delegate. Cinder simplifies construction of these policies using its fine-grained protection mechanism and reserves to provide the same result from user-space's perspective.

Both energy management works employ energy accounting Policy V that is based on a software-based energy model. Therefore, the management results are limited due to the fallbacks of Policy V as discussed in Chapter 2 and experimentally demonstrated in Chapter 6.

Chapter 10

Conclusion

This final chapter summarizes the research contributions of this thesis and explores related avenues for future work.

10.1 Contributions

The contributions of this thesis fall mainly into three areas: the theoretical framework of system energy accounting and management developed in Chapter 2-3, the practical solution of *In Situ E*(\mathbb{S}) estimation presented in Chapter 4-6, and the comprehensive treatment of power modeling and optimization for OLED displays presented in Chapter 7-8.

10.1.1 Theoretical Framework

The theoretical framework answered two important and long-standing questions about energy accounting. That is, how to evaluate an energy accounting policy? and does energy accounting matters for operating system-based energy management?

The answer to the first question is: Shapley value based energy accounting should serve as the ground truth for energy accounting. Chapter 2 analytically showed how existing energy accounting policies violated the four self-evident axioms that define the Shapley value. This result is fundamental to evaluating energy accounting policies that have been widely used in software evaluation and energy management.

The answer to the second question is: for existing budget based energy management

(BEM), energy accounting does not matter at all as long as it satisfies the Efficiency property, as discussed in Chapter 3. But for a novel optimal energy management (OEM), as much as information as needed by Shapley value based accounting is needed, i.e., $E(\mathbb{S}) \forall \mathbb{S} \subseteq \mathbb{N}$.

10.1.2 *In Situ* $E(\mathbb{S})$ Estimation

Chapter 4 discussed the system challenges to realize Shapley value based energy accounting and OEM based scheduling in real mobile systems, and identified the key information needed is $E(\mathbb{S})$. Chapter 5 and Chapter 6 then showed that such information can be practically gathered *in situ* with two novel system designs.

Chapter 5 presented self energy modeling of mobile systems, or Sesame, with which a mobile system constructs an energy model of itself with its existing battery interface. Such energy model can be used for *In Situ* $E(\mathbb{S})$ estimation in Shapley value based energy accounting and OEM based scheduling. Chapter 5 also reported a Linux-based prototype of Sesame and showed that Sesame was able to achieve 88% and 82% accuracy for a laptop and a smartphone, respectively, at 100 Hz, which was at least five times faster than existing system energy models.

Chapter 6 demonstrated the feasibility of *In Situ* $E(\mathbb{S})$ estimation using an improved battery interface and reported a prototype implementation of both Shapley value-based energy accounting and OEM based scheduling. Using such prototype, Chapter 6 experimentally demonstrated how erroneous existing energy accounting policies can be, showed that existing BEM solutions are unnecessarily complicated yet underperforming by 20% compared to OEM.

10.1.3 OLED Display Power Optimization

Chapter 7 provided models for efficient and accurate power estimation for OLED displays, at pixel, image, and code levels, respectively. The pixel-level model built from measurements achieved 99% accuracy in power estimation for 300 benchmark images. By aggregating the power of a small group of pixels instead of all the pixels in an image, the image-level power model reduced the computation cost by 1600 times, while achieving 90% accuracy in power estimation. The code-level model utilized specification of the GUI objects to calculate the power consumption, which guarantees 95% accuracy. Then, based on the models, Chapter 7 proposed techniques that adapt GUIs based on existing mechanisms as well as arbitrarily under usability constraints. Measurement and user studies showed that more than 75% display power reduction can be achieved with user acceptance.

Chapter 8 reported the design and realization of Chameleon, a color-adaptive mobile web browser to reduce the energy consumption by OLED mobile systems. Chameleon was able to reduce the system power consumption of OLED smartphones by over 40% for web browsing, without introducing any user noticeable delay. The field trial results showed that the transformed web pages provided by Chameleon were well accepted by users.

10.2 Suggestions for Future Work

The key to both Shapley value-based energy accounting and the proposed optimal energy management is $E(\mathbb{S}) \forall \mathbb{S} \subseteq \mathbb{N}$ where \mathbb{N} is the set of processes under question. While this thesis provided a suite of techniques to acquire $E(\mathbb{S})$ in Chapter 6, the solutions can be further augmented in three important ways.

What if one can track the use of a hardware unit by a process and know the unit's energy consumption via measurement or modeling? The reported implementation in Chapter 6

does not distinguish any hardware unit from the whole system. That is, it only cares if a process is active during the time under question without caring which hardware unit it actually uses; and it only cares the energy consumption of the entire system without needing that by any of its hardware units. In many systems, it is easy to track the use of some hardware units such as CPU, main memory, and even network interfaces; and the energy consumption by these units can be derived from model or measurement techniques like iCount [24]. In this case, these hardware units' energy consumption can be exactly attributed to processes that use them. Shapley value based energy accounting will still be necessary to properly attribute the energy consumption of the rest of the system. On the other hand, tracking the use of these hardware units and revoking their energy models incur runtime overhead. It is not clear to the authors if separating them out of the Shapley value framework is beneficial. More experimentation is necessary to settle this question.

Using the crowd: Chapter 6 presented a few heuristics to obtain $E(\mathbb{S})$ for $\mathbb{S} \subseteq \mathbb{N}$ that are not observed in the system under question. As a model of smartphone is usually used by over hundreds of thousands of users, the users of the same model can help each other by contributing their own observations of $E(\mathbb{S})$. Each observation includes the energy consumption, unique identities of the running processes (\mathbb{S}), device model, hardware state (σ). The observations made by one device can be compacted and uploaded to a central server occasionally, e.g., when connected via WiFi and wall-powered. Similarly, a device can look up the same server for observations for the same model that are locally unavailable.

Considering variance in $E(\mathbb{S})$: There are many factors other than hardware state, σ , that will contribute to the variance of $E(\mathbb{S})$. For example, the transmission power of the 3G/4G radio interface highly depends on the cellular signal strength. Such factors are not considered in our current implementation. However, the Shapley value energy accounting framework can readily include such factors by treating $E(\mathbb{S})$ as a random variable and

leverage all the theory work on random Shapley value that require the variance or the range [77] of $E(\mathcal{S})$. The only thing that requires modification in our system is Shapley value calculation, while the mean and variance of $E(\mathcal{S})$ can be obtained from the same data collection framework.

This work can also be extended to further attribute the energy use by the OS to processes. The current implementation treats the OS as a separate entity and allocates its own share of energy use. The OS, however, can perform a service on behalf of a process, e.g., sending a packet out through TCP/IP. Therefore, it is reasonable to attribute the energy use by such services to the process being served. The implementation can be easily modified for this purpose because a modern OS like Linux keeps track of the process that it is serving. The implementation will be able to calculate the energy use by the OS when serving the process; the only modification is to attribute this energy use to the process, instead of the OS itself.

Bibliography

- [1] A. Pathak, Y. Hu, and M. Zhang, “Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof,” in *Proc. ACM EuroSys*, 2012.
- [2] R. Mittal, A. Kansal, and R. Chandra, “Empowering developers to estimate app energy consumption,” pp. 317–328, 2012.
- [3] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat, “ECOSystem: Managing energy as a first class operating system resource,” in *ACM SIGPLAN Notices*, vol. 37, 2002.
- [4] A. Roy, S. Rumble, R. Stutsman, P. Levis, D. Mazieres, and N. Zeldovich, “Energy management in mobile devices with the Cinder operating system,” in *Proc. ACM EuroSys*, 2011.
- [5] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang, “Accurate on-line power estimation and automatic battery behavior based power model generation for smartphones,” in *Proc. IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and System Synthesis*, 2010.
- [6] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, “Appscope: Application energy metering framework for android smartphone using kernel activity monitoring,” in *USENIX Annual Technical Conf.*, 2012.
- [7] L. Shapley, “A value for n-person games,” in *Contributions to the Theory of Games, volume II* (H. Kuhn and A. Tucker, eds.), pp. 307–317, Princeton University Press,

1953.

- [8] H. Lim, A. Kansal, and J. Liu, "Power budgeting for virtualized data centers," in *2011 USENIX Annual Technical Conf.*, 2011.
- [9] L. Zhong and N. Jha, "Graphical user interface energy characterization for handheld computers," in *Proc. ACM CASES*, 2003.
- [10] J. Flinn and M. Satyanarayanan, "PowerScope: A tool for profiling the energy usage of mobile applications," in *Proc. IEEE WMCSA*, 1999.
- [11] <http://source.android.com/>.
- [12] <http://powertutor.org/>.
- [13] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. Bhattacharya, "Virtual machine power metering and provisioning," in *Proc. ACM Symp. Cloud Computing*, 2010.
- [14] R. Muralidhar, H. Seshadri, K. Paul, and S. Karumuri, "Linux-based ultra mobile PCs," in *Proc. Linux Symposium*, 2007.
- [15] J. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. Snoeren, and R. Gupta, "Evaluating the effectiveness of model-based power characterization," in *USENIX Annual Technical Conf.*, 2011.
- [16] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," June 2011.
- [17] V. Misra, S. Ioannidis, A. Chaintreau, and L. Massoulié, "Incentivizing peer-assisted services: a fluid Shapley value approach," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, 2010.

- [18] J. Feigenbaum, C. Papadimitriou, and S. Shenker, “Sharing the cost of multicast transmissions,” *Journal of Computer and System Sciences*, vol. 63, no. 1, 2001.
- [19] S. Ryffel, “LEA²P: the linux energy attribution and accounting platform,” Master’s thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 2009.
- [20] <http://www.dianxinos.com/>.
- [21] T. Lan, D. Kao, M. Chiang, and A. Subharwal, “An axiomatic theory of fairness for resource allocation,” 2010.
- [22] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
- [23] A. Pathak, Y. Hu, M. Zhang, P. Bahl, and Y. Wang, “Fine-grained power modeling for smartphones using system call tracing,” in *Proc. ACM EuroSys*, pp. 153–168, 2011.
- [24] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler, “Energy metering for free: Augmenting switching regulators for real-time monitoring,” in *Proc. IEEE. IPSN*, 2008.
- [25] X. Fan, W. Weber, and L. Barroso, “Power provisioning for a warehouse-sized computer,” *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 13–23, 2007.
- [26] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner, “Event-driven energy accounting for dynamic thermal management,” in *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP03)*, vol. 22, 2003.
- [27] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, “Full-system power analysis and modeling for server environments,” in *In Proceedings of Workshop on Modeling, Benchmarking, and Simulation*, pp. 70–77, 2006.

- [28] A. Shye, B. Scholbrock, and G. Memik, “Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 168–178, ACM, 2009.
- [29] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: a framework for architectural-level power analysis and optimizations,” in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 83–94, ACM, 2000.
- [30] G. Contreras and M. Martonosi, “Power prediction for intel xscale® processors using performance monitoring unit events,” in *Low Power Electronics and Design, 2005. ISLPED’05. Proceedings of the 2005 International Symposium on*, pp. 221–226, IEEE, 2005.
- [31] F. Rawson and I. Austin, “Mempower: A simple memory power analysis tool set,” *IBM Austin Research Laboratory*, 2004.
- [32] H. Falaki, R. Mahajan, S. Kandula, D. LyMBERopoulos, R. Govindan, and D. Estrin, “Diversity in smartphone usage,” ACM, 2010.
- [33] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. Rice, “Exhausting battery statistics: understanding the energy demands on mobile handsets,” in *Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds*, pp. 9–14, ACM, 2010.
- [34] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, “Quanto: Tracking energy in networked embedded systems,” in *8th USENIX Symposium of Operating Systems Design and Implementation (OSDI08)*, pp. 323–328, 2008.

- [35] P. Ranganathan and P. Leech, "Simulating complex enterprise workloads using utilization traces," in *10th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2007.
- [36] <http://http://www.lesswatts.org/projects/bltk/>.
- [37] A. Rahmati and L. Zhong, "A longitudinal study of non-voice mobile phone usage by teens from an underserved urban community," *arXiv preprint arXiv:1012.2832*, 2010.
- [38] <http://www.jbenchmark.com/>.
- [39] <http://www.futuremark.com/products/3dmarkmobile/>.
- [40] www.ti.com/lit/ds/symlink/bq2019.pdf.
- [41] S. Van Huffel and P. Lemmerling, "Total least squares and errors-in-variables modeling: Analysis, algorithms and applications," *status: published*, 2002.
- [42] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: measuring wireless networks and smartphone users in the field," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 15–20, 2011.
- [43] A. Perez, E. Bonizzoni, and F. Maloberti, "A 84dB SNDR 100kHz bandwidth low-power single op-amp third-order $\Delta\Sigma$ modulator consuming 140 μ W," in *Proc. IEEE ISSCC*, 2011.
- [44] F. Gatti, A. Acquaviva, L. Benini, and B. Ricco, "Low power control techniques for tft lcd displays," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 218–224, ACM, 2002.

- [45] W. Cheng, Y. Hou, and M. Pedram, "Power minimization in a backlit tft-lcd display by concurrent brightness and contrast scaling," in *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, p. 10252, IEEE Computer Society, 2004.
- [46] L. Zhong and N. Jha, "Energy efficiency of handheld computer interfaces: limits, characterization and practice," in *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pp. 247–260, ACM, 2005.
- [47] L. Cheng, S. Mohapatra, M. El Zarki, N. Dutt, and N. Venkatasubramanian, "Quality-based backlight optimization for video playback on handheld devices," *Advances in Multimedia*, vol. 2007, no. 1, pp. 4–4, 2007.
- [48] S. Forrest, "The road to high efficiency organic light emitting devices," *Organic Electronics*, vol. 4, no. 2, pp. 45–48, 2003.
- [49] S. Iyer, L. Luo, R. Mayo, and P. Ranganathan, "Energy-adaptive display system designs for future mobile environments," in *Proceedings of the 1st international conference on Mobile systems, applications and services*, pp. 245–258, ACM, 2003.
- [50] M. Raghunath and C. Narayanaswami, "User interfaces for applications on a wrist watch," *Personal and Ubiquitous Computing*, vol. 6, no. 1, pp. 17–30, 2002.
- [51] P. Ranganathan, E. Geelhoed, M. Manahan, and K. Nicholas, "Energy-aware user interfaces and energy-adaptive displays," *Computer*, vol. 39, no. 3, pp. 31–38, 2006.
- [52] H. Shim, N. Chang, and M. Pedram, "A backlight power management framework for battery-operated multimedia systems," *Design & Test of Computers, IEEE*, vol. 21, no. 5, pp. 388–396, 2004.

- [53] A. Bhowmik and R. Brennan, "System-level display power reduction technologies for portable computing and communications devices," in *Portable Information Devices, 2007. PORTABLE07. IEEE International Conference on*, pp. 1–5, IEEE, 2007.
- [54] W. Cheng, C. Hsu, and C. Chao, "Temporal vision-guided energy minimization for portable displays," in *Proceedings of the 2006 international symposium on Low power electronics and design*, pp. 89–94, ACM, 2006.
- [55] C. Poynton, *Digital Video and HD: Algorithms and Interfaces*. Morgan Kaufmann, 2012.
- [56] J. Shinar, *Organic light-emitting devices: a survey*. Springer, 2003.
- [57] F. Bunting, C. Murphy, and B. Fraser, "Real world color management," 2005.
- [58] T. Harter, S. Vroegindeweij, E. Geelhoed, M. Manahan, and P. Ranganathan, "Energy-aware user interfaces: an evaluation of user acceptance," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 199–206, ACM, 2004.
- [59] N. Kamijoh, T. Inoue, C. Olsen, M. Raghunath, and C. Narayanaswami, "Energy trade-offs in the ibm wristwatch computer," in *Wearable Computers, 2001. Proceedings. Fifth International Symposium on*, pp. 133–140, IEEE, 2001.
- [60] J. Chuang, D. Weiskopf, and T. Möller, "Energy aware color sets," in *Computer Graphics Forum*, vol. 28, pp. 203–211, Wiley Online Library, 2009.
- [61] W. Cheng and C. Chao, "Perception-guided power minimization for color sequential displays," in *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp. 290–295, ACM, 2006.

- [62] A. Iranli and M. Pedram, “Dtm: dynamic tone mapping for backlight scaling,” in *Proceedings of the 42nd annual Design Automation Conference*, pp. 612–617, ACM, 2005.
- [63] A. Iranli, H. Fatemi, and M. Pedram, “Hebs: Histogram equalization for backlight scaling,” in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 346–351, IEEE, 2005.
- [64] K. Vallerio, L. Zhong, and N. Jha, “Energy-efficient graphical user interface design,” *Mobile Computing, IEEE Transactions on*, vol. 5, no. 7, pp. 846–859, 2006.
- [65] <http://www.4dsystems.com.au/>.
- [66] <http://www.wm6themes.com/>.
- [67] A. Carroll and G. Heiser, “An analysis of power consumption in a smartphone,” in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pp. 21–21, USENIX Association, 2010.
- [68] <http://www.displaymate.com/mobile.html/>.
- [69] http://www.pcworld.com/article/204729/samsung_launches_galaxy_tab.html/.
- [70] A. Lääperi, “Disruptive factors in the oled business ecosystem,” *Information Display*, vol. 9, no. 09, p. 09, 2009.
- [71] Z. Wang, F. Lin, L. Zhong, and M. Chishtie, “Why are web browsers slow on smartphones?,” in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pp. 91–96, ACM, 2011.

- [72] M. Jones, G. Marsden, N. Mohd-Nasir, K. Boone, and G. Buchanan, “Improving web interaction on small displays,” *Computer Networks*, vol. 31, no. 11, pp. 1129–1137, 1999.
- [73] N. Bila, T. Ronda, I. Mohomed, K. Truong, and E. de Lara, “Pagetailor: reusable end-user customization for the mobile web,” in *Proceedings of the 5th international conference on Mobile systems, applications and services*, pp. 16–29, ACM, 2007.
- [74] G. Wyszecki, “Color science: concepts and methods, quantitative data and formulae, (paper),” 2001.
- [75] H. Zettl, *Sight, sound, motion: Applied media aesthetics*. Wadsworth publishing company, 2010.
- [76] X. Chen, Y. Chen, Z. Ma, and F. C. Fernandes, “How is energy consumed in smart-phone display applications?,” in *Proceedings of the 14th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, ACM, 2013.
- [77] S. Alparslan Gök, R. Brânzei, and S. Tijs, “The interval shapley value: an axiomatization,” *Central European Journal of Operations Research*, vol. 18, no. 2, 2010.