Smartphone Energy Consumption

With an ever-increasing number of applications available for mobile devices, battery life is becoming a critical factor in user satisfaction. This practical guide provides you with the key measurement, modeling, and analytical tools needed to optimize battery power by developing energy-aware and energy-efficient systems and applications.

As well as the necessary theoretical background and results from the field, this hands-on book also provides real-world examples, practical guidance on assessing and optimizing energy consumption, and details of prototypes and possible future trends. Uniquely, you will learn about energy optimization of both hardware and software in one book, enabling you to get the most from the available battery power.

Covering experimental system design and implementation, the book supports assignment-based courses with a laboratory component, making it an ideal textbook for graduate students. It is also a perfect guidebook for software engineers and systems architects working in industry.

Sasu Tarkoma is a Full Professor in the Department of Computer Science at the University of Helsinki, Finland. He has worked in the IT industry as a consultant and chief system architect as well as principal researcher and laboratory expert at Nokia Research Center. His interests include mobile computing, Internet technologies, and middleware.

Matti Siekkinen is a Teaching Research Scientist at Aalto University, Finland. He has co-authored over 40 scientific publications and his research interests include the efficiency of mobile devices and network measurements and protocols.

Eemil Lagerspetz is a doctural student in the Department of Computer Science at the University of Helsinki. His research interests include mobile energy awareness, data analysis, and cloud computing. He has published many scientific articles on mobile energy efficiency.

Yu Xiao is a Postdoctoral Researcher in the Department of Computer Science and Engineering at Aalto University. Her research interests include energy-efficient wireless networking, mobile cloud computing, and mobile crowd-sensing.

Smartphone Energy Consumption

Modeling and Optimization

SASU TARKOMA
University of Helsinki

MATTI SIEKKINEN
Aalto University

EEMIL LAGERSPETZ
University of Helsinki

YU XIAO
Aalto University

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

Published in the United States of America by Cambridge University Press, New York

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107042339

First published 2014

Printed in the United Kingdom by *******

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data
ISBN 978-1-107-04233-9 Hardback

Additional resources for this publication at www.cambridge.org/9781107042339

Contents

Preface

Energy modeling and optimization are very important parts of mobile and wireless application development. Recent studies suggest that the battery life of a smartphone has become a critical factor in user satisfaction. Typical mobile applications today consume much more energy than is strictly necessary because of the sub-optimal use of the smartphone's hardware by the software.

This book provides guidelines for smartphone users, methodologies for researchers, and in-depth knowledge of smartphone power management to the public at large. The techniques presented in the book are necessary for developing energy-aware and energy-efficient systems and applications. The book provides the necessary theoretical background and results from the field, and also practical guidance on assessing and optimizing energy efficiency.

In this book we study the following two questions: What is the power consumption of smartphones and applications, and what are the potential solutions for optimizing smartphone power consumption?

Mobile device power management is facing new challenges posed by the revolutionary development of mobile networks, devices, and applications. Smartphones are complex systems, and it is hard to anticipate user behavior and the way the operating system (OS) and applications use the underlying hardware resources. Thus, advanced techniques are needed first to understand the power-consumption behavior, and then to optimize the hardware/software design to improve energy efficiency.

The book has been written with three key aims in mind:

- Holistic: This book is not strictly about smartphone hardware or software; both are taken into account when considering energy optimization.
- Forward-looking: Some of the advanced techniques detailed in the book have been recently proposed in the scientific community.
- Hands-on: This book provides many practical examples.

Organization of the book

The book has three parts:

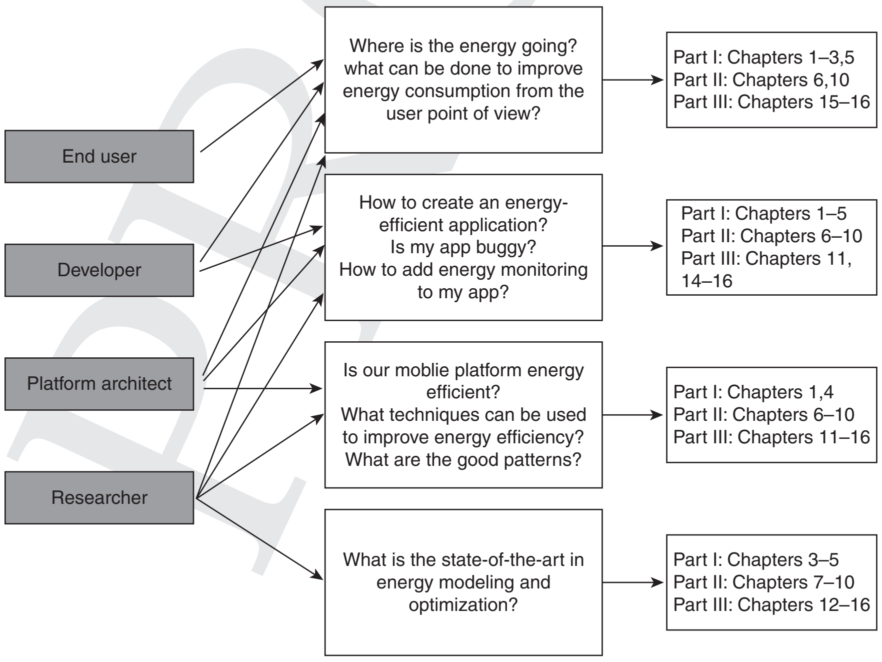
- Part I: Understanding energy consumption. This part presents the overview and the basic concepts relating to energy measurement. We concentrate on the basics of the energy consumption of smartphones, that is where does the energy go and why. We describe alternative ways to measure the energy consumption.

- Part II: Energy management and conservation. This part gives more detail by focusing on the main energy profiling, modeling, and conservation techniques. We consider power management in existing platforms for smartphones. We cover the three main platforms, that is, iOS, Android, and Windows Phone, and also provide examples based on others. We study different techniques to conserve energy by optimizing the design and implementation of mobile software.
- Part III: Advanced energy optimization. This part considers more advanced optimization techniques, such as traffic scheduling, use of multiple network interfaces, and mobile cloud offloading. We conclude the part and the book with a discussion of future trends.

Reading the book

The figure presents the key target audiences for the book: the end users of smartphones, mobile developers and platform architects, and students and researchers. The figure outlines the key questions that the book addresses as well as the pertinent chapters.

End users are typically interested in maximizing the remaining operating time of their device, and also knowing what use cases are energy consuming. This book explains


Where is the energy going? what can be done to improve energy consumption from the user point of view?
Part I: Chapters 1–3,5
Part II: Chapters 6,10
Part III: Chapters 15–16
End user
How to create an energy-efficient application? Is my app buggy? How to add energy monitoring to my app?
Part I: Chapters 1–5
Part II: Chapters 6–10
Part III: Chapters 11, 14–16
Developer
Platform architect
Is our moblie platform energy efficient? What techniques can be used to improve energy efficiency? What are the good patterns?
Part I: Chapters 1,4
Part II: Chapters 6–10
Part III: Chapters 11–16
Researcher
What is the state-of-the-art in energy modeling and optimization?
Part I: Chapters 3–5
Part II: Chapters 7–10
Part III: Chapters 12–16

The key target audiences for the book

how energy is consumed in smartphones, which can help the end users to adjust their use of their smartphones to extend the battery life. Chapter 5 focuses on human–battery interaction and getting the most out of the remaining battery life.

Mobile developers are interested in creating energy-efficient applications and identifying potential energy-related bottlenecks and bugs in the applications. This requires the use of energy-efficient solutions, as well as energy-profiling and analysis techniques. We cover well-known solutions for application energy profiling and diagnostics starting with basic-energy measurement solutions. Most of these solutions and techniques are covered in Part II of the book.

Platform architects are interested in OS- and middleware-level solutions for power management. These solutions are examined in Parts II and III. Part III focuses on advanced platform-level solutions and such as computational offloading and traffic scheduling and offloading.

Researchers are interested in the state-of-the-art techniques and either applying them to solve a specific problem or extending them beyond the state of the art. The book provides a summary of the state of the art for them. These techniques are covered in Parts II and III.

Contributors

We would like to thank people who made valuable contributions to this book. PhD student Aaron Yi Ding of University of Helsinki and Professor Jon Crowcroft of Cambridge University contributed materials that form Section 13.6. PhD student Samuli Hemminki of University of Helsinki contributed the sections on energy-efficient sensing in Sections 7.4 and 15.2.

Dr. Mohammad Ashraful Hoque (Sections 7.3, 12.2, and 15.1), Maria von Kügelgen (Section 5.2), Cenyu Shen (Section 5.3), Cagatay Ulusoy (Section 7.3), Swaminathan Vasanth Rajaraman (Section 7.5), and Ville Koivunen (Section 12.4) contributed by providing measurement data and graphical visualizations of it.

Acknowledgments

This work was supported by the Academy of Finland, grant numbers 139144, 253860 and 268096. This work was also supported by TEKES as part of the Internet of Things program of DIGILE (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business).

Abbreviations

3GPP	3rd Generation Partnership Project
ACF	Autocorrelation Function
ACK	Acknowledgment
ACPI	Advanced Configuration and Power Interface
ADSL	Asymmetric Digital Subscriber Line
ADU	Application Data Unit
AFH	Adaptive Frequency-Hopping
AIDL	Android Interface Description Language
AMOLED	Active-Matrix Organic Light-Emitting Diode
AP	Access Point
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
APM	Advanced Power Management
AR	Autoregressive
ARIMA	Autoregressive Integrated Moving Average
ARO	Application Resource Optimizer
BIOS	Basic Input/Output System
BLE	Bluetooth Low Energy
CAM	Continuously Active Mode
CBR	Constant Bit-Rate
CC/CV	Constant Current/Constant Voltage
CCD	Charged-Coupled Device
CCLF	Cold Cathode Fluorescent Lamps
CDMA	Code Division MultipleAccess
CIL	Common Intermediate Language
CISC	Complicated Instruction Set Computer
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLT	Central Limit Theorem
CMOS	Complementary Metal Oxide Semiconductor
CPC	Continuous Packet Connectivity
CPU	Central Processing Unit
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance

D2D	Device to Device
DASH	Dynamic Adaptive Streaming over HTTP
DDI	Device Driver Interface
DDMS	Dalvik Debug Monitor Server
DEF	Dalvik Executable Format
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DMS	Domain Management System
DNS	Domain Name System
DOD	Depth of Discharge
DPM	Dynamic Power Management
DPS	Dynamic Power Switching
DRAM	Dynamic Random Access Memory
DSA	Digital Signature Algorithm
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
ECDSA	Elliptic Curve Digital Signature Algorithm
eNBs	Evolved Node B
EPS	Evolved Packet System
ESSID	Electronic Service Set Identifier
FD	Fast Dormancy
FSM	Finite State Machine
GPRS	General Packet Radio Service
GPU	Graphics Processing Unit
GSM	Global System for Mobile Communications
HAL	Hardware Abstraction Layer
HBI	Human–Battery Interaction
HPC	Hardware Performance Counter
HSPA	High Speed Packet Access
i.i.d.	independent identically distributed
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
IPS LCD	In-Plane Switching Liquid Crystal Display
ISP	Image Signal Processor
ISP	Internet Service Provider
ITU	International Telecommunication Union
IVA	Image, Video, Audio
JNI	Java Native Interface
JVM	Java Virtual Machine
LCD	Liquid Crystal Display
Li-Ion	Lithium-Ion

Li-Po	Lithium-Polymer
LTE	Long Term Evolution
LTE-A	Long Term Evolution Advanced
M2M	Machine-to-Machine
MA	Moving Average
MAC	Medium Access Control
MAPE	Mean Absolute Percentage Error
McBSP	Multichannel Buffered Serial Port
MIMO	Multiple Input, Multiple Output
MIPI	Mobile Industry Processor Interface
MSE	Mean Square Error
NDK	Native Development Kit
NEP	Nokia Energy Profiler
NFC	Near Field Communication
NFI	Newton Forward Interpolation
NiMH	Nickel Metal Hydride
NMSE	Normalized Mean Square Error
NSRM	Network Socket Request Manager
OCV	Open Circuit Voltage
OLED	Organic Light-Emitting Diode
OFDMA	Orthogonal Frequency Division Multiple Access
OMAP	Open Media Applications Platform
OS	Operating System
P2P	Peer to Peer
PACF	Partial Autocorrelation Function
PCA	Principal Component Analysis
PDA	Personal Data Assistant
PDU	Protocol Data Unit
PS	Packet-Switched
PSM	Power Saving Mechanism or Power Saving Mode
PSMP	Power Save Multi-Poll
QQ	Quality and Quantity
QoE	Quality of Experience
QoS	Quality of Service
RF	Radio Frequency
RIL	Radio Interface Layer
RISC	Reduced Instruction Set Computer
RMS	Root Mean Square
RRC	Radio Resource Control
RSA	Right Shift Algorithm
RSSI	Received Signal Strength Indicator
RTT	Round-Trip Time

SDK	Software Development Kit
SER	Standard Error of Regression
SMPS	Spatial Multiplexing Power Save
SNR	Signal-to-Noise Ratio
SOC	State of Charge
SoC	System on a Chip
SOD	State of Discharge
SOH	State of Health
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TDM	Time-Division Multiplexing
TDMA	Time Division Multiple Access
TFT	Thin Film Transistor
TLS	Transport Layer Security
UDP	User Datagram Protocol
ULP	Ultra Low Power
UMTS	Universal Mobile Telecommunications Service
USB	Universal Serial Bus
VoIP	Voice-over-IP
VoLTE	Voice over LTE
WCDMA	Wideband Code Division Multiple Access
WLAN	Wireless Local Area Network
WNI	Wireless Network Interface

Part I

Understanding energy consumption

1 Introduction

The last ten years has been the era of personal and social communications, with the rapid proliferation of smartphones that provide always-on connectivity with people and information. The mobile computing environment has changed over the years in many ways. Today's devices are powerful computers and sensing systems that have versatile communications capabilities, and they are capable of running native and web browser-based applications that can tap into system resources such as various on-board sensors. The devices also come in many forms and shapes, such as small and large form-factor phones, tablets, and wristwatches.

Understanding smartphone and mobile device energy consumption is a vital and challenging problem. It is vital, because the remaining operating time of a device should be understood and maximized when necessary. It is challenging, because a device consists of various hardware and software systems that work together. Various modeling, prediction, and optimization techniques are needed to engineer energy-efficient mobile systems.

1.1 Overview and the environment

Energy efficiency in mobile computing, especially in the wireless data transmission involved in mobile applications, is one of the challenges that has attracted much attention from mobile device manufacturers, application providers, and network operators. Compared with traditional telephone services, such as voice calls and short message service, executing modern mobile applications consumes much more computing and networking resources and therefore demands much more energy. However, battery technology has not developed as fast as mobile computing technology and has not been able to satisfy the increasing energy demand. This has directly resulted in a dramatic decrease in battery life, that is the time until the next charge.

For example, the battery life of a mobile device may drop to between three and six hours, if the mobile user is using internet services such as video streaming and web browsing. Hence, energy efficiency in mobile computing, although it is a research area that has been established for more than a decade, has once again become a hot topic. A major target of this research area is to develop techniques for reducing the energy consumption of mobile devices while trying to maintain the device performance, which is possible because very often the devices consume more energy than is strictly necessary for a particular task.

About a decade ago when this research area was popular, the research focus was on the energy efficiency of computation [1], such as the energy consumption of microprocessors, since mobile internet services such as email were still in their early stages. Today, mobile devices, as well as application scenarios, have changed drastically.

With mobile internet services becoming popular, wireless data transmission is becoming a major source of energy consumption on mobile internet devices. Additionally, with more sensors, such as global positioning system (GPS) receivers, available on the devices, the context monitoring and its energy consumption also becomes a challenge. Hence, now is the right time to revisit energy-efficient techniques and to develop techniques to solve the existing and upcoming challenges. Indeed, the mobile systems research community has become very active in the area of smartphone energy consumption analysis and optimization with many recent proposals and results in the field.

1.1.1 A holistic approach

A holistic approach is needed to understand the energy consumption of a complex system such as a smartphone. Figure 1.1 gives an overview of the different facets that need to be considered when examining smartphone energy consumption. The energy draw of the device is determined by its physical properties and hardware components. Therefore the power profiles of the chipset, hardware accelerators, and dedicated hardware-level functions are crucial. At the device level, pertinent issues include parallelism and scheduling over multiple cores, as well as intra-device optimization. At this level, the impact of applications and processes needs to be understood by mapping software activities to hardware resources. This mapping can be done on different levels of abstraction from the process level to the system call level.

Expanding beyond the device, at the inter-device level, optimizations can be carried out across devices typically in the local communication context. This is a new and emerging area of optimization with only a few examples at the moment. A more frequently employed approach is to connect the smartphone operating system and applications to the Internet to be able to send power-hungry tasks to the fixed network for processing. This process is called offloading and it is frequently used to alleviate resource limitations at the device.

The computing environment is becoming increasingly complex and distributed. A smartphone may interact with other phones and wearable items, such as smart watches, and augmented reality devices, such as the Google Glass prototype and product. Figure 1.2 illustrates the distributed mobile environment. Smartphones typically have one or more connections with internet servers. These connections are used to synchronize the platform and application state between the mobile device and the cloud. For example, an always-on connection is used to push asynchronous updates to mobile devices. Smartphones can communicate in the local context through protocols such as Bluetooth, Bluetooth Low Energy, and Wi-Fi Direct. The local communication today is still quite

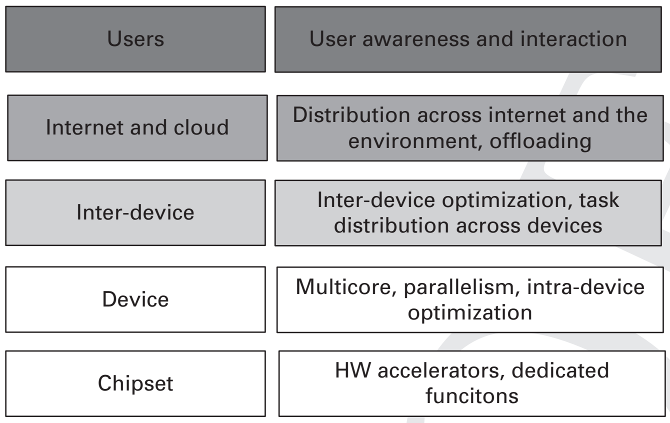

Users
User awareness and interaction
Internet and cloud
Distribution across internet and the environment, offloading
Inter-device
Inter-device optimization, task distribution across devices
Device
Multicore, parallelism, intra-device optimization
Chipset
HW accelerators, dedicated funcitons

Figure 1.1 A holistic view to smartphone energy consumption

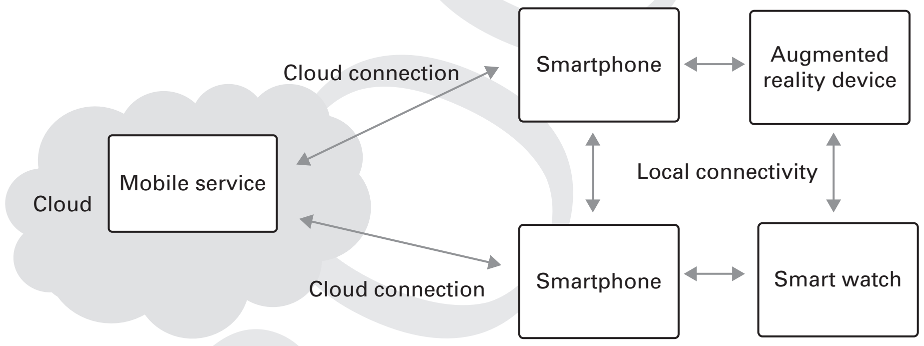


Figure 1.2 Overview of the mobile environment

limited, but with the advent of wearable devices, such as smart watches, and augmented reality devices, this is expected to change. This environment will be a fertile ground for new innovative applications that sense and interact with users in myriad ways; however, it also presents many challenges including energy efficiency, user interaction, security, software configuration, and communications.

Finally, we have the user and groups of users that carry the devices and interact with their software and hardware. Ultimately, the way people interact with the devices has a very large impact on the use of hardware resources and thus the overall energy consumption. In addition, the system can give recommendations to users on how to improve the energy efficiency of the system. This can be achieved through options that the users can set, such as the back-light and connectivity modes, or by recommending software updates or software-use strategies. For example, on the Samsung Galaxy S4 users can control the battery use by adjusting the power-saving mode settings. These settings relate to the CPU, screen, background color, and haptic feedback. Indeed, the term human–battery interaction (HBI) has recently been coined to describe how users charge and discharge the smartphone battery and manage the settings relating to battery use [2]. Thus we have to factor in the human element to understand the real-life energy consumption of smartphones.

1.1.2 Stakeholders

Figure 1.3 presents the key stakeholders for feature phones and smartphones. Feature phones, or GSM phones, are closed embedded devices that typically do not support extensive third-party software. Today the differences between feature phones and smartphones relate specifically to the extent of the application ecosystem, the hardware and software subsystems available for applications, and advanced operating system features such as multitasking. The stakeholders have an interest to ensure that the devices and the ecosystem around the devices function properly and that the devices are energy efficient. Today this is much harder to ensure given the complexity of the devices and their capabilities, as well as the huge numbers of applications using the device capabilities in unpredictable ways.

The stakeholders have different interests and requirements:

- End users require the mobile devices to have long operating times without affecting usability and performance.
- Device manufacturers are responsible for the smartphone hardware integration into a complete product and sometimes also the mobile platform. Each hardware component needs to be optimized for energy efficiency; but this is not enough, the combinations of components in runtime usage also need to be considered and optimized.
- Telecom equipment manufacturers, such as Nokia Solutions and Networks (NSN), Ericsson, and Huawei, develop wireless network technologies, such as the 3G and 4G cellular networks. Energy efficiency is a key requirement for these networks both for the mobile devices and the infrastructure.

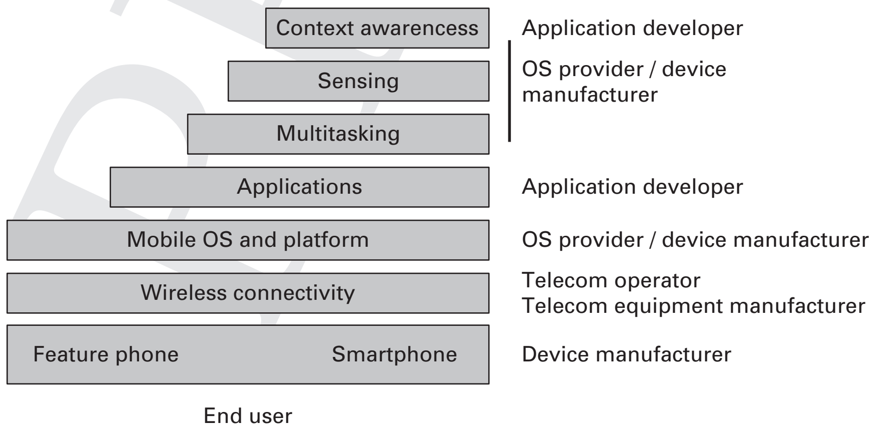


Figure 1.3 Key stakeholders in smartphone development

- Telecom operators are interested in keeping the end users satisfied and offering high-quality services. Therefore operators have an interest in ensuring that harmful applications are not run on the phones and that the network usage is efficient. On the other hand, mobile phone energy consumption is not the primary concern of mobile network operators. Operators are indirectly addressing mobile phone energy consumption through the configuration and deployment of wireless access networks.
- Platform providers are responsible for the operating system and many of the middleware components inside the mobile platform. For example, Google, Microsoft, and Mozilla are developing their mobile platforms that are then used by third parties. Energy efficiency is a key requirement for the OS and the middleware. The current mobile platforms employ various solutions to monitor and control energy consumption.
- Application developers aim to create the best possible applications for end users. Low energy consumption is one requirement for a good application. The end users would not consider purchasing an application that drains their battery in one hour if there are competing applications that work much longer. Therefore the mobile developer is interested in ensuring that the overall energy consumption of the application is in line with expectations and that there are no energy- or performance-related software bugs.

1.2 Smartphone hardware and software

From the hardware perspective, the modern smartphone is a general purpose computing device and a sensing system that consists of the essential components of a computer with dedicated signal processing and radio subsystems as well as a battery. The rechargeable battery stores energy that is discharged when powering the device. The software part of the device consists of the operating system (OS) including the device drivers, the middleware and libraries, and mobile applications running on top of the OS and middleware.

Compared to the feature phone, the smartphone has gained quite a lot in terms of software and hardware complexity. Early feature phones, or GSM handsets, were closed systems that did not support third-party software on the devices. These devices were designed for two specific tasks: voice communication and simple text messaging. For such devices it is relatively straightforward to develop energy-consumption models and then optimize the devices based on the models. In contrast, smartphones support third-party software and have many new functions, such as multiple radios and communication techniques: TCP/IP, web and email, FM radio, GPS reception, music and video playback, and so on.

Table 1.1 presents example smartphones and their characteristics (adapted from [3]). We observe that many of the smartphone properties are evolving at an exponential pace. The cellular technology evolution can be seen through five year leaps. The downlink speeds have improved by a ten-fold factor at every leap. The display size is increasing in powers of two and the size of the software on the device is increasing in powers of ten, with the latest generation having a preset footprint of several gigabytes and the capacity

Table 1.1. Evolution of mobile phones

	1995	2000	2005	2010	2015
Cellular generation	2G	2.5–3G	3.5G	Transition towards 4G	4G
Standard	GSM	GPRS	HSPA	HSPA, LTE	LTE, LTE-A
Downlink (Mb/s)	0.01	0.1	1	10	100
Display pixels (× 1000)	4	16	64	256	1024
Comms modules	–	–	WiFi, Bluetooth	WiFi, Bluetooth	WiFi, Bluetooth LE, RFID
Battery capacity (Wh)	1	2	3	4	5

for tens of gigabytes of user content. The battery capacity in terms of watt-hours is increasing linearly.

Battery technology has not developed as fast as smartphone power requirements, making energy a limited resource. Battery capacity is determined and limited by its chemical properties. Thermal limitations must also be taken into account in the design of smartphones and other mobile battery-powered devices. Without active cooling the power consumption of a small device is limited to approximately three watts [4]. Battery capacity can be increased with physically larger batteries; however, this would lead to larger devices which is not a desirable solution. Indeed, alternative solutions are highly sought after by academia and industry.

An important part of the added value of the devices comes from the application-centered ecosystems, in which third-party developers develop and distribute mobile applications through application stores or similar marketplaces. The mobile application market started its tremendous growth in 2008 with the advent of the Apple AppStore and since then hundreds of thousands of mobile applications have been made available in various stores and distributed to mobile devices.

Interest in the energy consumption of devices has shifted from the energy-efficient design of feature phones for a small number of basic tasks to the overall energy optimization of smartphones that run arbitrary third-party software. In addition to energy-efficient hardware, the OS and platform developers need to consider the energy consumption of the device and its components to be able to optimize energy consumption in specific usage scenarios.

Energy problems have increased with the growth of the smartphone and tablet markets and with the phenomenal success of mobile software. Smartphone users are frequently asking for advice on how to improve the operating time of their devices, and there are many mobile applications for tracking energy usage of the devices. The general advice for smartphone users is to turn off extra functionality such as Wi-Fi or GPS, dim the background light, and avoid the use of applications that consume a lot of energy. Concern over energy consumption has also resulted in new kinds of bug called energy bug (ebugs) that are software or hardware errors that manifest in excessive energy consumption [5].

Indeed, the above concerns and requirements toward energy efficiency have resulted in the rise of battery-monitoring applications and overall interest in detecting unusual energy behavior of applications and hardware components.

Given that heavy battery use may discourage consumers from downloading and using an application, application developers have become interested in optimizing their application design for energy efficiency. This is a new field and, although some tools exist, there is no well-established technique for analyzing the energy consumption of mobile applications.

In this book, our central tenet is that smartphone energy consumption can be significantly reduced by focusing on the design of applications, middleware, and the OS taking the underlying hardware into account. Thus it is central to understand the higher-level activities of the applications, what happens within the software stack, and how those activities map to hardware resource consumption.

1.2.1 Increased capabilities of smartphones

The increasing coverage of mobile broadband networks has been accompanied by a significant boost in the capabilities of mobile devices. Feature phones and personal data assistants (PDAs) are being replaced by smartphones and sub-notebook form-factor tablet devices, such as Apple's iPad series, Samsung's Galaxy Tab, and other Android tablets. The new devices are equipped with high-performance processors, large-volume storage, multiple network interfaces, high-resolution displays, and rich sensors. All these capabilities together make it possible for mobile devices to handle much more complex tasks. It has opened a door to mobile applications that require heavy computation, high-speed data transmission, and rich context information.

Table 1.2 presents example smartphones and their capabilities. The ARM processors dominate the smartphone market and almost all current smartphones use ARM-based CPUs. The clock speed of the devices has increased over the years: more than tripled from 2008 to 2013. In addition, the battery capacity has increased, but as mentioned above the capacity has fallen behind the other capabilities.

The sensing capabilities of smartphones have also increased over the years. Today's smartphone has many ways to probe the internal and external operating environment. Internal sensors monitor the status of the battery, CPU, and wireless networks. External sensors monitor and estimate the orientation and acceleration of the device, the location, physical proximity, compass direction, temperature, atmospheric pressure, humidity, and user gestures. Box 1.1 outlines the sensors on the Samsung Galaxy S4 smartphone. This smartphone has an interesting solution for controlling the onboard sensors. The device uses an Atmel AVR microcontroller-based sensor hub for managing the sensors in real time.

Table 1.2. Example smartphones and their characteristics

Device	CPU	Clock speed	Issue	Year	Battery
Apple iPhone 3G	Samsung ARM11	412 MHz	Single	2008	1219 mAh
Nokia N97	ARM11	434 MHz	Single	2009	1500 mAh
Nokia N900	TI ARM Cortex A8	600 MHz	Dual	2009	1320 mAh
Apple iPhone 4	Apple ARM Cortex A8	800 MHz	Dual	2010	1420 mAh
Samsung Galaxy Nexus	TI ARM CortexA9	1200 MHz	Dual	2011	1750 mAh
Apple iPhone 5	Apple A6	1300 MHz	Dual	2012	1440 mAh
Nokia Lumia 920	Qualcomm APQ8055 (Snapdrag on S4)	1500 MHz	Dual	2012	2000 mAh
Samsung Galaxy S4	Qualcomm Krait: ARM Cortex A15 and ARM Cortex A7	1600 MHZ Cortex A15 and 1200 MHz Cortex A7	Quad	2013	2600 mAh

Box 1.1 Samsung Galaxy S4 sensors

- Accelerometer
- Gyroscope
- Proximity
- Compass
- Barometer
- Temperature
- Humidity
- Gesture

Example of context-awareness: The Motorola Moto X[a] released in August 2013 is based on the Qualcomm Snapdragon S4 pro (1.7 GHz dual-core Krait CPU), quad-core Adreno 320 GPU, a natural language processor, and a contextual computing processor. The device has a 2200 mAh battery and runs the Android 4.2.2 OS extended with context-awareness support. Specifically, the new feature of the phone is continuous audio sensing with the Google Now product. The device can wake at any time to hear the user's voice and react to the command and the surroundings. Indeed, context awareness is now being introduced to mobile applications and operating systems. This places even greater requirements on the energy efficiency of the software and hardware to keep the operating times of the devices reasonable.

[a] `http://www.motorola.com/us/shop-all-mobile-phones/Moto-X/FLEXR1.html` accessed January 6, 2014

1.3 Mobile communications

During the previous decade, the wireless industry has moved into the Internet era and we have witnessed revolutionary changes in wireless networks, mobile devices, and mobile applications. Mobile broadband networks have become ubiquitous with the advent of cellular technologies, such as 3G and LTE, and WLAN networks. Wireless Local Area Networks (WLANs) are non-cellular wireless broadband networks. The Wi-Fi Alliance defines Wi-Fi as any wireless network based on IEEE 802.11 standards. In practice, WLAN and Wi-Fi are considered to be synonymous, although WLAN also relates to wireless standards other than 802.11. In this book, by WLAN we mean specifically IEEE 802.11 technologies unless indicated otherwise.

1.3.1 Network traffic growth

Mobile broadband networks that provide high-speed internet access have been widely deployed. In 2010, 3G cellular networks were available in 143 countries and had 694 million subscribers. In addition, over 750,000 Wi-Fi hotspots had been installed and were used by 700 million people around the world[1]. According to the estimates by the ITU, there were over 1 billion 3G subscribers at the end of 2011 with an annual growth rate of 45%. In addition to 3G, significant growth is expected for the next generation Long Term Evolution (LTE) and LTE Advanced (LTE-A) networks that are now being introduced by mobile operators. Cisco's recent estimates[2] state that 76% of global mobile devices are supported by 2G networks, 23% by 3G networks, and 1% by 4G networks. Cisco estimates that the share of 4G will grow to 10% by 2017 with 45% share of the total mobile data traffic. The role of Wi-Fi networks is becoming important with 33% of mobile data traffic being offloaded from cellular to Wi-Fi or fixed networks in 2012. Cisco's report expects this number of grow to 46% by 2017.

A recent Ericsson Mobility Report[3] reported 6.6 billion mobile subscriptions including over 2 billion mobile broadband users in 2013. The report anticipates 9.3 billion mobile subscriptions by the end of 2019, with a four-fold increase of mobile broadband from 2 to 8 billion. The number of LTE subscriptions is expected to exceed 2.6 billion by the end of 2019.

Smartphone traffic has grown tremendously over the last five years. Figure 1.4 illustrates the growth predictions of mobile traffic by device type. In addition, laptops and tablets are also driving traffic growth. Machine-to-Machine (M2M) traffic is a new development, and it is expected to grow significantly in the next five years, because of the emergence of various networked sensors and actuators.

According to Cisco's Visual Networking Index report published in 2013, the average mobile user consumed 201 megabytes of data per month in 2012. This traffic volume is

[1] Cisco. The future of hotspots: Making Wi-Fi as secure and easy to use as cellular, 2011.
[2] Cisco. Visual Networking Index: Global Mobile Data Traffic Forecast Update 2012–2017, 2013.
[3] `http://www.ericsson.com/mobility-report`, accessed January 6, 2014

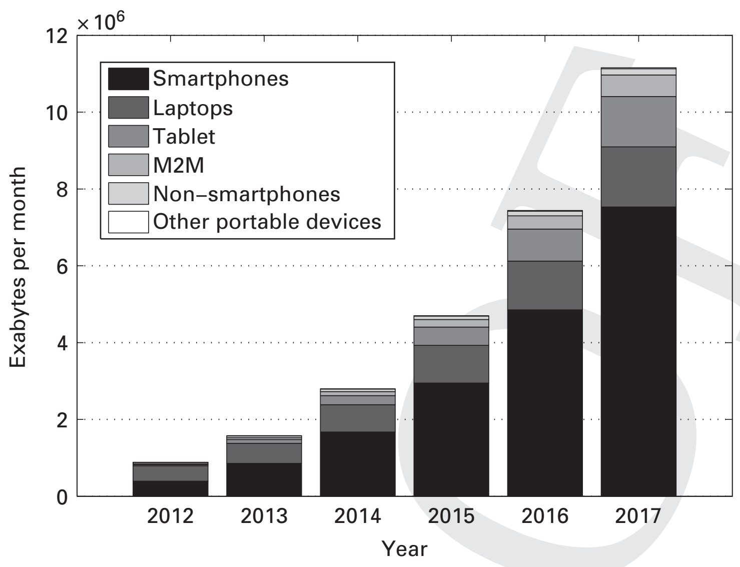


Figure 1.4 Traffic growth by device type

expected to grow to 2 gigabytes of data per month by 2017. The video and audio playback hours are growing fast, as are other application-related communications. Mobile video is expected to represent 66% of global mobile data traffic by 2017.

1.3.2 Cellular standards

Figure 1.5 illustrates the evolution of the network architecture of the different cellular technologies. A cellular network is a wireless network that consists of cells that are served by at least one fixed-location basestation. Neighboring cells employ different frequencies to minimize interference and to provide sufficient bandwidth. The mobile phone system is an example of a cellular network that is based on licensed spectrum. The 3rd Generation Partnership Project (3GPP) is the main standardization organization for 2G–5G cellular standards.

The overall design has evolved from a hierarchical circuit-switched design to a flat packet-based design. At the same time, the speeds of the wireless links and the core network have increased tremendously from the slow speed of 9.6 kbps in 1991 to several hundreds of kbps in early 2000 to the current speeds of hundreds of Mbps. The next generation cellular networks are expected reach gigabit speeds. Figure 1.6 depicts the evolution of the transfer speeds from 2G GSM to 4G LTE-A. It is notable that the transfer speeds are asymmetric and wireless network designers typically assume that downlink is more frequently used than uplink. This has partially changed over the last decade with user-generated content placing more emphasis on uplink. Indeed, the more recent cellular standards have improved the uplink speeds. 5G is the next generation

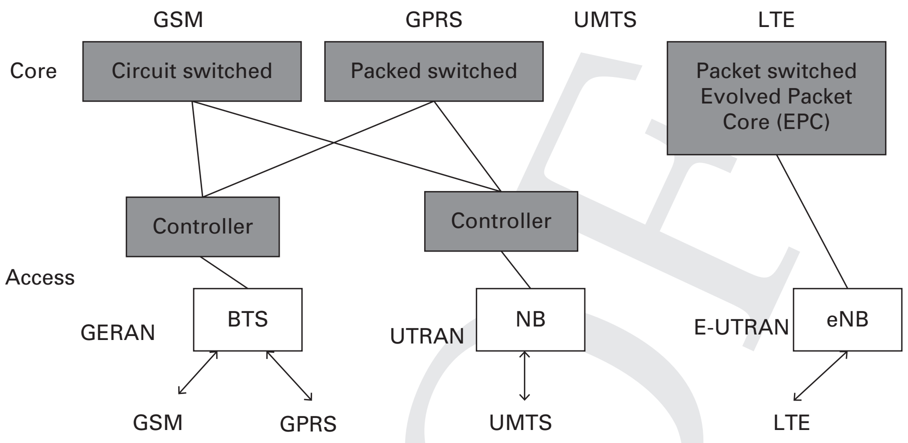
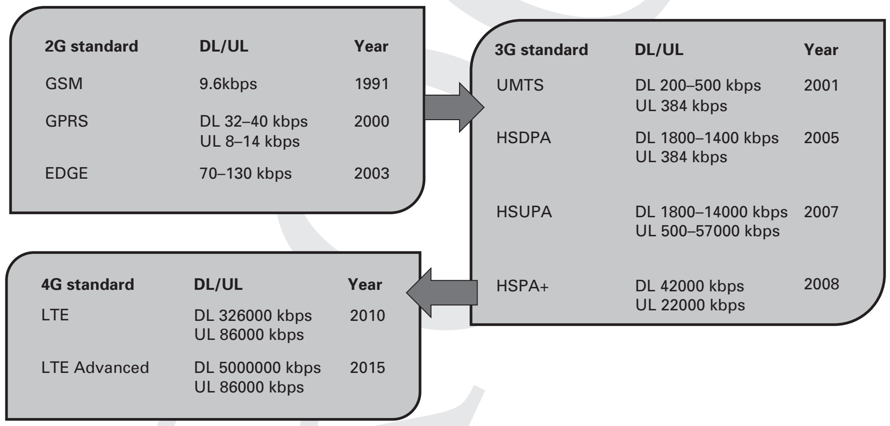


Figure 1.5 3GPP cellular standards overview



2G standard	DL/UL	Year
GSM	9.6kbps	1991
GPRS	DL 32–40 kbps UL 8–14 kbps	2000
EDGE	70–130 kbps	2003

3G standard	DL/UL	Year
UMTS	DL 200–500 kbps UL 384 kbps	2001
HSDPA	DL 1800–1400 kbps UL 384 kbps	2005
HSUPA	DL 1800–14000 kbps UL 500–57000 kbps	2007
HSPA+	DL 42000 kbps UL 22000 kbps	2008

4G standard	DL/UL	Year
LTE	DL 326000 kbps UL 86000 kbps	2010
LTE Advanced	DL 5000000 kbps UL 86000 kbps	2015

Figure 1.6 Evolution of 3GPP cellular standards

cellular network architecture that is now being designed and researched. We discuss 5G briefly in Chapter 16.

GSM was designed for real-time services building on circuit-switched networking. While this design favors real-time voice communications, it is not ideal for data services that have to be realized through slow circuit-switched modem connections. GPRS was designed to address the data limitations of GSM by providing more efficient IP (Internet Protocol) services relying on the same air interface and TDMA (Time Division Multiple Access). With the development of Universal Mobile Telecommunication System (UMTS), the access method was improved by basing it on CDMA (Code Division Multiple Access). UMTS supports both circuit-switched and packet-switched operation. The former is used for voice communications whereas the latter is needed for data services. The user equipment receives an IP address when data services are first used,

Table 1.3. 802.11 wireless standards

Standard	Year	Band	Data rate
802.11b	1999	2.4 GHz	11 MB/s
802.11a	1999	5 GHz	54 Mb/s
802.11g	2003	2.4 GHz	54 Mb/s
802.11n	2009	2.4 GHz, 5 GHz	600 MB/s
802.11ac	–	6 GHz	
802.11ac	–	6 GHz	6.93 Gb/s
802.11ad	–	2.4 GHz, 5 GHz, 60 GHz	6.76 Gb/s

and this IP is given up when the data service is no longer needed. As a consequence, circuit-switched paging is needed for establishing data services.

The current generation of wireless access networks is based on the evolved packet system (EPS) that is IP based. Both voice and data services are built on top of the IP. In EPS, the mobile device receives an IP address when it is turned on and the address is released when the device is turned off. 4G wireless access networks are based on the LTE standard that uses OFDMA (Orthogonal Frequency Division Multiple Access) to obtain faster data rates. Performance improvements include large bandwidth, high-order modulation, and multiple antenna transmission. In addition to LTE, the core network (Evolved Packet Core) can support also non-3GPP technologies, such as Wi-Fi based access networks.

The access network structure has been simplified for LTE with the network consisting of the base stations and the Evolved Node Bs (eNBs). There is no centralized controller and eNBs are interconnected toward the core network through the S1-interface. The aim of this new design is to improve connection setup and handoff performance.

1.3.3 Wi-Fi

Compared with cellular networks, such as 3G or LTE, Wi-Fi is based on unlicensed spectrum and is primarily used as a wireless hotspot technology. Wi-Fi can provide a much higher data rate, up to 7 Gbps with the 802.11ad protocol or 54 Mbps with the older 802.11g WLAN. Table 1.3 gives an overview of the 802.11 standards. The Wi-Fi networks are typically based on hotspots that cover a certain area, such as an office or a café. The coverage of a Wi-Fi base station is smaller than of a 3G/LTE base station and in general the Wi-Fi coverage is fragmented to isolated hotspots whereas cellular coverage is more uniform. Thus it is popular to have both 3G/LTE and Wi-Fi interfaces in a single mobile device and switch between the interfaces depending on the availability and quality of wireless networks.

1.3.4 Near-field standards

Table 1.4 presents an overview of near-field wireless communication standards. The key standards are Bluetooth (802.15.3), IrDA, and ZigBee and IEEE 802.15.4. Of these

Table 1.4. Near-field communication standards

Standard	Frequency	Data rate	Range	Use cases
Bluetooth, 802.15.3	2.4 GHz, 5–7 GHz	1 Mbit/s, up to 480 Mbits/s	5–10 meters	Peripherals
IrDA	870 nm infrared	2.4 kbit/s to 16 Mbits/s	1 meter	Remote control, peripherals
ZigBee, IEEE 802.15.4	868, 915, 2.4GHz	20, 40, 250 kbits/s	100 meters	Wireless sensors

Bluetooth is available on most smartphones and the new Bluetooth Low Energy is becoming more popular as well.

1.3.5 Power saving mechanisms

Wireless standards typically include power-saving mechanisms and techniques. For example, the Wi-Fi 802.11 standard has a power saving mechanism (PSM) that aims to keep the device in the sleep state for as long as possible by switching the mode after data transmission or reception. The sleep state consumes significantly less energy than the receive and transmit states. In 3G WCDMA networks, the Radio Resource Control (RRC) protocol controls the radio resources. The RRC has four different states and it controls state transitions between them. Typically a state transfer from a high-power state into a lower power state occurs after a timeout expires. The state has low energy consumption; however, the state transitions incur IDLE signaling cost that burdens the network. The Fast Dormancy extension aims to minimize the so-called tail energy phenomenon, in which the system lingers in a high-power state after data communication has taken place. With Fast Dormancy, the phone can directly request the network to transition to a low-power state. In 4G LTE networks, the state machine is simplified with only two states: connected and idle. A timer with a typical timeout of 10 s is associated with the transition from the connected to the idle state. In addition, LTE has a discontinuous reception mode that keeps the state machine in the connected mode, but with a duty cycle that periodically wakes to check for incoming packets. This mode has a timer that causes a state transfer to the idle state when it expires. The discontinuous mode significantly reduces the energy consumption of the wireless protocol.

1.4 Anatomy of a smartphone

To develop energy-efficient techniques, the first step is to understand how energy is consumed on a smartphone. A smartphone consists of hardware components, such as microprocessors, wireless network interfaces, storage, cameras, and a touchscreen, and software running on top of them. The hardware components are the actual energy consumers. Experimental measurements of smartphones indicate that applications running on the devices spend a major part of their energy in I/O heavy activities, such as using the cellular/Wi-Fi interfaces and GPS [5]. We illustrate the differences between GSM feature phones and smartphones by comparing their hardware and software. We will return to these components of the smartphone later in the book and develop both component- and system-specific energy models.

1.4.1 Feature phones and smartphones

For GSM feature phones, the digital processing units, RF transceivers, and power management units are all integrated on a single system on a chip (SoC). The power amplifier, DRAM (Dynamic Random Access Memory), and flash memories are provided by separate chips. The two classical use cases for feature phones are the standby and talk modes. Today, feature phones can be used to access web resources and run limited applications. The power profiles of the modes are very different. The expected power dissipation of the standby mode is about 5mW and the talk mode is about 800mW. The radio and the power amplifier take a large part of the power dissipation when making a call. The CPU, DSP (Digital Signal Processing), and memories account for approximately 5% of the overall power dissipation in idle and talk modes. In talk mode, the power amplifier is responsible for 1W of power dissipation during a call. In standby mode, the RF receiver dominates due to paging with the base station [3].

The situation is not so clear-cut for smartphones that have specialized subsystems on the SoC for various tasks, and that run much more complex software processes. The modern smartphone consists of various programmable cores and accelerators that are clustered by their function. These components are connected by a locally optimized memory hierarchy or interconnect, and they share access to the off-chip memory. The ARM architecture is the key hardware environment for smartphones and mobile devices. Android, iOS, Windows Phone, Blackberry 10, Firefox OS, and Tizen are all based on the ARM architecture.

The typical smartphone hardware design differs from traditional laptops and desktops in that the highly integrated hardware modules provide specific functionality. Laptops and desktops typically rely more on the general purpose CPU rather than auxiliary processors.

1.4.2 System on chip (SoC)

Figure 1.7 illustrates the hardware subsystems of a typical smartphone. The heart of the SoC is the ARM CPU complemented by a GPU and a Digital Signal Processor (DSP). Typically the GPU is programmable and compliant with specific OpenGL and DirectX standards. The DSP typically features the Image, Video, Audio (IVA) accelerator and an

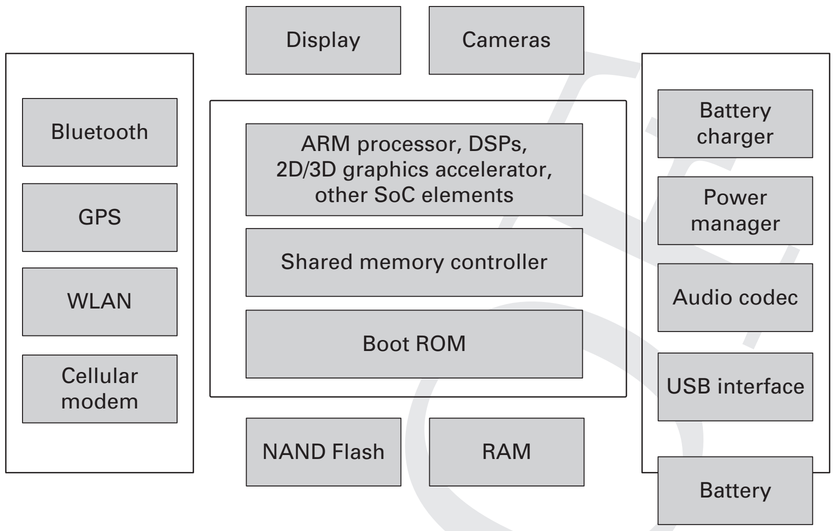


Figure 1.7 Overview of typical smartphone subsystems

Image Signal Processor (ISP). The I/O controllers on the right-hand-side of the diagram are implemented by a specific chip and the internal configuration varies with the design. The Wi-Fi, cellular modem, and other auxiliary devices are connected via system buses, such as the internal USB or multichannel buffered serial port (McBSP) [6, 7]. The Wi-Fi and cellular modem can also be an integrated part of the SoC.

Lower power serial busses facilitate the communication between the internal system components, such as wireless radios, displays, sensors, and the battery. The I^2C bus is a typical example of a low-power serial bus that connects these components together. For mobile devices, these busses need to be low power and this is reflected in their maximum data rates.

The number of CPU cores has been increasing in recent years. First dual-core systems became popular and today we have quad-core and beyond systems. A multicore smartphone may also have a different kinds of cores, for example high-performance cores for heavy tasks and low-performance cores for background processing, such as mail synchronization. Certain OS functions are also being introduced into the hardware, such as the coordination of sensing functions on a smartphone.

Thus the smartphone hardware is becoming more complex and non-deterministic. While these developments aim to improve performance and energy efficiency, they also make it more difficult to understand the energy consumption of the smartphone.

1.4.3 Battery

A battery is a chemical system that stores energy. It consists of reactive components that, when combined, result in the full reaction. The components of a battery are the

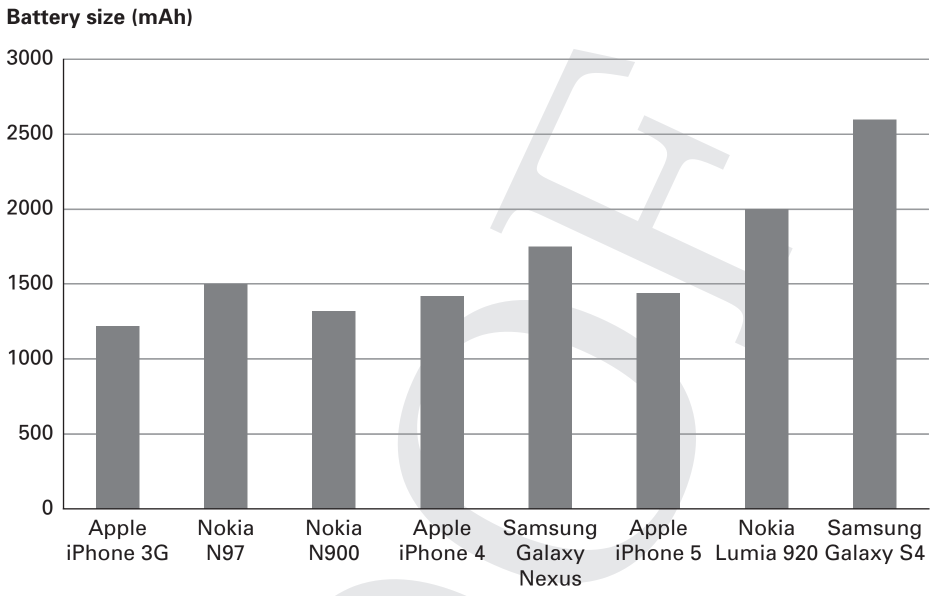


Figure 1.8 Evolution of smartphone battery capacity

metallic lithium, cathode, and electrolyte. The battery life obtainable from a battery can be determined by dividing the capacity, in mWh by the product of the voltage and current demand [8].

Batteries can be classified into non-chargeable primary batteries and rechargeable secondary batteries. Smartphones and other mobile devices have secondary batteries. Lithium-ion batteries are the most popular batteries for smartphones at the moment. They have become popular due to their thinness, lightness, and efficiency. Lithium-ion batteries do not have any memory effect so it is not necessary to empty the battery before recharging. The battery type has the lowest self-discharge rate of 5–10% per month that compares favorably with nickel–metal hydride (NiMH) and nickel–cadmium batteries.

Rechargeable lithium-ion (Li-ion), also known as lithium–metal hydride (LiMH), and lithium polymer (Li-Po) batteries are used in today's smartphones. For example, the iPhone 5 has a lithium-ion battery with a capacity of 1440 mAh and the Galaxy S 4 has a battery with a capacity of 2600 mAh. Figure 1.8 illustrates the evolution of smartphone battery capacities. As mentioned at the beginning of this chapter, batteries have not kept up with Moore's Law that states that the performance of electronics doubles every 18 months. This is due to the chemical nature of batteries and the fact that there are theoretical limits on the amount of electrical energy that can be obtained from the materials [8].

Today's smartphones have considerably larger batteries than emerging wearable computing devices, such as smart watches and augmented reality devices. Many of the current smart watches have smartphone features or integrate with smartphones. Emerging smart watches from Sony and Samsung run a modified version of the Android

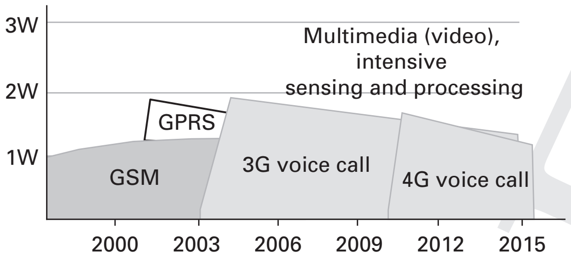


Figure 1.9 Power output limits the phone

operating system. The batteries of these wearable devices are considerably smaller than the batteries of smartphones. For example, the Samsung Gear smart watch has a 315 mAh battery and the Sony Smartwatch 2 has a 140 mAh battery. The emerging Google Glass augmented reality device is expected to have a battery in the 500–600 mAh range.

When discharging and drawing current from the battery, the temperature rise is limited by the available energy. When charging there is no such limit and a safety mechanism must be used to detect when the battery has become fully charged. Minimizing internal resistance also minimizes the heat generation.

The environment also has an impact on the behavior of a battery. High battery temperatures cause heat transfer to the environment through conduction, convection, and radiation. Low battery temperatures on the other hand cause the battery to gain heat from the surroundings. If the temperature of the environment is very high, the thermal management faces significant challenges in maintaining a reasonable operating temperature. Low-power batteries, such as those in smartphones, use protection circuits to maintain the recommended operating temperature limits.

1.4.4 Thermal limits

Figure 1.9 illustrates the power requirements for communications [4]. The power output of the wireless subsystems on a phone is governed by regulation. Three watts is considered to be the cut-off point after which special cooling solutions are needed. A frequently used power budget example for a 3GPP feature phone streaming a video at 384 kb/s consists of 1.2 W for cellular communications, 1 W for A/V and backlight display, and 0.8 W for the CPU and memory [9]. Given the three watt limit, both hardware and software systems need to be optimized to cope with the increasing complexity of the device. Current and forthcoming smartphones use energy optimizations throughout the hardware and software design to minimize power use.

1.4.5 Operating system

The operating system (OS) is responsible for overseeing and managing the hardware and software components on the smartphone and it is responsible for the system-level

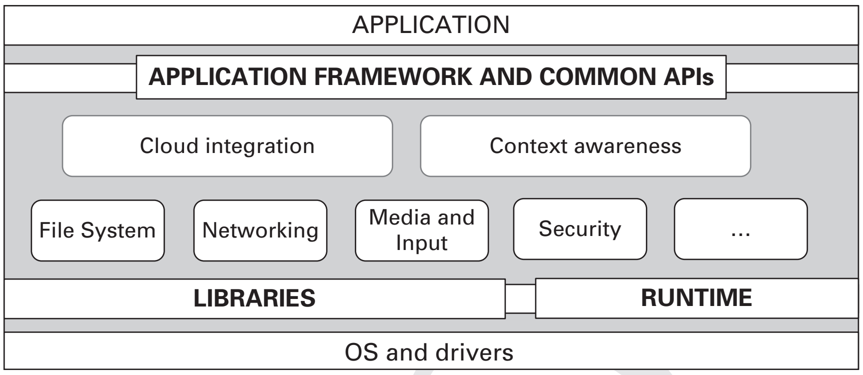


Figure 1.10 A generalized view to the mobile OS

power management. We look at the mobile OSs in more detail in Chapter 8. Figure 1.10 presents a generalized view the mobile OS. The typical OS has a layered structure with the kernel and the drivers being closest to the hardware. The middleware layer includes the libraries, services, and APIs for application support and development. Newer features of mobile OSs include cloud integration and context awareness.

Each mobile OS has its own power management. For example, Android OS has its own power management on top of the standard Linux power management. The design goal is to ensure that the system does not draw power if the applications or services do not require power.

A classical example of OS-level power management is the Advanced Configuration and Power Interface (ACPI) [10] aimed at personal computers. ACPI replaces the older Advanced Power Management (APM) solution and provides an industry standard for OS-level device configuration and power management. The previous standard, APM, allowed control at the BIOS level, whereas ACPI allows this control at the OS level. In APM, the power management starts when the device becomes idle, and the OS has no control or knowledge over this power state change.

1.4.6 Mobile applications

In the last five years, we have seen an explosion of the mobile application market. Today, thousands of new applications are published every day through online shops like Google's Play Store, Amazon's App Store, Apple's App Store, and Microsoft's Windows Phone Marketplace. Figure 1.11 illustrates the growth of these application platforms. The most popular mobile applications are no longer telephone services, but internet services, such as Facebook, YouTube, Google Maps, Twitter, and Pandora. This phenomenon also reflects the ongoing transformation of the wireless industry. Companies such as Apple, Google, and Microsoft, which previously were not wireless companies, are now playing increasing important roles in the mobile device and

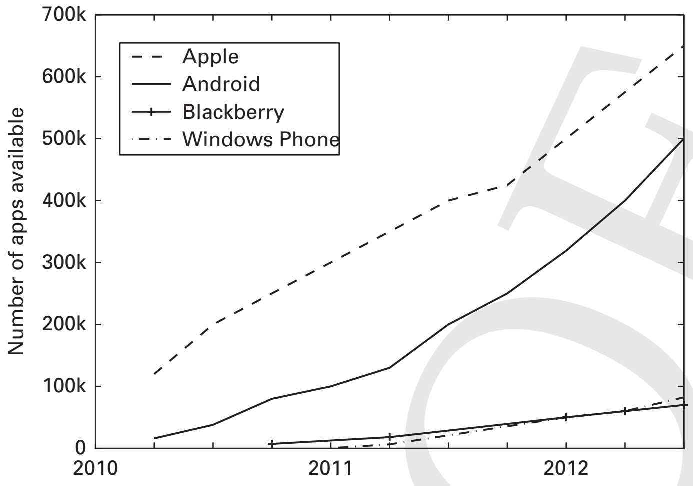


Figure 1.11 Number of applications for different platforms (Source: `http://www.mobilestatistics.com/` accessed January 6, 2014)

application markets. On the other hand, the change in the usage of mobile devices poses new challenges to the research on mobile computing.

1.5 Summary

In this chapter, we introduced the smartphone operating environment and the key challenges from the energy efficiency perspective. Smartphone batteries have not evolved as fast as system and networking performance, emphasizing the need for energy-efficient hardware and software. The smartphone operating environment can be seen to have multiple layers: chipsets and hardware subsystems, integrated device level, inter-device level, Internet and cloud, and the user level. All of these layers need to be considered to understand and optimize smartphone energy consumption.

We discussed how third-party applications may use the hardware of the smartphone in new ways, while feature phones had no such applications and were more predictable in their energy use. We introduced the SoC-based architecture of a smartphone, and how that makes measuring individual components difficult. We discussed the battery, which is limited in size and allowed maximum temperature. The battery technology defines its energy capacity and what kind of hardware it is capable of supporting. We touched on the topic of operating system power management.

References

[1] S. Albers, "Energy-efficient algorithms," *Commun. ACM*, vol. 53, no. 5, pp. 86–96, May 2010.
[2] A. Rahmati, A. Qian, and L. Zhong, "Understanding human-battery interaction on mobile phones," in *Proc. 9th Int. Conf. on Human Computer Interaction with Mobile Devices and*

Services. New York, NY, USA: ACM, 2007, pp. 265–272. [Online]. Available: http://doi.acm.org/10.1145/1377999.1378017

[3] C. H. K. van Berkel, "Multi-core for mobile phones," in *Proc. Conf. Design, Automation and Test in Europe (DATE '09)*, 2009, pp. 1260–1265.

[4] Y. Neuvo, "Cellular phones as embedded systems," in *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2004, pp. 32–37.

[5] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proc. 7th ACM European Conf. on Computer Systems*. New York, NY, USA: ACM, 2012, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168841

[6] J. Kurtto, "Mapping and improving the energy efficiency of the Nokia N900," Master's thesis, Department of Computer Science, University of Helsinki, 2011.

[7] S. Madhavapeddy and B. Carlson, *OMAP 3 Architecture from Texas Instruments opens new horizons for mobile Internet devices*, Texas Instruments, 2008, Whitepaper.

[8] D. Linden and T. B. Reddy, *Handbook of Batteries*. 3rd ed. McGraw-Hill Professional, 2001,

[9] K. Pentikousis, "In search of energy-efficient mobile networking," *IEEE Communications Magazine*, vol. 48, no. 1, pp. 95–103, 2010.

[10] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba, "The ACPI specification: revision 5.0," 2011. [Online]. Available: http://www.acpi.info/spec.htm

2 Energy and power primer

In this chapter, we define the basic concepts necessary to undertake smartphone energy modeling and optimization. We start with the definitions of energy and power and then consider wireless communications and mathematical tools. The key mathematical tools include error, deviation, and statistical significance, as well as linear regression and prediction techniques.

2.1 Energy and power

The two crucial concepts are **energy** and **power**. The former denotes the capacity of a system to perform work. The latter is the rate of energy consumption, that is, how much the system is doing work. The SI unit for energy is the joule (J) defined as the amount of energy required to continuously produce one watt for one second (Ws). The SI unit for power is the watt (W) defined as one joule per second (J/s). Watts describe the instantaneous energy consumption rate of a system at which energy must be provided for the system to operate. Thus watts describe the electric load of a system.

More formally, we defined electric power as,

$$P = \frac{QV}{t} = IV, \tag{2.1}$$

where Q is the electric charge measured in coulombs (C), V is the electric potential energy per unit charge measured in volts (V), t is the time, and I is the electric current measured in amperes (A). One ampere is one coulomb per second. Thus the electric power, P, is produced by the current, I, with charge, Q, every t seconds over a potential difference of V.

The energy cost of a task can be measured by the power, P, and the time duration of the task, T, giving

$$E = PT. \tag{2.2}$$

The energy cost of an arbitrary task can be determined by integrating the power over time as illustrated by Figure 2.1. Given the power as a function, $P(t)$, the previous

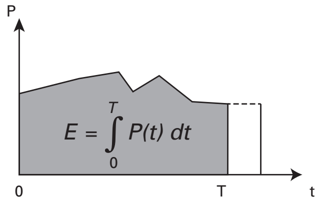


Figure 2.1 The energy cost of a task in time

equation becomes

$$E = \int_0^T P(t)\mathrm{d}t. \tag{2.3}$$

The watt-hour (Wh) is a unit of energy equal to one watt for one hour, or 3.6 kilojoules. For example, if a 1 W device runs for 7 hours, it has consumed 7 Wh. The kilowatt-hour is defined similarly, and it is commonly used as a billing unit for energy to households by electric companies.

The energy content of a battery is typically given in ampere-hours (Ah). The ampere-hour is a unit of electric charge with one ampere-hour being equal to 3600 coulombs. Smartphone battery capacities are commonly given in milliampere-hours (mAh), one mAh being 3.6 coulombs. We can convert watt-ours to ampere-hours by dividing the watt-hour value by the power source voltage (Wh = V Ah). This conversion is approximate, because the voltage is not constant during energy discharge.

2.1.1 Ohm's Law and resistance

The power delivered by the battery or another source of electricity is given by $P = VI$; however, the actual current at any particular instance depends on the load. If we ignore the internal resistance of the battery, the current drawn by the load is given by Ohm's Law. Ohm's Law introduces the constant of proportionality, the resistance (R), by stating that the current through a conductor between two points is directly proportional to the potential difference between the points.

$$I = \frac{V}{R}, \tag{2.4}$$

where I is the current, V is the voltage, and R is the resistance of the conductor in Ohms.

Charging or discharging a battery generates heat due to the internal resistance of the battery. This phenomenon is known as joule heating. The power dissipated due to internal resistance is given by:

$$P = I^2 R_i, \tag{2.5}$$

where I is the current and R_i is the internal resistance.

For example, let us assume that a GPS module takes 280 mW. With 3.7 V this gives the current draw of 280 mW/3.7V=75 mA. Given the battery capacity of a typical smartphone, 2200 mAh, the battery lasts 2200 mAh/75 mA = 29 hours.

2.2 Ideal voltage source and open-circuit voltage (OCV)

An ideal voltage source can maintain a fixed voltage irrespective of the load resistance or the output current. An ideal voltage source alone is not sufficient to model real-life batteries and other sources of electrical energy; however, the ideal source combined with additional combinations of impedance elements are frequently used in modeling. These battery models aim to find relationships between currents and voltages.

Open-circuit voltage (OCV) is frequently used to estimate the state of charge (SOC). The OCV is the difference of the electrical potential between the two terminals of the battery when there is no external load. The internal resistance of the battery is an important parameter of an OCV that has nonlinear properties. The internal resistance depends on the SOC of the battery, the battery temperature, and the method that is used to measure it. The modeling is also complicated by the observation that the OCV and SOC relationship varies between batteries and is affected by temperature, capacity loss, and activation polarization effects [1, 2, 3].

Several nonlinear equations have been developed to model the nonlinear part of the OCV circuit. These models typically include the hysteresis effect that causes the discharge curve to be under the charging curve for the same amount of charge. We will return to these models in Chapter 3.

2.3 Power computation

Energy can be saved on multiple levels in the hardware and software architecture. At the circuit and transistor levels, energy can be saved by changing the voltage and frequency of the circuit. The total power of CMOS logic circuits is determined by the clock rate, the supply voltage, and the capacitances of the transistors. The larger the capacitance, the more the transistors need to work to change the output charge. The higher the source voltage, the more the output has to change. The higher the clock frequency, the more power is wasted during switching. The capacitances are determined during the chip design; however, the clock rate and voltage can be varied at runtime. By varying the clock rate and the supply voltage, it is possible to obtain linear and quadratic improvements, respectively.

The overall power consumption is given by

$$P = P_{switch} + P_{static}, \tag{2.6}$$

where P_{switch} is the dynamic switching power and P_{static} is the static leakage power. The formula does not include the lost power due to the short circuit due to a transistor switching that is a relatively small term in the dynamic switching power [4].

The CPU consists of transistors that can switch states every clock cycle. Each switch of states between a complementary pair of transistors results in wasted power. Typically, most of the gates in the CPU switch states every clock cycle. Thus the higher the clock rate, the higher the amount of wasted energy. In addition to the clock rate, the voltage also affects the power consumption of the CPU.

The dynamic switching loss is the power used by switching of the state of the transistors. This loss can be determined using the formula

$$P_{switch} = \alpha \times C \times V^2 \times f, \tag{2.7}$$

where α denotes the switching activity, C is the average capacitance of the transistors, V is the supply voltage, and f is the clock frequency [5]. The power consumption is proportional to the product of the square of the supply voltage and the frequency. The power consumption decreases quadratically with the reduction of the voltage. The formula is not exact for modern chips, because memory circuits and static leakage current also need to be taken into account.

The clock rate and the supply voltage are parameters for the energy optimization of the CPU [6, 7]. Major energy savings can be made by adjusting either the supply voltage or the clock rate or both of them at the same time. Indeed, dynamic voltage scaling is widely used to improve the power consumption of battery-powered devices. Both low voltage and lowered clock frequencies are used to minimize the power consumption of components such as CPUs and DSPs.

The operating frequency has an approximately linear relationship with the operating voltage given by the following equation [4]:

$$V_{norm} = \beta_1 + \beta_2 \times f_{norm}, \tag{2.8}$$

where $\beta_1 = V_{th}/V_{max}$ and $\beta_2 = 1 - \beta_1$. V_{th} is the threshold voltage at which a transistor conducts and begins to switch.

Figure 2.2 illustrates the relationships among the operating frequency, process completion time, and energy consumption. The process is executed on two different systems at different voltage and frequency levels. The top process runs at high frequency and is completed quickly. The lower process runs at a halved processor frequency and supply voltage and takes much longer to complete. The lower frequency and supply voltage lead to a smaller energy consumption [7]. Suppose that the new frequency is $f/2$, then

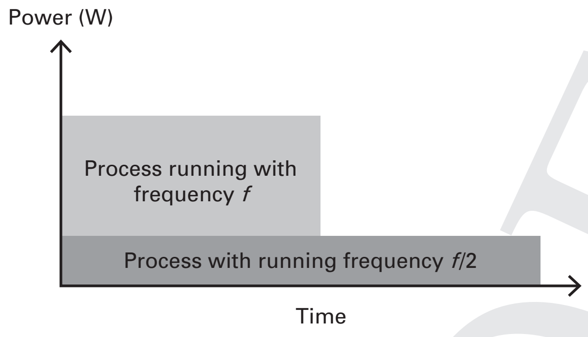


Figure 2.2 Example of frequency, time, and energy consumption

from Eq. (2.7) we obtain the new switching power

$$P'_{switch} = \alpha \times C \times \left(\frac{V}{2}\right)^2 \times \frac{f}{2} = \frac{1}{8}P_{switch}. \tag{2.9}$$

Now, the completion time, T, of the process is doubled for the lower frequency:

$$E'_{switch} = P'_{switch} \times T \times 2 = \frac{1}{8}P_{switch} \times T \times 2 = \frac{1}{4}P_{switch} \times T, \tag{2.10}$$

giving an improvement of $1/4$ with the lower frequency while significantly increasing the execution time of the task. To compensate for the performance loss, we can divide the task into two subtasks and run these in parallel using two cores. The dynamic power can decrease by a factor of two compared to the original case.

Running a task at a slower speed saves energy; however, it will take longer to execute the task thus affecting the performance. Voltage and frequency scaling can help to reduce the energy cost of the CPU; however, it may also increase the energy cost of the overall device due to hardware subsystem usage. Dynamic voltage and frequency scaling favor parallelism and having multiple voltage/frequency scaled cores executing tasks.

2.3.1 Power loss

There are two types of power loss in a CPU chip, resistive loss and leakage. Resistive loss occurs in the conducting lines of the chip and increases when the line width decreases. The chemistry of the metal contributes to the resistive loss. Leakage is caused by quantum tunneling of the electrons through the transistor insulating layers and between interconnections.

Leakage power is a dominant consumer of power in the CPU. The leakage depends on the supply voltage and the area of the gates:

$$P_{leakage} \propto V \times Area. \tag{2.11}$$

A linear decrease of voltage results in a quadratic decrease of the switching loss; however, it only results in a linear decrease of the static leakage. Thus static leakage can become dominant with low-voltage integrated circuits with a high density of transistors. If voltage scaling is not useful due to the dominant role of the leakage power, the processors will become less power efficient.

To address leakage, mobile SoCs are typically designed to allow portions of the chip to be powered off through power gating. Modern SoCs consist of multiple power domains that can be switched off when necessary.

2.4 Radio power

A radio's transmit and receive power is typically measured in the power ratio in decibels (dBm) with reference to one milliwatt. This unit is a convenient measure of absolute power, because it can express both very large and small values. The dBW is a similar unit referenced to one watt. Since dBW is referenced to the watt, it differs from the decibel (dB) that is a dimensionless unit used for assessing the ratio between two values.

The power, P, can be converted to dBm with the formula:

$$x = 10 \times \log 10(1000P). \tag{2.12}$$

Similarly, the dBm value of x can be converted into watts:

$$P = \frac{1}{1000} \times 10^{\frac{x}{10}}. \tag{2.13}$$

Table 2.1 gives example dBm values and the corresponding power levels.

The received signal strength indicator (RSSI) is an important metric in wireless communications that measures the amount of energy associated with the received bits. RSSI is an indicator of the RF signal strength received by the device. RSSI is vendor specific and typically it is indicated in dBm or percentages. For example, the 802.11 standard does not define any mapping between RSSI and the power consumption.

The ambient level of radio energy on the specific channel, the noise floor, should be understood. Ambient noise can stem from microwave ovens, Bluetooth, and other devices.

The signal-to-noise ratio (SNR) measures the level of a signal to the level of the background noise. SNR is defined as a logarithm of the ratio of signal power to the noise power, given in the following equation

$$\log\left(\frac{\mathit{signal\ power}}{\mathit{noise\ power}}\right) = \log(\mathit{signal\ power}) - \log(\mathit{noise\ power}). \tag{2.14}$$

The signal levels in wireless cellular communications are typically given in dBm. The RX level denotes the power of the received cellular signal and the TX level corresponds to the transmission power of the cellular communications. The RX levels are typically

Table 2.1. Example dBm values

dBm	Power	Example
60 dBm	1 kW	Combined radiated RF power of microwave oven
33 dBm	2 W	Maximum output of a Power Class 1 3G phone
27 dBm	500 mW	Typical cellular phone call
20 dBm	100 mW	Bluetooth Class 1 radio (range up to 100 meters)
15 dBm	32 mW	Typical Wi-Fi transmission power
4 dBm	2.5 mW	Bluetooth Class 2 radio (range approx. 30 meters)
0 dBm	1 mW	Bluetooth Class 1 radio (approx. one meter range)

from -30 to -120 dBm and for TX the levels are given in positive dBm. For RX a lower value means a poorer signal and for TX a higher value means higher transmission power.

2.5 Duty cycle

Many devices have a dynamic power demand, for example smartphones and embedded sensing devices. These devices have an idle or sleep power consumption of the order of micro-watts and their active peak power consumption can be of the order of watts. The difference between these two modes, idle and active, can be several orders of magnitude. The battery and power delivery system must support these wide power-demand bandwidths.

Figure 2.3 illustrates the duty cycle that is defined by the active time as a proportion of the total time duration of the cycle. Given a periodic event, the duty cycle is the ratio of the duration of the active state, T_p, to the total duration of the cycle, T:

$$D = T_p/T. \tag{2.15}$$

Figure 2.3 illustrates the transition between the idle and active states and the corresponding times T_p for active and T_s for sleep states for the duration $T = T_p + T_i$. The power spent during time, T, can be determined as the power consumption of the states P_p and P_i for the active and idle states. The power consumption of the duty state is significantly larger than for the idle state. The average power consumption is given by

$$P_{avg} = \frac{P_p T_p + P_i T_i}{T}. \tag{2.16}$$

The duty cycle is very important for wireless communications that typically have at least three states: idle, transmitting, and receiving. The latter two states are the high-power states of wireless radios. The overall power can be optimized by making the

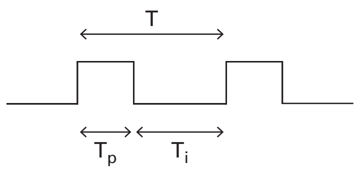


Figure 2.3 Example of a duty cycle

duty cycles short so that the device spends most of its time in a lower power state and minimizes the time in the high-power state. For example, with the wireless example, the radio would have longer intervals between transmissions and receptions and keep the active states as short as possible. Minimizing the duty cycle is also crucial for sensing applications.

A modern smartphone consists of many subsystems that have their own duty cycles. The duty cycle parameters can, in many cases, be changed through software, allowing possible energy improvements. Microcontrollers are responsible for managing various sensors and typically the low-level code is event-driven to avoid polling that can be harmful for performance and has a high overhead. Recently, hardware-based sensor hub solutions have been introduced to even further optimize many on-device sensors. The Samsung Galaxy S4 has an Atmel AVR microcontroller-based sensor hub solution that gathers and processes data from the onboard sensors in real-time. With this kind of coordinated hardware-based solution, it is possible to optimize the duty cycles of the various sensing subsystems. On the other hand, this solution makes it difficult to model the device and allow third-party applications to tune system parameters.

2.6 Mathematical tools

This section discusses mathematical tools for characterizing data and identifying meaningful data properties. We start with a simple example of statistical significance and error, and follow with increasingly advanced tools.

2.6.1 Error, deviation, and statistical significance

With any experiment in any area of natural science, published results should be accurate and reliable. When measuring variable quantities like energy consumption, measurement errors can occur. Typically, hardware measurement devices come with specifications of how small the error is guaranteed to be, such as $\pm$ 2 mA. However, the energy consumption can also vary during the measurement, giving different values even if the error was negligible. Therefore, a sufficient number of repeats for each experiment is required. Using the average of many experiment runs minimizes the error caused by natural variance of the underlying system. The standard deviation or error of the measurements should be mentioned together with the average when publishing the results. With large errors, few conclusions can be made from the result.

Statistical indicators example

Consider the following example. We connect a power meter with high accuracy and sampling rate to a smartphone. Assume that the measurement error of the power meter is ± 8mW. We run an experiment where the phone screen is on, the phone is connected to the Internet via Wi-Fi, and is otherwise idle. We then measure the average power consumption over 10 minutes. We repeat the experiment five times. Assume we get the following results:

1. 1301 mW
2. 1112 mW
3. 978 mW
4. 1212 mW
5. 1155 mW

Our mean (average) power consumption m_x would be 1151.6 mW. The average error or deviation from the mean is given by the formula

$$Error_X = \frac{1}{n}\sum_{i=1}^{n}|x_i - m_x|, \tag{2.17}$$

where x_i is an experiment result and n is the number of results, 5 in our case. Our error would then be $\frac{1}{5} \times (|1301 - 1151.6| + |1112 - 1151.6| + |978 - 1151.6| + |1212 - 1151.6| + |1155 - 1151.6|) = 85.28$ mW. This error is small compared to the mean, $\approx 7.4\%$, so the result is statistically significant. If our measurements were widely different, the error would be larger than the average, and the results would not be reliable.

Another commonly used measure of reliability is the variance and its square root, the standard deviation. The variance of our results is calculated in a similar way to the error, but each difference from the mean is squared:

$$Var_X = \frac{1}{n}\sum_{i=1}^{n}(x_i - m_x)^2, \tag{2.18}$$

The variance of our results is therefore $Var_X = 11537.04$ mW and the standard deviation $\sigma = 107.41$ mW.

2.6.2 Linear regression

Equation (2.19) shows a general form of a linear regression model with p predictor variables [8].

$$f(y_i) = \beta_0 + \sum_{j=1..p} \beta_j g_j(x_{i,j}), \tag{2.19}$$

where $g_j(\mathrm{x}_{i,j})$ is a preprocessing function of the original value of the predictor variable $x_{i,j}$.

Given n observations including the values of p predictor variables and the values of the corresponding responses y_i (i = 1..n), the values of the intercept, β_0, and each coefficient, β_j (j = 1..p), are automatically adjusted during the model fitting toward a model in which the response can be the best predicted from the predictor variables. After the model is built, given the values of the predictor variables, the estimated power consumption will be found.

2.6.3 Prediction techniques

We introduce two predictive models that are widely used in context prediction for context-aware power management. One is a linear predictive model called ARIMA (Autoregressive Integrated Moving Average), while the other is called NFI (Newton Forward Interpolation). The first one requires offline model training, while the other one can be used for online prediction without offline training.

Autoregressive integrated moving average (ARIMA)

Classical linear time series models are based on the idea that the current value of the series can be explained as a function of the past values of the series and some other independent variables. ARIMA is one of the most famous linear predictive models used for network prediction. ARIMA is an integration of three models: an autoregressive model of order p, a moving average model of order q, and a differencing model of order d.

According to the definition in [9], a process, X_t , is said to be ARIMA (p, d, q) if

$$\varphi(B)(1-B)^d X_t = \theta(B)w_t, \tag{2.20}$$

where B is a backward shift operator, $\varphi(B)$ is an autoregressive operator, $\theta(B)$ is a moving average operator, and w_t is assumed to be Gaussian white noise. The three operators can be expressed as below.

$$B^d X_t = X_{t-d}, \tag{2.21}$$

$$\varphi(B) = 1 - \varphi_1 B - \varphi_2 B^2 - \cdots - \varphi_p B^p (\varphi_p <> 0), \tag{2.22}$$

$$\theta(B) = 1 + \theta_1 B + \theta_2 B^2 + \ldots + \theta_q B^q (\theta_q <> 0) \tag{2.23}$$

where $\varphi_1 \ldots \varphi_p$ and $\theta_1 \ldots \theta_q$ are constants.

Let $\hat{Y}(t)$ be the predicted context value at time, t, and $Y(t-i)$ be the observed context value at time $(t-i)$. The time series of the context is considered to be an ARIMA (p, d, q) model if

$$\hat{Y}(t) = \mu + \sum_{i=1}^{d} Y(t-i) + \varphi \sum_{d=1}^{p} Y(t-i) - \theta \sum_{i=1}^{q} e(t-i), \tag{2.24}$$

where $e(t-i)$ is the residual of prediction at time $(t-i)$ and μ is the intercept.

The values of p, d, and q are identified based on the analysis of the autocorrelation function (ACF) and the partial autocorrelation function (PACF), while the estimation of parameters θ, φ, and μ are based on the least squares method which aims to minimize the mean square error (MSE) of residuals as shown in Eq. (2.25).

$$MSE = E(Y(t) - \hat{Y}(t))^2 \tag{2.25}$$

Newton forward interpolation (NFI)

The observation at time, x, in a time series is defined as a function $f(x)$, and the values of x are tabulated at interval h as shown in Eq. (2.26).

$$x = x_0 + i \times h, \tag{2.26}$$

where the first value of x is x_0 and i is the number of intervals between x and x_0. When only a few discrete values of $f(x), i = 0, 1, 2, \ldots$ are known, NFI aims to find the general form of $f(x)$ based on known values by using the finite difference formulas.

Let $f_i \equiv f(x) \equiv f(x_0 + i \times h)$, and the finite forward difference of a function is defined by $\Delta f_i \equiv f_{i+1} - f_i$. Higher orders, such as the $jth(j \le i)$ order forward difference $\Delta^j f_i$, can be obtained by repeating the operations of the forward difference operator j times. For example,

$$\Delta f_0 = f_1 - f_0, \Delta^2 f_0 = \Delta(\Delta f_0)) = f_2 - 2f_1 + f_0 \tag{2.27}$$

The NFI model fits the observation at time x with an i_{th} degree polynomial as below.

$$\begin{aligned} f(x) = f_0 + (x-x_0)\frac{\Delta f_0}{h} + \frac{1}{2!}(x-x_0)(x-x_1)\frac{\Delta^2 f_0}{h^2} \\ + \cdots \frac{1}{(i-1)!}(x-x_0)(x-x_1)\ldots(x-x_{i-1})\frac{\Delta^i f_0}{h^i} \end{aligned} \tag{2.28}$$

When only the past i observations are known, up to the $(i-1)th$ order forward difference can be developed. The prediction of the SNR at time x, corresponding to the $(i+1)th$ observation, is defined as a function $g(x)$.

$$\begin{aligned} g(x) = f_0 + (x-x_0)\frac{\Delta f_0}{h} + \frac{1}{2!}(x-x_0)(x-x_1)\frac{\Delta^2 f_0}{h^2} \\ + \cdots \frac{1}{(i-1)!}(x-x_0)(x-x_1)\ldots(x-x_{i-2})\frac{\Delta^{i-1} f_0}{h^{i-1}} \end{aligned} \tag{2.29}$$

Compared to $f(x)$, the error term, $e(x)$, is approximated as Eq. (2.30).

$$e(x) \cong f_x - g(x) \cong \frac{1}{N!}(x-x_0)(x-x_1)\ldots(x-x_{N-1})\frac{\Delta^N f_0}{h^N} \tag{2.30}$$

Unlike ARIMA, NFI can predict future values online directly based on past observations without offline training. Since the prediction accuracy of NFI depends on the size of N, the size of N can be selected by applying the least squares method to the training data sets as used in ARIMA model fitting.

2.7 Summary

In this chapter, we presented a primer on the concepts relating to energy and power:

- The two key concepts are energy and power. The energy cost of a task can be measured by the power, P, and the time duration of the task, T, by $E = PT$ (Eq. (2.2)).The energy cost of an arbitrary task can be determined by integrating the power over time.
- The energy content of a battery is typically given in ampere-hours (Ah).
- The clock rate and the supply voltage are parameters for the energy optimization of the CPU. Major energy savings can be made by adjusting either the supply voltage or the clock rate or both of them at the same time.
- The ideal voltage source combined with additional combinations of impedance elements are frequently used in modeling (open-circuit voltage (OCV)). These battery models aim to find relationships between currents and voltages.
- A duty cycle determines the active time of a device or a subsystem. The duty cycle is very important for wireless communications that typically have at least three states: idle, transmitting, and receiving.
- The key mathematical tools include error, standard deviation, statistical significance, linear regression, and ARIMA and NFI.

References

[1] S. Lee, J. Kim, J. Lee, and B. Cho, "State-of-charge and capacity estimation of lithium-ion battery using a new open-circuit voltage versus state-of-charge," *J. of Power Sources*, vol. 185, no. 2, pp. 1367–1373, 2008.

[2] O. Erdinc, B. Vural, and M. Uzunoglu, "A dynamic lithium-ion battery model considering the effects of temperature and capacity fading," in *2009 Int. Conf. Clean Electrical Power*, 2009, pp. 383–386.

[3] D. Linden and T. B. Reddy, *Handbook of Batteries*. 3rd ed. McGraw-Hill Professional, 2001.

[4] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003. [Online]. Available: http:// dx.doi.org/10.1109/MC.2003.1250885

[5] D. C. Snowdon, S. M. Petters, and G. Heiser, "Accurate on-line prediction of processor and memory energy usage under voltage scaling," in *Proc. 7th ACM & IEEE Int. Conf.*

on Embedded Software. New York, NY, USA: ACM, 2007, pp. 84–93. [Online]. Available: http://doi.acm.org/10.1145/1289927.1289945

[6] F. Yao, A. Demers, and S. Shenker, “A scheduling model for reduced CPU energy,” in *Proc. 36th Ann. Symp. on Foundations of Computer Science*, ser. FOCS ’95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 374–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795662.796264

[7] S. Zhuravlev, J. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, “Survey of energy-cognizant scheduling techniques,” IEEE Trans. *Parallel Distrib. Syst.*, vol. 24, no. 7, pp. 1447–1464, 2013.

[8] F.I. Kushnirskii and M.E. Primak, “Regression with nonnegative coefficients,” in *Cybernetics and Systems Analysis*, vol. 9, no. 1, 1973.

[9] R. H. Shumway and D. Stoffer, “Time series analysis and its applications: With r examples,” in *Springer Texts in Statistics*, 2006, pp. 85–154.

3 Smartphone batteries

In this chapter we give an overview of smartphone batteries and their modeling. Battery models are important when the remaining battery capacity needs to be estimated. In this chapter, we give an overview of smartphone Li-Po batteries, their static and dynamic parameters and characteristics, and charging technology, and then consider techniques for assessing and modeling a battery SOC.

3.1 Overview

Figure 3.1 presents an overview of the Li-ion battery charging and discharging process. Current is carried by lithium ions (Li+) that move from negative (anode) to positive (cathode) electrodes during discharging. The lithium ions move in the reverse direction when charging the battery. The ions move through the non-aqueous electrolyte and the separator. Applying a charge places the battery in a closed circuit voltage, in which the voltage behavior is set by the internal battery resistance. Charging and discharging distorts the battery and it can take up to 24 hours for the battery to stabilize.

Battery life is proportional to the active reaction sites across the cathode. When the discharge current is low, inactive reaction sites are uniformly distributed over the volume of the cathode. When the discharge current is high, the volume of inactive sites at the outer surface of the cathode is large, causing active sites to become unreachable. As a consequence, the battery capacity is reduced at high discharge rates. When a high current is drawn from the battery, the diffusion rate cannot match the rate at which ions are being absorbed at the cathode. Thus the concentration of positively charged ions diminishes near the cathode while increasing near the anode. This change in the concentration causes the voltage to drop. If the battery is allowed to rest, diffusion causes the concentration gradient to decrease resulting in the recovery of charge.

Lithium batteries have been investigated since 1912; however, the commercialization of the technology started in the early 1970s. The batteries first used lithium metal on the anodes that was later observed to be an unstable anode material when charging. This called for a non-metallic solution and lithium ions proved to be a more stable and efficient solution although Li-Ion batteries require safety measures when charging and discharging. The first commercial Li-Ion battery was released by Sony in 1990. The Li-Ion battery technology has become very successful since the 1990s, and today most portable devices use these batteries. Improved material developments have increased the

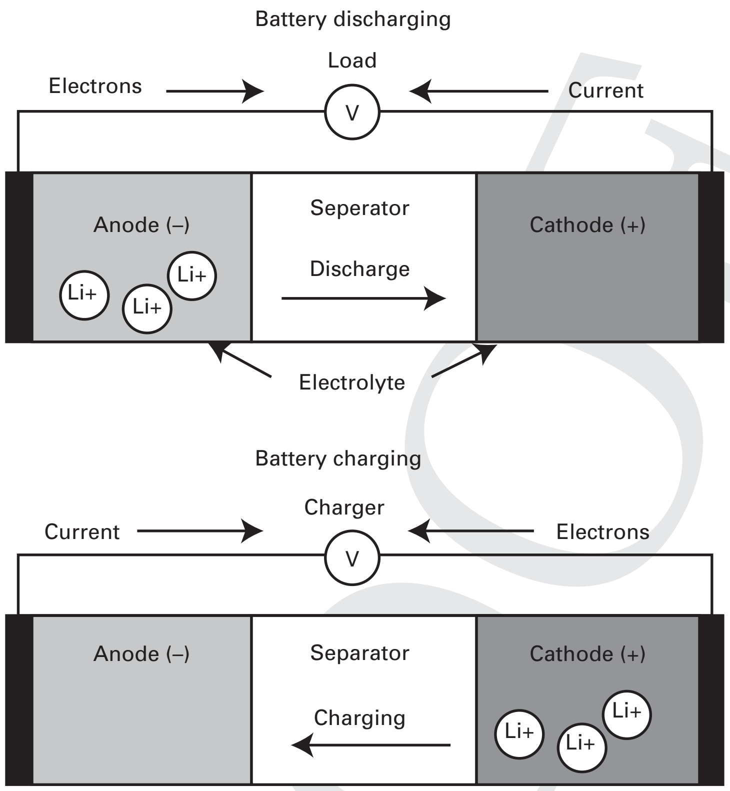


Figure 3.1 Overview of Li-Ion battery charging and discharging

energy density from 75 to 200 Wh/kg and the cycle life of the batteries has increased as well. Li-Ion batteries have efficiencies very close to 100% [1].

Lithium cobalt oxide is a frequently used compound in smartphone Li-Ion batteries; however, there are many other compounds that have their own advantages and disadvantages. The voltage of a typical Li-Ion battery is 3.6 V or 3.7 V, the specific energy is 130–200 Wh/kg with cycle lives of up to 1000 and more depending on the technology [2, 3]. The self-discharge rate of Li-Ion batteries is low, between 5 and 10%. Moreover, Li-Ion batteries do not suffer from the memory effect found in nickel–cadmium (NiCd) and nickel metal hydride (NiMH) batteries, in which the batteries require full discharge cycles. The main drawback of Li-Ion batteries is that they require protection circuits to be built into the batteries.

Lithium polymer (Li-Po) batteries have become a very popular solution for smartphones. Li-Ion and Li-Po batteries have similar characteristics with the key difference being the electrolyte state that is liquid in Li-Ion and polymer gel in Li-Po. The Li-Po batteries can be shaped better due to the polymer gel and thus they are the preferred technology for smartphone and mobile device batteries [2]. The thin form-factor and packaging materials with the current Li-Po implementations allow for a small increase in the energy density; however, the production cost is increased as well.

Battery properties can be characterized by static and dynamic parameters. The former defines the physical characteristics of the battery, such as the nominal voltage and cycle life. These characteristics stem from the electrochemical nature of the battery. The latter, on the other hand, defines the parameters that affect the runtime usage of the battery, most importantly charging and discharging of the battery.

3.2 Static battery parameters

The static battery parameters define the physical characteristics of the battery. These parameters are defined and tested by the manufacturer of the battery. When the chemical construction of the battery is complete, it is programmed with the static information such as the maximum voltage and capacity.

Important battery parameters are [2]:

- Theoretical voltage: the theoretical voltage is determined by the electrode materials.
- Nominal voltage: the normal voltage of the battery measured across the positive and negative terminals of the battery The nominal voltage is what can be achieved in practice whereas the theoretical voltage cannot be achieved due to battery properties.
- Specific energy: the amount of energy in watt-hours (Wh) per unit mass (kg) that the battery can deliver.
- Cut-off voltage: the minimum voltage for the battery that defines the empty state.
- Capacity or nominal capacity: the total ampere-hours available when discharging the battery at a certain discharge current. The discharge current is typically specified as a C-rate from a full state of charge to the cut-off voltage.
- Maximum continuous discharge current: the maximum current draw for the battery.
- Energy or nominal energy: the total watt-hours available when the battery is discharged at a certain current specified by the C-rate from a full charge to the cut-off voltage.
- Cycle life: The number of discharge–charge cycles the battery supports before it fails to meet the specific performance requirements. Typically cycle life is defined as the number of discharge–charge cycles before the battery reaches 80% capacity [4]. The cycle life is specific for charge and discharge conditions. Li-Ion AND Li-Po batteries typically sustain from 400 to 1000 cycles.
- Self-discharge rate: the battery slowly discharges energy even if it has no load because of the electro-chemical reactions. The rate of this energy depletion is called the self-discharge rate. Li-Ion and Li-Po batteries typically have small self-discharge rates.
- Shelf-life: the battery shelf-life is the time that a battery can be stored in an inactive state before the capacity reaches 80%. The battery depletion in the inactive state is due to the loss of active materials.

The discharge current is often expressed as a C-rate that normalizes it against the battery capacity. A C-rate is a measure of the discharge rate relative to the maximum capacity of the battery:

$$I = M \times C_n, \tag{3.1}$$

where I is the discharge current, C denotes the C-rate in ampere-hours, n is the time for which the C-rate is the determined, and M is a constant factor of C.

For example, a C-rate of 1, denoted by 1 C, states that the discharge current will empty the battery in 1 hour. A rate of 0.1 C means that the battery is emptied in 10 hours or 10% discharge per hour. For a 2500 mAh battery a C-rate of 0.5 C means discharging the battery for two hours with a 1250 mA current. Li-Ion/polymer batteries are electronically protected against high discharge currents. Depending on the battery type, the discharge current is limited somewhere between 1 C and 2 C.

3.3 Dynamic battery parameters

Dynamic battery parameters determine the runtime usage, operating environment, and operating condition of the battery. Important variables for describing battery condition include the following:

- State of charge (SOC) gives the battery capacity in percentages.
- State of discharge (SOD), or depth of discharge (DOD), gives the capacity that has been discharged as a percentage of the maximum capacity. A discharge to 80% or more DOD is said to be a deep discharge.
- Terminal voltage gives the voltage between the battery terminals with load. The terminal voltage varies with the current and SOC.
- Open-circuit voltage (OCV) defines the voltage between the battery terminals with no load. OCV depends on SOC and increases with the charge.
- Internal resistance gives the resistance in the battery. Increasing internal resistance reduces battery efficiency as more heat is produced.
- Service life gives the time that a battery can be used in various operating conditions (loads and temperatures).
- State of health (SOH) is a figure that represents the health of a battery. A SOH value of 100% means that the battery's condition is ideal. A smaller value means that the battery performance degraded from the ideal condition defined by battery specification.

3.3.1 Temperature

The battery temperature during discharging has a significant effect on the capacity and the voltage. Temperature affects the internal resistance and increases the chemical activity of the battery. With higher temperatures, the internal resistance decreases and the discharge voltage increases as well as the capacity and power. However, high temperatures increase the self-discharge effect resulting in reduced capacity. Lower temperatures, on the other hand, result in a decrease in capacity due to slowing of the chemical activity.

The internal properties of the Li-Ion battery have an Arrhenius dependence on temperature. The relationship is given by Eq. (3.2), in which Φ represents the conductivity of the electrolyte. Φ_{ref} and T_{ref} denote the values at a reference temperature. $E_a(\Phi)$ denotes the activation energy of the evolution process and the sensitivity of Φ to temperature [5].

$$\Phi = \Phi_{ref} \exp\left[\frac{E_a(\Phi)}{R}\left(\frac{1}{T_{ref}} - \frac{1}{T}\right)\right] \tag{3.2}$$

3.3.2 Battery age and service life

Batteries are based on chemical reactions that deteriorate over time. The chemical properties as well as the operating and storage temperatures and the discharge behavior affect the performance and capacity of the battery. Low temperatures decrease the self-discharge as well as discharge behavior.

A rechargeable battery sustains a certain number of charge–discharge cycles before the battery is exhausted. An internal cycle counter within the smart battery keeps track of the cycle count. The cycle count is incremented when 70–80% of the capacity has been used from a full charge. Intermittent charging is not taken into account.

The service life gives the length of time that a battery is expected to operate. In an ideal setting, the service life of a battery would simply be the capacity of the battery divided by the load in terms of current [6]. This simple approach does not work in realistic situations, because it does not capture the various nonlinear effects of the battery, including the internal resistance, temperature, and age.

The service life of a battery can be approximated in a given discharge setting. Peukert's equation is frequently used to assess the service life under constant load [2]:

$$n \log I + \log t = \log C, \tag{3.3}$$

where I is the discharge rate (the load), t is the time, and n is the slope of the line. It should be noted that the curve is linear on a log–log plot with the exception of the ends, where there are nonlinear artefacts due to battery limitations with high rates and self-discharge at low rates.

Peukert's equation has been found to be reasonably good for predicting service times for constant loads. The model can be extended for non-constant loads by replacing the constant, I, by a function representing the discharge distribution. In addition to Peukert's equation, a number of modeling techniques have been proposed, such as analytical diffusion models, the Kinetic Battery Model, and various stochastic models [6].

3.3.3 State of health

Battery diagnostics is important for ensuring proper battery operation. The performance of a battery varies with time, charging and discharging patterns, and the operating environment. As the battery ages, the performance reduces because of changes in the battery chemistry [7]. In addition, many different harmful events can result in battery damage, such as over-voltage, over-charging, and over-depleting the battery.

The SOH is a figure that represents the health of a battery. A SOH value of 100% means that the battery's condition is ideal when compared to the battery's specification. The SOH value of a battery is typically maximum when it is manufactured, but this value decreases over time due to usage. The SOH value is useful in deciding when to replace a battery as well as in determining the battery's service life.

A battery management system within the smart battery evaluates the health of the battery and determines the SOH value. The system uses a combination of battery parameters to make this estimation. These parameters include the capacity, voltage, self-discharge, number of cycles, and internal resistance.

The two main methods for determining the SOH for Li-Ion batteries are the impedance- and capacitance-based techniques [8]. In the former technique the impedance is measured for a new battery, R_0, is the initial measurement, and then the measurement is repeated to capture battery aging effects with R_i for the ith measurement. The SOH is then given by the following equation in percentages:

$$SOH = \frac{R_i}{R_0} \times 100. \tag{3.4}$$

In the latter technique, the SOH is determined based on capacitance instead of impedance. In this case, C_0 is the initial battery capacitance and C_i is the ith capacitance measurement:

$$SOH = \frac{C_i}{C_0} \times 100. \tag{3.5}$$

The full discharge test estimates the SOH by first fully discharging the battery and measuring the charge delivered by the battery, and then comparing this value to the charge delivered when the battery was new [7].

For a new, fully charged battery both the SOH and SOC are at 100%. When the battery is discharged, the SOC gives the percentage of the remaining capacity and the SOH describes the full charge that can be obtained from the battery.

3.4 Smart batteries

Today's smartphone batteries have hardware interfaces for reading both static and dynamic parameters. Batteries that have such interfaces are called smart batteries. A smart battery is a battery that has special hardware for monitoring its internal state and environment. A battery typically has five terminals: namely the plus and minus, The heart of a smart battery is the fuel gauge IC. The fuel gauge typically measures the battery voltage, temperature, and sometimes current. Based on the measurements, the chip estimates the SOC and SOD. For example, the Maxim MAX17040 uses the voltage and temperature for capacity estimation based on its internal battery model [9]. Another technique is to use an internal current sense resistor to estimate the amount of charge discharged from the battery [10].

clock, data, and a safety signal. The safety signal typically consists of the operating temperature of the battery.

The monitoring hardware typically estimates the voltage, current, and temperature of the battery. These basic variables are then used to estimate the SOC and SOH. This information can be accessed over a low-power serial bus, such as the SMBus (System Management Bus). A smartphone or a smart battery charger can access this information over the bus and make decisions regarding the battery.

The smart battery exposes registers that store parameter values. These registers are updated periodically with new values. The smartphone OS and its drivers can access these values through the serial bus. The access cost in terms of energy consumption is negligible [9].

The smart battery interfaces vary from device to device. For example, some smart battery units are instrumented with the selection of a suitable sense resistor. A higher resistance value will result in a smaller error but higher power dissipation. Therefore the sense resistor size is typically chosen to be as small as possible in smartphone designs [10].

Table 3.1 presents the update rate, sampling rate, resolution, and error for three frequently used smart battery units. The update rates of the interfaces are limited and are typically well below 4 Hz. The resolution and error vary between units. The instant battery readings have been demonstrated to have a high error [9].

The limitations of the smart battery, namely update rate and accuracy, can be partly addressed in software. One solution is to average across several smart battery readings. This increases the accuracy at the price of a reduced update rate [9]. The second solution is to detect the battery update rate and synchronize measurement operations on the device [11, 12]. We discuss these techniques in Chapter 10.

The smartphone and the charger typically access the following information from the smart battery interface:

1. Static parameters for voltage, current, and temperature.
2. Dynamic voltage, current, power, and temperature. Estimation of the SOC and SOH
3. Battery cycle count.
4. Battery design capacity.
5. Remaining time to full charge.

Table 3.1. Example smart battery interfaces

Battery unit	Update rate	Sampling rate	Resolution	Error
Maxim DS2784 (Google Nexus One)	0.28 Hz	18.6 kHz	104 μ A	±1%.
Maxim MAX17040 (Samsung Galaxy Nexus)	2 Hz	32 kHz internal clock	Voltage: 1.25 mV SOC: 1/256%.	±12.5 mV
TI BQ27200 (Nokia N900)	0.39 Hz	100 kHz internal clock	2.7 mv	±25 mV, Current gain variablility ±0.5%

6. Remaining run time.
7. Manufacturer and product information.

3.5 Chargers

The energy supply of a rechargeable battery is replenished in the charging process that takes charge into the battery. Charging is implemented by a charging device, the battery charger. The charger needs to monitor the battery during charging to prevent overload and thus safety is one key design goal for chargers. In addition to safety, the charging process needs to be efficient and fast. The charger has the following key functions [2]:

- Taking charge into the battery in the charging process.
- Monitoring and stabilizing the charging rate to optimize the charging process.
- Terminating charging when the maximum charge has been taken to the battery.

3.5.1 Charging process

Li-Po and Li-Ion batteries are typically charged with the constant current (CC) or constant current/constant voltage (CC/CV) schemes. In the CC method, the charger varies the voltage to maintain a constant flow of current to the battery. The charging process ends when the voltage reaches the full charge level. In the constant voltage (CV) method, the charger is a DC power supply for the battery. The standard method to charge a Li-Ion or Li-Po battery is to combine the CC and CV methods [3]. In the CC/CV charging method, the charging current needs to be near the C-rate of the battery. This means that the charger must be capable of safely giving current at the C-rate. A lower charging rate is also possible, but will result in a longer charging time.

The CC/CV method consists of four stages, as illustrated in Figure 3.2 [3, 4]:

- Stage 1: Voltage rises with constant current to the maximum voltage. The charge rate is typically between 0.5 C and 1 C. This stage ends when the voltage peaks after

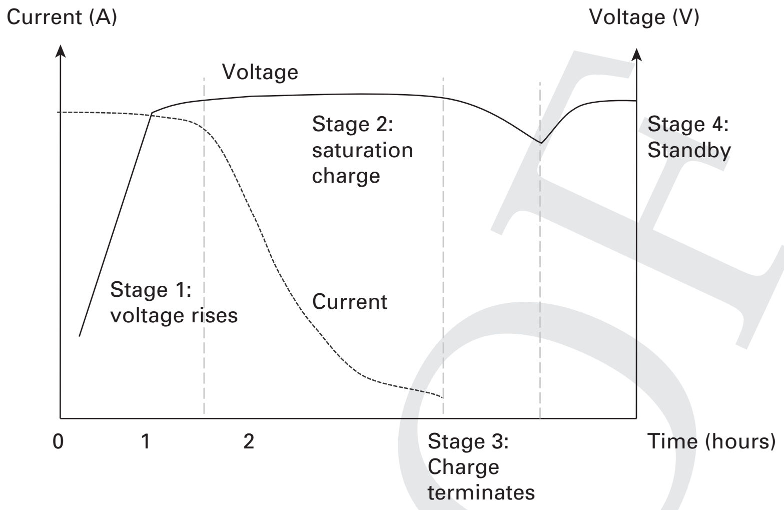


Figure 3.2 Four charging stages

which the next stage starts. The SOC is about 60–85% at this point and the next stage slowly increases this toward the maximum.

- Stage 2: The battery is saturated in this stage by applying a constant voltage (typically 4.2 V) causing the current to drop. The current waveform resembles an exponential decay.
- Stage 3: Charging is terminated when the current drops below 3% of the rated current.
- Stage 4: If the device is in the charger, a topping charge may be applied to compensate for the self-discharge.

3.5.2 Charger circuit

Figure 3.3 presents an overview of the battery charger circuit. The charger measures the voltage and follows the above stages 1 to 4. First the voltage rises with constant current and then the battery is saturated. The battery is fully charged when the voltage reaches 4.2 V and the drop across the resistor is negligible. At this point there is only a very small current going into the battery [13].

The charger measures the voltage, V_B, at the terminals of the battery. This voltage is the sum of the voltage drop, V_R, across the internal resistor (equivalent series resistance) and the battery cell voltage, V_C. The following equations describe the voltage of the battery during charging [13]:

$$V_R = I_C \times R, \tag{3.6}$$

$$V_B = V_C \times V_R, \tag{3.7}$$

$$V_B = V_C + (I_C \times R). \tag{3.8}$$

Table 3.2. Summary of USB charging specifications

Standard	Voltage	Maximum current
USB 1.0 and 2.0	5 V	500 mA
USB 3.0	5 V	900 mA
USB Battery Charging specification 1.0 2007	5 V	1500 mA
USB Battery Charging specification 1.2 2010	5 V	5000 mA

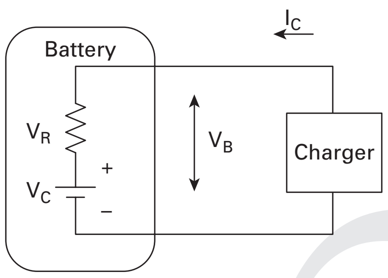


Figure 3.3 Charger circuit

In addition to CC, CV, and CC/CV, pulsed chargers charge the battery in pulses. The rate is controlled by adjusting the length of the pulses. A rest period between the pulses is used to allow the battery to stabilize.

3.5.3 USB charging

The mobile phone industry has agreed on a standard for charging mobile phones with a micro USB connector. These chargers use 5 V for charging and the smart controller is inside the phone rather than in the charger [14]. The controller monitors the voltage and other parameters during the charging process. The USB socket consists of four pins: two inside pins for data and two outside pins provide the 5 V power supply. The current provided by the USB power supply depends on the type of USB port. The USB Battery Charging Specification of 2007 defines three different port types: a downstream port, a charging downstream port, and a charging port. The last one is used by USB wall chargers. With the older USB 1.0 and 2.0 standards, the downstream port can deliver up to 500 mA and with USB 3.0 up to 900 mA. The charging downstream and charging ports can deliver up to 1500 mA. The maximum current for the charging downstream port was increased to 5 A in the 1.2 version (2010) of the specification. There are several USB extensions, such as the USB Power Deliverable specification, that increase the charging power of the basic connector. Table 3.2 gives a summary of the USB charging specifications.

A study of over 4000 Android users indicated that AC-based wall chargers are the popular choice for overall phone charging with the exception of short charges where non-AC USB is much more popular [15]. USB, as reported by the Android API, was used 39% of the time for charging mobile phones.

The USB interface is frequently used for charging; however, older versions of it could only support currents up to 1.5 A. A 2500 mAh battery can be charged in two hours with a charge rate of C/2 = 1250 mA. The same battery needs four hours to charge with C/4 = 625 mA. This simple example illustrates the impact of the charging rate; however, in practice this analysis does not apply to Li-Ion and Li-Po batteries, because the C-rate changes during the charging process.

3.5.4 Wireless charging

In addition to USB, wireless charging has also become more popular. The main components of a wireless charging system are a power transmitter and a receiver. A stable power source, typically an AC line, powers the transmitter. The receiver, on the other hand, is located on a mobile device powered by a battery. The transmitter then sends power wirelessly to the receiver.

This technique is based on charging that uses an electromagnetic field to transfer energy to the smartphone's battery. Energy is sent through an inductive coupling within a charging base and the smartphone to transfer energy from the base to the mobile device.

Standards such as the Wireless Power Consortium (WPC)[1] established in 2008 and the Alliance for Wireless Power[2] established in 2012 are now being implemented in commercial products. WPC's Qi is supported by many Android and Windows Phone smartphone manufacturers and it supports the transfer of power over distances of up to 4 cm. This system is based on a power transmission pad and a receiver in a portable device.

3.5.5 Charging behavior and cycle age

In addition to the charging current, the charging behavior of the mobile phone user also plays an important role for the longevity of the battery. Mobile phones are typically recharged when the battery SOC reaches an average low level of 30% of the battery capacity [15]. This charging behavior is beneficial for Li-Ion and Li-Po batteries used in the phones and the number of cycles is increased by a factor of 5 to 10 [4].

[1] `http://www.wirelesspowerconsortium.com` accessed January 6, 2014.
[2] `http://www.rezence.com` accessed January 6, 2014.

To increase the cycle life a battery [4]:

- Recharge the battery when it reaches below 30% SOC. This extends by the cycle life a factor from 5 to 10.
- Avoid full discharge cycles and charging to 100%.
- Select a charger with a minimum charge current termination, as this also helps to avoid charging to full capacity thus increasing the cycle life.
- Avoid extreme battery temperatures that are harmful for the internal chemistry of the battery.

3.6 Discharge current

In an ideal case, a battery is exhausted by discharging it at the theoretical voltage defined by the electrode materials of the battery. The battery is discharged until its capacity is fully used and the voltage drops to zero. In practice, when a battery is discharged, its voltage is below the theoretical voltage due to the properties of the battery [2]. This difference is due to the internal resistance and processes, such as the polarization of active materials and the accumulation of discharge products, that occurs during discharge. The energy stored by a battery is not in practice fully used due to the average discharge voltage being lower than the theoretical voltage, and because the battery is not fully discharged to zero volts.

The two key nonlinear battery effects are [6]:

- Rate capacity effect, in which the voltage drops faster for high discharge currents. This means that an increase in the discharging current results in a decrease in the discharging efficiency.
- Recovery effect, in which the battery recovers during idle periods resulting in an increase in voltage. This is visible in a sawtooth pattern in the discharge curve.

Figure 3.4 illustrates the discharge curve of the Li-Ion battery. The curve is monotonically decreasing. The energy capacity and discharge curve of the battery are affected by the discharge current, temperature, battery age, and the nonlinear effects. As the current is increased both the voltage and capacity are reduced.

The discharge process can be characterized through three key discharge modes: constant resistance, constant current, and constant power. In the first, the external resistance is fixed and the current and voltage vary with time following the equation $V = IR$. The current decreases in proportion to the decrease in battery voltage due to internal resistance. In the second, the current remains constant and the C-rate describes the discharging. In the third, the power, $P = IV$, is constant resulting in increasing current to compensate for the decreasing voltage due to internal resistance. Comparing these three modes, the voltage drops fastest for the constant power mode resulting in the smallest

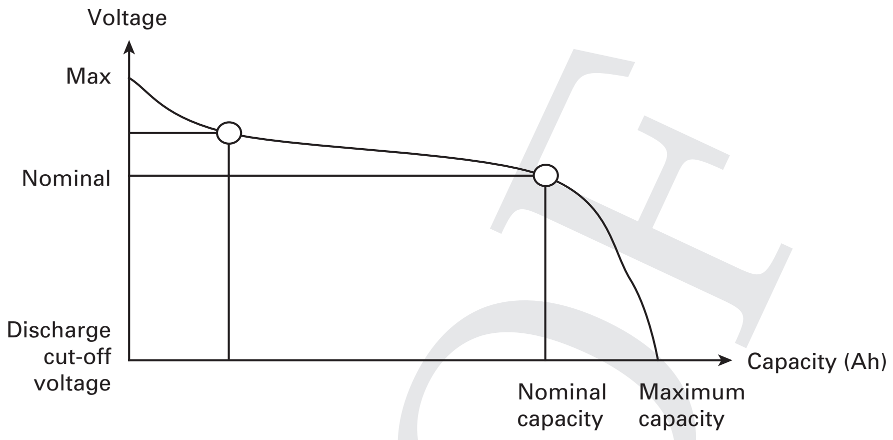


Figure 3.4 The discharge curve with voltage and capacity

charge capacity. Discharging at a constant external resistance gives the largest charge capacity.

In addition, the discharge type plays an important role. By discharge type we mean the temporal discharge of the battery, for example whether it is continuous or intermittent. If the discharge is intermittent, the battery will be in idle mode after a continuous discharge allowing it to recover some battery voltage. This results in a sawtooth discharge in which the voltage recovers during idle periods and decreases during discharges.

Smartphones operate with many workloads, so the discharge load changes thus affect the battery voltage and the service time. Since the voltage recovers in the sawtooth pattern, the overall service time is improved, resulting in a longer operating time. The smart battery typically has a capacitor in parallel that provides instantaneous high current. This helps the system to cope with intermittent discharge, such as the GSM pulse mode communication. For GSM pulse mode, the high current pulse is of the order of 1 A and the low current pulse is 0.1 A. A basic smartphone typically operates in constant power mode with an average current around 300 mA and maximum current peak at 1 A [10].

3.6.1 Discharge efficiency

The discharge efficiency is defined as the ratio of the battery's output current to the degradation rate of its stored charge. As mentioned above, the rate capacity effect describes the phenomenon that an increase of battery discharging current leads to a decrease in the discharge efficiency. It is evaluated by an empirical equation, called Peukert's formula, as follows [2]:

$$T = \frac{C}{I^n}, \tag{3.9}$$

where C is the theoretical capacity (in Ah, equal to capacity at one ampere), I is the current, T is the battery life, and n is the Peukert number for the battery.

The equation is often formulated to a known capacity and discharge rate:

$$T = H\left(\frac{C}{IH}\right)^{k}, \tag{3.10}$$

in which H is the rated discharge time, C is the rated capacity for that discharge rate, I is the actual discharge current, k is the Peukert constant, and T is the discharge time of the battery.

If the Peukert's exponent is known, it is possible to determine the battery capacity at any given C-rate:

$$C_{eff} = C\left(\frac{C}{IH}\right)^{k-1}. \tag{3.11}$$

3.6.2 State of charge (SOC) and state of discharge (SOD)

The state of charge (SOC) of a battery gives the level of the charge in the battery in percentages, in which 0% denotes an empty battery and 100% a full battery. The SOC is typically determined by

$$SOC = (C_n - Q_b)/C_n, \tag{3.12}$$

where C_n is the nominal capacity and Q_b is the net discharge.

As discussed in this chapter, the fuel gauge chip has an internal algorithm for determining the SOC value. In addition, the battery driver software running on the smartphone may apply corrections to the value given by the chip. Some fuel gauge chips measure voltage, current, and temperature and allow the driver to obtain this information. For example, the Maxim MAX17047 chip has registers for current measurement values that can be read over the I^2C bus.

The state of discharge (SOD), or depth of discharge (DOD), is an alternate form of SOC, being its inverse. Thus, 100% SOD denotes an empty battery and 0% denotes a full battery.

The discharge rate is an important measure of how fast the battery level, as measured by the SOD, is being depleted per second. The discharge rate in percentages per second is given by the following equation:

$$dr = \frac{\Delta SOD}{\Delta t}, \tag{3.13}$$

where ΔSOD is the difference in the battery level during Δt.

3.7 State of charge (SOC) measurement

The SOC is typically measured by indirect means with the exception of batteries that offer access to the liquid electrolyte. In this case the SOC value can be determined based on the pH level of the electrolyte. For other types of battery, there are in principle three estimation techniques frequently used by commercial products:

- Load voltage: this technique is suitable for applications that have constant load.
- Coulomb or ampere counting: this technique estimates the difference between the accumulated coulombs from the beginning of the discharge and the known full-charge capacity. The technique has limited accuracy for variable loads, because it does not take the nonlinear discharge effect into account.
- Internal resistance: this technique determines the battery state by measuring the frequency response of the battery. This typically requires a testing period.

The load voltage and coulomb counting techniques can be combined for an overall battery estimation technique that takes into account cycle counts and temperature. In the following, we briefly outline the load voltage and coulomb counting techniques, and their combination.

3.7.1 Load voltage

The load voltage technique is based on converting the battery voltage to the SOC with a known discharge curve of the battery. Due to the internal properties of the battery, the voltage is affected by the battery current and the temperature. A charge or discharge changes the voltage so that it no longer represents the true state of charge. The voltage reading can thus be made more accurate by applying a correction term that is proportional to the current. In addition, the result can be further improved by correcting the OCV to take the temperature into account. For accurate results, this technique may require considerable time for a battery to attain equilibrium.

The SOC estimation typically relies on the OCV of a Li-Ion battery, more formally $SOC = f(OCV)$. This is complicated by the observation that the OCV cannot be directly measured at the battery terminals. As a consequence, the SOC is calculated with only the battery voltage and current at the terminals.

3.7.2 Coulomb counting

In coulomb counting the idea is to measure the current going in and out of the battery. If we charge a battery for two hours at one ampere, we should have the same energy available when discharging the battery with the exception of the energy that was lost due to inefficiencies in the battery and the storage loss. The energy loss results in a measurement error for this technique.

The coulomb counting technique determines the SOC by measuring the battery current and integrating it in time. The main limitation of the approach is that full charge should happen regularly, and when it does not occur often enough the error term increases. The technique needs to be recalibrated regularly, for example by setting the SOC to full charge when charging has been completed.

The SOC can be calculated as

$$SOC = SOC_{init} - \int (I_{bat}/C_{usable})\,\mathrm{d}t, \tag{3.14}$$

where C_{usable} depends on capacity fading [16], that is the irreversible loss of the capacity due to temperature, time, and cycle number. Calendar life relates to losses when the battery is inactive. Usable battery capacity is then

$$C_{usable} = C_{initial} \times CCF, \tag{3.15}$$

where $CCF = 1 - ($ Calendar life losses $+$ Cycle life losses$)$.

3.7.3 Combined techniques

The load voltage and coulomb counting techniques are frequently combined to overcome their specific limitations. Their combination has been observed to work well with Li-Ion batteries [17, 18]. The battery current is integrated with the coulomb counting to obtain the relative charge in and out of the battery. The voltage is measured to calibrate the SOC. Many of the proposed combined estimation techniques rely on an electrical model for the state of charge. A Kalman filter is then used to predict over-voltage and improve the estimation accuracy [17].

A combined SOC can be obtained with the following equation [19]:

$$SOC = \alpha SOC_c + (1\alpha)SOC_v, \tag{3.16}$$

where SOC_c is the coulomb-counting based SOC and SOC_v is the estimated OCV of the battery. The parameter $\alpha \in [0, 1]$ is a weighting factor.

3.8 Open-circuit voltage (OCV) models

For proper power management, we need reliable information on the power source and its condition (remaining capacity, maximum rated capacity), and the power characteristics with voltage, current, and SOC. This information is typically provided by the smart battery interface; however, as we have observed in this chapter, the information provided by the smart battery interface is typically coarse grained and has limited update rates. In addition, battery parameters change as a function of the discharge rate and temperature, and they are affected by factors such as self-discharge and battery age. Smartphones have complex discharge profiles making parameter estimation more challenging.

Assuming that we obtain a power supply with unknown properties, there are two basic measurements that can be carried out to probe the properties of this black box. First, it is possible to measure the OCV across the terminals without any load. This measurement can be done with a voltmeter. This OCV denotes the highest voltage that can be obtained from the power supply. When a load is applied to the battery, the voltage across the positive and negative terminals decreases due to the internal resistance. The second basic measurement is to determine the output current that can be determined by short-circuiting the output terminals.

Equivalent circuit models are used to capture the properties and characteristics of the battery. A battery circuit model typically consists of resistors, capacitors, and voltage sources. An ideal voltage source typically represents the OCV and the rest of the circuit represents the internal resistance and dynamic properties of the battery. The SOC can be determined based on the measured OCV using a lookup table. The OCV voltage is needed for estimating the SOC, because the SOC cannot be determined directly with current sensing technologies.

3.8.1 The Rint model

Many OCV models have been proposed in literature. The simplest one is the Rint model outlined in Figure 3.5 that has an ideal voltage source that defines the OCV. The terminal voltage, V, can be obtained from the open-circuit measurement. The resistance, R_0, can be obtained with an additional measurement with a fully charged battery with load at the terminal. The model does not take into account the dynamic properties of the battery, such as the electrolyte concentration and internal impedance.

The OCV and the resistance are both functions of the SOC and temperature. This model does not take into account the transient behavior of the Li-Ion cells and thus it is not useful for estimating the SOC in dynamic operation [20].

The Rint model is based on an ideal voltage source, OCV, that defines the OCV, the terminal voltage V, the resistance R, and the current I. This model is given by the following equation:

$$V = OCV - I \times R. \tag{3.17}$$

3.8.2 The Thevenin model

To address the limitations of the Rint model, many more elaborate models have been proposed [20, 18, 17]. The Thevenin model is frequently used to extend the Rint model to take into account the dynamic properties of the battery. This model is illustrated in Figure 3.6 and it includes the OCV and the internal resistances and capacitances. The internal resistance consists of the ohmic resistance, R, and the polarization resistance, r. The equivalent capacitance, C, describes the transient response during charging and discharging. V_c is the voltage across C. The electrical behavior of the model is described

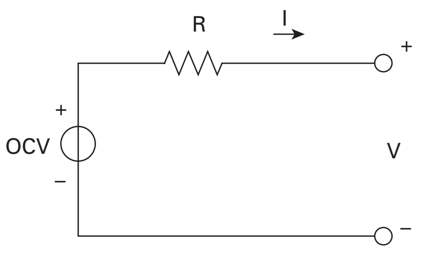


Figure 3.5 The basic Rint model

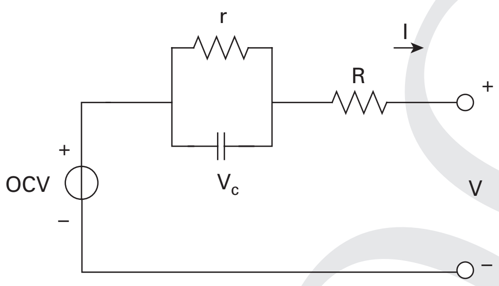


Figure 3.6 The basic Thevenin model

by the equation:

$$V = OCV - V_c - I \times R. \tag{3.18}$$

3.8.3 OCV–SOC relationship

Figure 3.7 illustrates the relationship between the battery voltage and the SOC. Energy is lost in the charging and discharging process due to chemical hysteresis [3]. The actual battery voltage depends on the SOC and when the battery is in an equilibrium state it will be between the charge and discharge curves.

The OCV depends on the current SOC level; however, the OCV deviates from its true value due to hysteresis. The observed voltage for a given SOC is greater than the OCV after charging, and it is less than the OCV of the same SOC after discharging.

Figure 3.8 presents an example discharge scenario, in which part of the battery capacity has been used. The difference between the actual discharge curve and the OCV curve is explained by the load $I \times R$ and the dynamic properties of the battery. The expected future discharge curve will meet the cutoff point before the OCV curve due to the load and the dynamic battery properties [21].

It has been shown that there is a time-independent bijection between the OCV and the SOC [19]. The OCV–SOC curve is nonlinear, but the SOC variations are expected to be small with ordinary current rates. The OCV–SOC curve can be mapped with a line

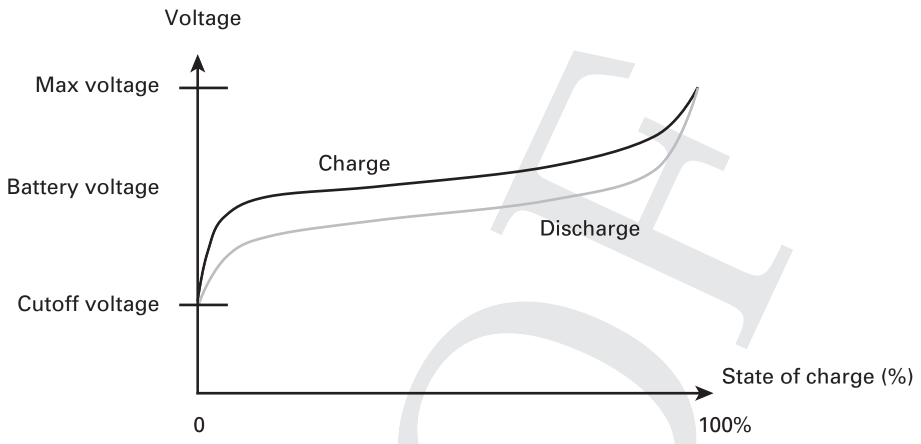


Figure 3.7 Relationship between the battery voltage and the SOC

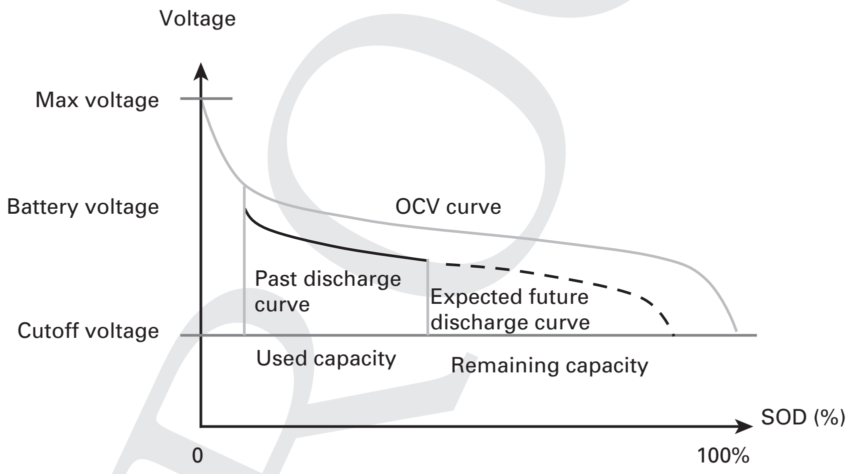


Figure 3.8 Relationship between the OCV and the SOD

with slope b_1 and OCV intersection b_0 [22]:

$$OCV = f(SOC) = b_1 \times SOC + b_0. \tag{3.19}$$

The parameter b_0 is the battery terminal voltage when the SOC is zero and b_1 is determined when b_0 and OCV are known when SOC is at 100%. The challenge with this approach is that the OCV measurement requires the battery to be disconnected from the load. There are several proposals that relax this requirement and allow OCV estimation under load conditions. We discuss this estimation in the following subsections.

3.8.4 Lookup tables: from voltage to SOD

If the rated battery energy capacity, E, and the discharge curve of the battery are known, then the energy draw can be estimated based on the following equation [23]:

$$P \times (t_1 - t_2) = E \times (SOD(V_1) - SOD(V_2)), \tag{3.20}$$

where P is the average power in the time interval $[t_1, t_2]$ and $SOD(V_i)$ is the battery state of discharge (the inverse of SOC) level at voltage V_i. Given the voltage measurements, the SOD values can be obtained using a battery-specific lookup table. The lookup table characterizes a battery from the fully charged state to the fully discharged state using a constant discharge current.

PowerBooter is an example of a power profiler that does not need any external measurement equipment, but can perform online power modeling of a smartphone and its components. The technique uses on-device voltage sensors and battery discharge curves to estimate power consumption [23]. PowerBooter and its application, PowerTutor, are investigated in more detail in Chapter 10.

3.8.5 Obtaining current from voltage

Assuming the Thevenin model, the battery voltage, V, at a certain point in time can be determined from

$$V = OCV - V_c - R_b \times I, \tag{3.21}$$

where OCV is the open-circuit voltage of the remaining capacity of battery, V_c is the voltage drop on the capacitance, R_b is one of the two internal resistors, and I is the discharge current. When the current changes, the product of $R_b \times I$ is affected. This voltage drop is illustrated in Figure 3.9. The voltage drop is called the internal resistance effect. After the fast change, the voltage slowly decreases because of the current draw due to discharging.

Equation (3.21) does not take the operating temperature into account. This can be introduced with a new term, ΔT [16]:

$$V = OCV - V_c - R_b \times I - \Delta T. \tag{3.22}$$

The instantaneous voltage change, $R_b \times \Delta I$, has been called the V-edge that is linearly proportional the change of current. Decomposing the delta, we have $V_{edge} = R_b \times I - R_b \times I_0$. Thus the relationship

$$I = \frac{1}{R_b} \times V_{edge} + I_0, \tag{3.23}$$

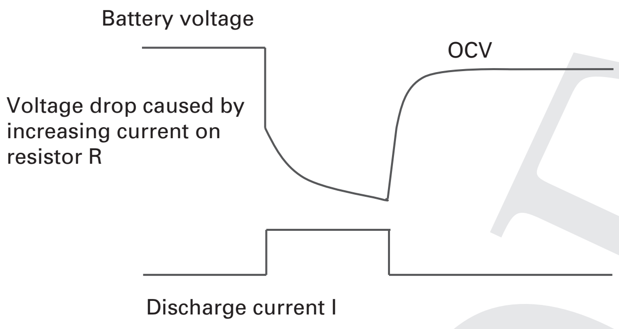


Figure 3.9 The voltage drop in OCV with increasing current

can be used to determine the current value. A calibrated battery model is needed to use this observation for obtaining the current [12].

The V-edge system builds an online power model an a smartphone by measuring voltage dynamics. V-edge estimates the power draw based on the instantaneous voltage that is calibrated once with SOD metering. The model is faster to create and update than SOD-only approaches. Online power profilers are examined in more detail in Chapter 10.

3.9 Summary

In this chapter, we have examined the characteristics and modeling of smartphone batteries. Battery models are vital to understand the remaining operating time of smartphones and other devices. Battery capacity estimation is made difficult by the dynamic operating environment. Furthermore, it is difficult to obtain reliable information regarding the battery SOC and condition.

The key observations are the following:

- Battery properties can be characterized by static and dynamic parameters. The static parameters define the physical characteristics of the battery, such as the nominal voltage and cycle life. The dynamic parameters define the parameters that affect the runtime usage of the battery, most importantly the charging and discharging of the battery.
- Today's smartphone batteries have hardware interfaces for reading both static and dynamic parameters. A smart battery is a battery that has special hardware for monitoring its internal state and environment. The heart of a smart battery is the fuel gauge IC.
- Power management requires reliable information about the power source and its condition and the power characteristics (voltage, current, SOC).

- Battery capacity measurement is challenging, because the capacity changes as a function of the discharge rate and temperature. In addition, factors such as self-discharge and age need to be taken into account.
- Three SOC estimation techniques are frequently used in commercial products: load voltage, coulomb or ampere counting, and internal resistance. The techniques require calibration.

References

[1] H. Chen, T. N. Cong, W. Yang, C. Tan, Y. Li, and Y. Ding, "Progress in electrical energy storage system: A critical review," *Progress in Natural Science*, vol. 19, no. 3, pp. 291–312, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.pnsc.2008.07.014

[2] D. Linden and T. B. Reddy, *Handbook of Batteries*. 3rd ed. McGraw-Hill Professional, 2001.

[3] *Battery University Website*, Jul. 2013. [Online]. Available: http://www.batteryuniversity.com.

[4] F. Hoffart, "Extend battery life with proper charging, discharging," *EE Times Asia*, Jun. 2008. [Online]. Available: http://www.eetasia.com/STATIC/PDF/200806/EEOL_2008JUN16_POW_TA_01.pdf.

[5] P. Rong and M. Pedram, "An analytical model for predicting the remaining battery capacity of lithium-ion batteries," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 5, pp. 441–451, 2006.

[6] M. Jongerden and B. Haverkort, "Battery modeling," Tech. Rep. TR-CTI, January 2008. [Online]. Available: http://doc.utwente.nl/64556/

[7] M. Coleman, W. Hurley, and C. K. Lee, "An improved battery characterization method using a two-pulse load test," *IEEE Trans. Energy Convers.*, vol. 23, no. 2, pp. 708–713, 2008.

[8] D. Liu, J. Pang, J. Zhou, Y. Peng, and M. Pecht, "Prognostics for state of health estimation of lithium-ion batteries based on combination gaussian process functional regression," *Microelectronics Reliability*, vol. 53, no. 6, pp. 832–839, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026271413000747

[9] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proc. 9th Int. Conf. Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2011, pp. 335–348.

[10] M. Yu and J. Qian, *How to Use bq27505 in a Smart Phone System*, Texas Instruments, Aug. 2009, Application Report.

[11] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, "Devscope: a nonintrusive and online power analysis tool for smartphone hardware components," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2012, pp. 353–362. [Online]. Available: http://doi.acm.org/10.1145/2380445.2380502

[12] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: fast self-constructive power modeling of smartphones based on battery voltage dynamics," in *Proc. 10th USENIX Conf. Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2013, pp. 43–56.

[13] C. Simpson, *Battery Charging*, National Semiconductor, 2011, Literature Number: SNVA557.

[14] *USB Battery Charging 1.2 Compliance Plan Revision 1.0*, 2011.

[15] D. Ferreira, A. K. Dey, and V. Kostakos, "Understanding human-smartphone concerns: A study of battery life," in *Pervasive*, 2011.

[16] O. Erdinc, B. Vural, and M. Uzunoglu, “A dynamic lithium-ion battery model considering the effects of temperature and capacity fading,” in *2009 Int. Conf. on Clean Electrical Power* 2009, pp. 383–386.

[17] C. Zhang, J. Jiang, W. Zhang, and S. M. Sharkh, “Estimation of state of charge of lithium-ion batteries used in HEV using robust extended kalman filtering,” *Energies*, vol. 5, no. 4, pp. 1098–1115, 2012.

[18] S. Lee, J. Kim, J. Lee, and B. Cho, “State-of-charge and capacity estimation of lithium-ion battery using a new open-circuit voltage versus state-of-charge,” *Journal of Power Sources*, vol. 185, no. 2, pp. 1367–1373, 2008.

[19] B. Xiao, Y. Shi, and L. He, “A universal state-of-charge algorithm for batteries,” in *Proc. 47th Design Automation Conf.* New York, NY, USA: ACM, 2010, pp. 687–692. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837449

[20] H. He, R. Xiong, and J. Fan, “Evaluation of lithium-ion battery equivalent circuit models for state of charge estimation by an experimental approach,” *Energies*, vol. 4, no. 4, pp. 582–598, 2011. [Online]. Available: http://www.mdpi.com/1996-1073/4/4/582

[21] S. Wen, “Design Fuel Gauging for Multicell Li-Lion Battery Pack (Part 2),” *EE Times Asia*, Mar. 2008. [Online]. Available: http://www.eetasia.com/articleLogin.do?artId=8800508929

[22] S. Pang, J. Farrell, J. Du, and M. Barth, “Battery state-of-charge estimation,” in *American Control Conf.*, vol. 2, 2001, pp. 1644–1649.

[23] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, “Accurate online power estimation and automatic battery behavior based power model generation for smartphones,” in *Proc. Eighth IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2010, pp. 105–114.

4 Energy measurement

To improve the energy efficiency of a smartphone, a reliable way to measure the energy consumption of the phone is required. A reliable energy measurement method gives consistent results under the same usage scenario, and responds quickly to variations in energy use. Any energy-efficiency improvements, and energy performance with different workloads can then be measured with the energy-measurement method.

This chapter discusses energy measurement. We discuss the nature of energy and how it must be measured, and the case of the smartphone and its subsystems as the object of study. We explain direct hardware measurement of power, profile-based software power estimation, building such profiles from hardware measurements, and conclude with a hands-on example.

4.1 Hardware- and software-based energy measurement

Draining energy from a battery is a continuous chemical process. The only way to be sure how much energy is being consumed is to measure it constantly as it is being drained. However, doing this affects the use of energy in different ways depending on the measurement method used.

There are two main types of energy measurement: hardware-based and software-based. Hardware-based measurement uses physical tools like power meters that are wired to the smartphone hardware and/or the battery interfaces. Software-based measurement runs on the smartphone energy-profiling software that records the readings of platform-specific built-in indicators of energy use. Note that the software used for measurement also drains the smartphone battery.

In hardware-based measurement, the measurement tools are not powered by smartphone batteries, and therefore do not affect the energy drain rate of the smartphone. Many measurement tools also consist of a DC power supply that can be used for powering the smartphone. An external DC power supply is used to eliminate the variance of supply voltage caused by the chemical characteristics of batteries. The sampling rate of hardware-based power measurement can reach 5000 Hz, depending on the measurement tools used.

Energy-profiling software used in software-based power measurement reads battery statistics and other real-time battery information, such as the level of the remained battery capacity, the voltage/current drained from the battery, and temperature, through

APIs provided by mobile OSs. They can also depend on other information from the mobile OS, such as the system call use, and which features are used, such as GPS or Wi-Fi. This software can sample battery information at intervals dictated by the operating systems. The sampling frequency ranges from one sample in minutes up to several Hz, typically much lower than with hardware power measurement devices. The primary problem with increasing the sampling frequency is that taking a sample drains energy and affects the results of energy measurement. The higher the sampling frequency, the greater the error of energy measurement.

When measuring new systems with either software or hardware, a calibration step is recommended. When using a fixed-voltage power source, calibration is not needed. With the phone battery and a hardware measurement device, accuracy depends on the size of the resistor used. For an example, see Worked examples at the end of this chapter. To calibrate software tools, we can measure energy consumption with the hardware tool and then with the software tool, and adjust the software's internal model and correction factors accordingly.

When measuring a specific hardware component's energy impact, we can first measure the base energy drain of the device without that component, and use that as a baseline. It can then be subtracted from the measured energy use with the component enabled, giving an indication of the component's energy impact.

4.2 Measuring smartphone subsystems

Modern smartphones are built with many power-hungry hardware components, such as multicore processors, large and bright screens, powerful communication chips, and various sensors. In addition, a significant amount of energy is consumed by the software processes that run in the background. Examples include the processes used for synchronizing calendar events, emails and application data; staying connected to messaging and social networking services; and handling incoming calls. In this section, we discuss these hardware subsystems and software components from an energy perspective measurement.

The modern smartphone is a highly integrated piece of hardware. This tight integration means that measuring the energy use of a single subsystem is difficult without disassembling the device [1]. The information required for this, such as the circuit design specifications, is not publicly available, except for a very few products [2]. However, it is possible to measure the energy use of the entire device with the control over the subsystems to some extent. Truly independent measurement of some subsystems may not be possible, since none of the hardware components in a smartphone are isolated. For example, it is common to have Bluetooth and Wi-Fi radio embedded on the same chip. If a user turns on the Wi-Fi radio, the entire chip will be powered on. Therefore, if we measure the energy use of Wi-Fi and Bluetooth independently, and then measure the energy use of the smartphone when both Wi-Fi and Bluetooth are turned on, the sum of the two independent measurements will be greater than the combined energy use of both media.

4.2.1 Hardware subsystems

The most notable energy consumers among the hardware subsystems are the screen, the wireless network interfaces, and the CPU. Certain sensors such as the GPS receiver and the camera can also drain comparable amounts of energy when fully powered. These are discussed in detail in Section 7.4.

To measure the energy consumption of the screen alone is difficult, since mobile operating systems couple having the screen on with peaks of CPU activity. Only when all user processes are terminated, and all wireless communication and sensor hardware is turned off is it possible to get close to the scenario where only the screen is consuming energy. We attempt this in the worked example in Section 4.6.1. Similarly, to measure the energy consumption of the CPU, we can turn off the screen and request full CPU power from the operating system (i.e. with the WakeLock mechanism in Android). Communications and sensors should still be turned off as before. However, for the CPU case, we need to define the workloads that we want to measure the energy consumption of. We could saturate the CPU with computation to get the 100% CPU usage energy consumption, or generate a light load to determine the minimum consumption of the CPU required for the CPU to be on and stay in its lowest power state.

During wireless communication, the CPU is always on to handle the traffic. Therefore, it is hard to completely decouple the cost of the CPU from the power measurement. We can, however, send messages that are as large as possible to ensure that the cost of traffic handling is ignorable, compared to the cost of uploading/download data. See Section 7.3 for more detail on the energy characteristics of different wireless network interfaces.

4.2.2 Battery

Battery technology is described in Chapter 3. Measurement of the battery characteristics can be done easily if the battery can be detached from the smartphone. The smart battery interface of the smartphone can often estimate the voltage and SOC of the battery, which can be used to establish a battery profile. Such a profile can enhance the accuracy of software-based energy measurement by relating changes in the SOC to the amounts of energy consumed.

4.2.3 Operating system and software

The operating system and other software running on the smartphone greatly influences the energy consumption of the underlying hardware components. For measurement of operating system components, knowing the structure of the system and which components are active at a certain time is necessary. The most detailed information for this purpose can be obtained by instrumenting the operating system itself, and tracking the system calls. This approach has been tried on Android and Windows phones [3] with customized images of mobile operating systems.

Some factors of the operating system can also be measured using a custom application, given that other applications are not running. The application would use the operating system service whose energy use is to be measured, while the smartphone is connected to measurement hardware. Proper care must be taken to make sure that other applications and services are not running. In cases where this is difficult, repeating the measurements and operating system calls can also mitigate the error caused by other applications.

4.2.4 Sensors

Sensors such as the GPS receiver and the compass can drain the battery quickly when active. Obtaining the location from network information instead of GPS fixes is often less accurate but more energy efficient. Section 7.4 gives a detailed picture of the energy efficiency of different sensors.

4.3 Estimating device lifetime

The simplest way to determine how long the smartphone battery will last while doing a task is to perform that task, starting with a full battery, until the smartphone turns off due to low battery. The time from the beginning with full battery to the time the smartphone turned off is the estimated smartphone lifetime for that task.

Letting the battery drain from 100% to 0 can be time consuming, and when comparing different tasks, it must be repeated for each task. To conserve time, a measurement device can be used that shows how much energy is being consumed by the task. Thus scenarios of a fixed length can be planned, where many tasks are done in a shorter period of time, and their energy consumptions compared with each other as well as other such scenarios.

4.4 Energy measurement techniques

The goal of energy measurement is to collect data that can be used for analyzing the energy consumption behavior of the smartphone and for building models for describing the behavior. Power models based on energy measurement can later be used to estimate the energy consumption, without depending on the energy-measurement framework that was used to build the profile. The benefit of such a model is that once built, it can be used on many systems of the same type. For smartphones, such models can reach a high accuracy on device models of the same type [4].

4.4.1 Software-based measurement

By definition, software-based measurement suffers from a feedback loop. The energy-profiling software causes an energy overhead, thus biasing the measurement result. One way to mitigate the error is to quantify the measurement overhead using hardware-based energy measurement, and then subtract the overhead from the readings of the energy-profiling software. Such mitigation is valid only under the conditions that the overhead has been measured accurately. It is therefore important to be conservative in uncertain conditions, such as when running the energy-profiling software on a different device, with unmeasured hardware components, or with other software running on the device. Chapter 10 discusses energy-profiling software in detail. The Android battery statistics system discussed in Chapter 10 gives a good example of the workings of a software energy-profiling tool. The next section discusses smart battery APIs, an example of software-based energy measurement.

4.4.2 Smart battery APIs

Power measurement relies on the equations $P = IV$ and $E = PT$, and in the straightforward case we can multiply the measured voltage with the current for some time period to obtain the energy cost. The energy cost is thus easy to determine, given that the voltage and current values can be measured accurately, for example, with an external power meter. Unfortunately, this is challenging especially with on-device profiling using the standard APIs provided by mobile OSs. The APIs typically allow applications to query and subscribe to coarse-grained information, such as the battery voltage and the SOC. Table 4.1 presents an overview of the APIs available on modern smartphone operating systems.

As discussed in Chapter 3, the power measurement of batteries typically relies on the load voltage or coulomb counting methods. The former requires the load voltage and discharge curve of the battery, whereas the latter requires that current can be measured. Nonlinear discharge effects need to be addressed in both methods to obtain accurate results for the battery SOC. For online on-device power profiles we have the following limitations [5, 6]:

- Update rate of the battery status is low.
- The readings of the smart battery interface are not accurate.
- The required information may not be available, namely the current.

Figure 4.1 shows an overview of battery power estimation techniques. Depending on the smart battery API, voltage or current may not be available. The SOC is often available, so a coarse measurement of battery percentage change over time can be done on-device relatively easily. However, the voltage and current are required for finer-grained attribution of energy consumption to parts of the smartphone system, or applications that run on it. If these are not available, they can be estimated by software to some extent. For example, after measuring a battery's discharge rate from full to empty in a laboratory environment, its discharge curve that is how quickly the battery drains over time can be constructed. Combined with knowledge about the discharge process outlined in Chapter 3, this can can be used to estimate the voltage on similar batteries and smartphones. However, with a different battery or smartphone, results will vary. Current is usually not available, or is coarse grained. In summary, the smart battery API

Table 4.1. Overview of smartphone energy APIs

API	SOC	Health	Battery capacity	Cycle count	Voltage	Current	Temperature
Android	Yes, 1% granularity	Yes (categories)	Not in the API (some devices support/ sys/access)	No	Yes	Not in the API (some devices support/ sys/access)	Yes
iOS	Yes, 5% granularity	No	No	No	No	No	No
Windows Phone	Yes	No	No	No	No	No	No
Blackberry	Yes	Yes (categories)	Yes	Yes	Yes	No	Yes
W3C Battery status, API	Yes, 1% granularity	No	No	No	No	No	No

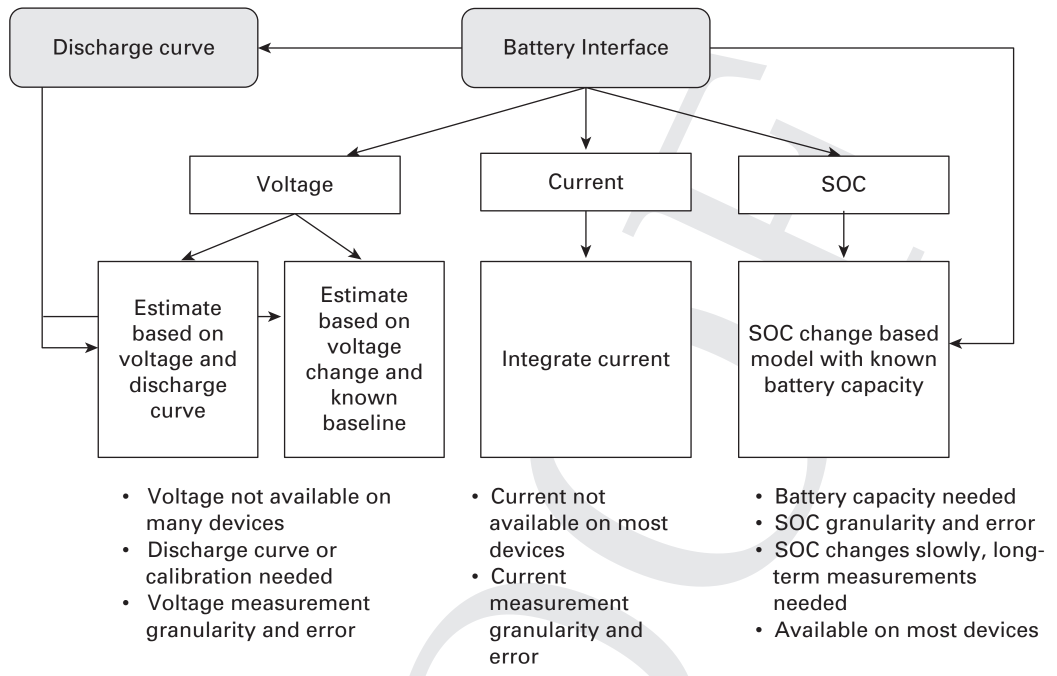

Discharge curve
Battery Interface
Voltage
Current
SOC
Estimate based on voltage and discharge curve
Estimate based on voltage change and known baseline
Integrate current
SOC change based model with known battery capacity
• Voltage not available on many devices
• Discharge curve or calibration needed
• Voltage measurement granularity and error
• Current not available on most devices
• Current measurement granularity and error
• Battery capacity needed
• SOC granularity and error
• SOC changes slowly, long-term measurements needed
• Available on most devices

Figure 4.1 Overview of power estimation techniques

is well-suited for coarse grained battery monitoring, but is insufficient for fine-grained energy measurement, and therefore hardware energy measurement is the best option for this purpose. Table 3.1 in Section 3.4 gives an indication of the sampling rate and accuracy achievable with smart battery APIs. These can be improved with software:

- The smart battery accuracy limitation can be alleviated by averaging over several smart battery readings. This increases accuracy while reducing the update rate [6].
- The smart battery update rate can be improved by first detecting the update rate and then synchronizing on-device measurement operations to minimize delay and inaccuracy [5, 7].

The next section considers hardware-based energy measurement and using such measurements in the construction of power profiles.

4.5 Hardware energy measurement and ground truth

The most accurate energy-profiling method is hardware-based energy measurement. As described in Section 4.1, the sampling rate obtained with a hardware measurement device is in 1000 s of Hz. A popular piece of power measurement hardware, the Monsoon Power Monitor[1], has an output sampling rate of 5000 Hz. A hardware energy

[1] `http://msoon.com/LabEquipment/PowerMonitor/` accessed January 6, 2014.

measurement device should be accurate and have a high sampling rate. It should offer a way to measure energy unsupervised, storing measured data on a digital medium. Finally, it should allow observation of the momentary voltage, current, and power as they are being measured. Controlling the measurement device with a computer enables all of these properties.

For hardware measurement, we need to measure the energy used by the smartphone under the conditions that we want to model. For example, to model the energy usage of the screen of the smartphone, we would keep the screen on at various brightness levels, one at a time, for a fixed period of time. We could then construct a model using standard techniques. The model would be able to estimate the energy usage of the screen when given the screen brightness level of the smartphone.

Hardware measurement is affected by some of the same issues as software-based measurement. Since the hardware measurement device measures the energy used by the entire smartphone, not just a single hardware component, the model will always contain some noise. Shutting down applications and hardware components that are not being profiled is essential. This noise cannot be eliminated completely without full control of the software and hardware of the smartphone, however. The noise can be mitigated with longer profiling periods. The longer the smartphone is in a particular model state, the more accurately its average energy use will be measured.

4.6 Worked examples

This section illustrates setups for hardware power measurement. First, we examine the case where both the measurement device and the power source are external. In this case, we consider two ways to feed power to the smartphone: using a fake battery and using the original battery with the power terminals insulated. We then briefly consider the case where the original battery is used as the power source, and only the measurement device is an external piece of hardware.

4.6.1 Measuring powered by an external power source

The most reliable way to do hardware-based energy measurement is to use a fake battery. This avoids contact errors and allows a greater range of movement for the device than using raw wires. Such a fake battery is shown in Figure 4.2. A fake battery is a piece of circuit board, or an original battery casing, with + and − terminals in the same order as the original battery, and leads connected to them for easy measurement. The fake battery should be the same size as the original battery so it stays firmly in the battery enclosure. In the best case, the device backplate can be replaced with the leads coming out between it and the phone.

Some smartphones will not boot up if they do not detect the presence of a battery, so, fake batteries for smartphones should also include a sense resistor connected in series with the − or ground lead, and exposed as another terminal. For example, Rice et al. [8] use a resistor of 0.02 Ω. This provides the phone with a fabricated estimate of the remaining battery capacity. A battery temperature sensing pin may also be needed.

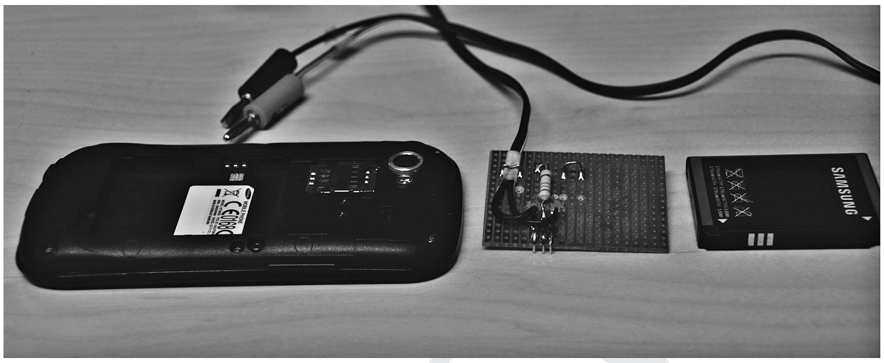


Figure 4.2 A Google Nexus S smartphone, a fake battery, and the original battery

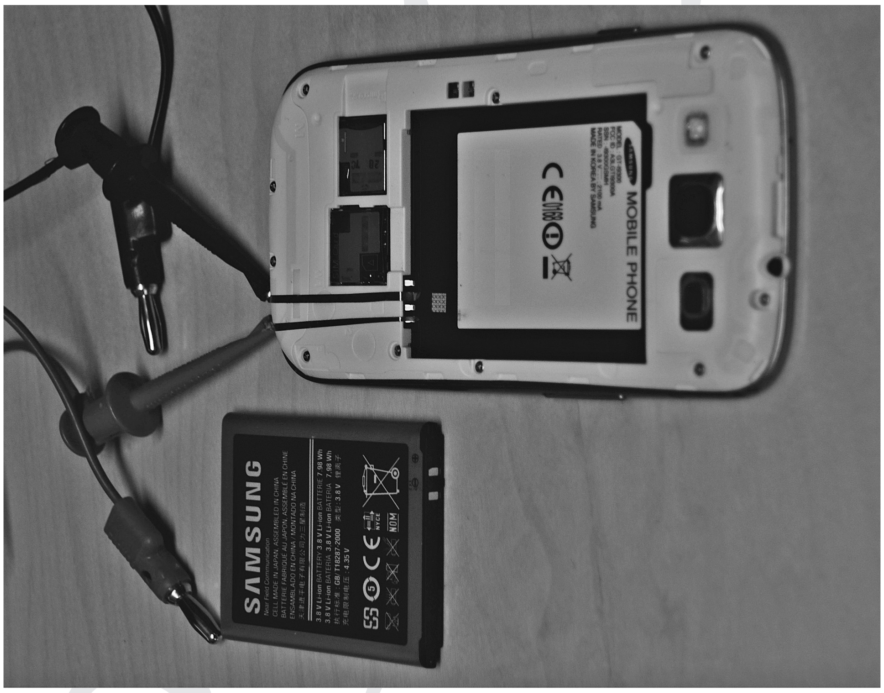


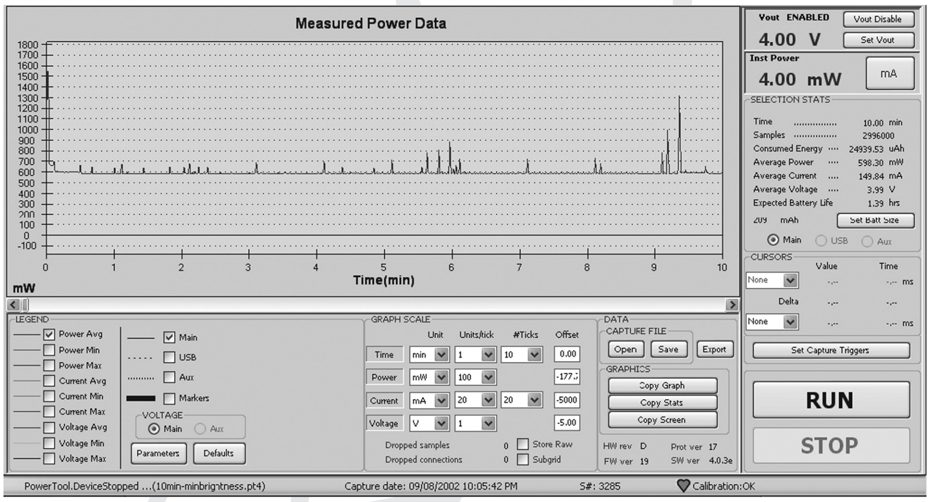
Figure 4.3 A Samsung Galaxy S III ready for hardware power measurement, with battery terminals taped over and phone terminals connected to standard leads with copper tape

A quicker way to start hardware power measurement, without a fake battery, is by using the original battery, insulation tape, and copper tape. This method is prone to contact errors and the lead may accidentally be pulled out. For experiments that require the use of the GPS, accelerometer, or orientation sensors, this is not recommended.

Figure 4.3 illustrates this approach. The Samsung Galaxy S3 smartphone in the picture has two strips of copper tape leading out of the phone casing, from the + and − terminals of the phone to the red and black leads, respectively. The original battery has the corresponding terminals taped over with insulation tape, so that the phone will be

Table 4.2. Fake battery compared with the original battery, when used with an external power source and measurement device

Property	Original battery	Fake battery
Contact errors	Yes	No
Free device movement	No	Yes
Requires electrical engineering	No	Yes
Reusable and quick to attach and detach	No	Yes
Usable on all devices	Yes	No
Minimal time to build and setup	Yes	No
Full battery information available	Yes	Possible
Usable with all hardware power sources	Yes	Yes



Vout ENABLED 4.00 V

Inst Power 4.00 mW

SELECTION STATS

Time	10.00 min
Samples	2996000
Consumed Energy	24939.53 uAh
Average Power	598.30 mW
Average Current	149.84 mA
Average Voltage	3.99 V
Expected Battery Life	1.39 hrs

209 mAh

[✓ Main] USB Aux

CURSORS

	Value	Time
None	-,--	-,-- ms
Delta	-,--	-,--
None	-,--	-,-- ms

LEGEND: [✓ Power Avg] [✓ Main]

VOLTAGE: [✓ Main] Aux

GRAPH SCALE

	Unit	Units/tick	#Ticks	Offset
Time	min	1	10	0.00
Power	mW	100		-177.:
Current	mA	20	20	-5000
Voltage	V	1		-5.00

Dropped samples 0 Store Raw
Dropped connections 0 Subgrid

HW rev D Prot ver 17
FW ver 19 SW ver 4.0.3e

PowerTool.DeviceStopped ...(10min-minbrigntness.pt4) Capture date: 09/08/2002 10:05:42 PM S#: 3285 Calibration:OK

Figure 4.4 A screenshot of the PowerTool software, measuring the energy use of a smartphone while powering it at 4 V

powered with an external power source only. Note that the other terminals of the battery should be left open. We can now replace the original battery and close the phone casing, leaving the copper tape sandwiched between the insulation tape and the phone terminals. The original battery's temperature and sense terminals will be connected to the phone as usual, and the system will not be able to distinguish this setup from the original battery.

Table 4.2 compares the approaches of hardware power measurement, using an external power source and either the original battery or a fake battery. With a correctly constructed fake battery, reuse is quick and easy, the smartphone is fully movable for use of, for example the accelerometer and orientation sensors, and the phone shutting down due to contact errors (i.e. the circuit breaking because wires are disconnected) is very rare. However, fake batteries take some time and effort to construct, require basic

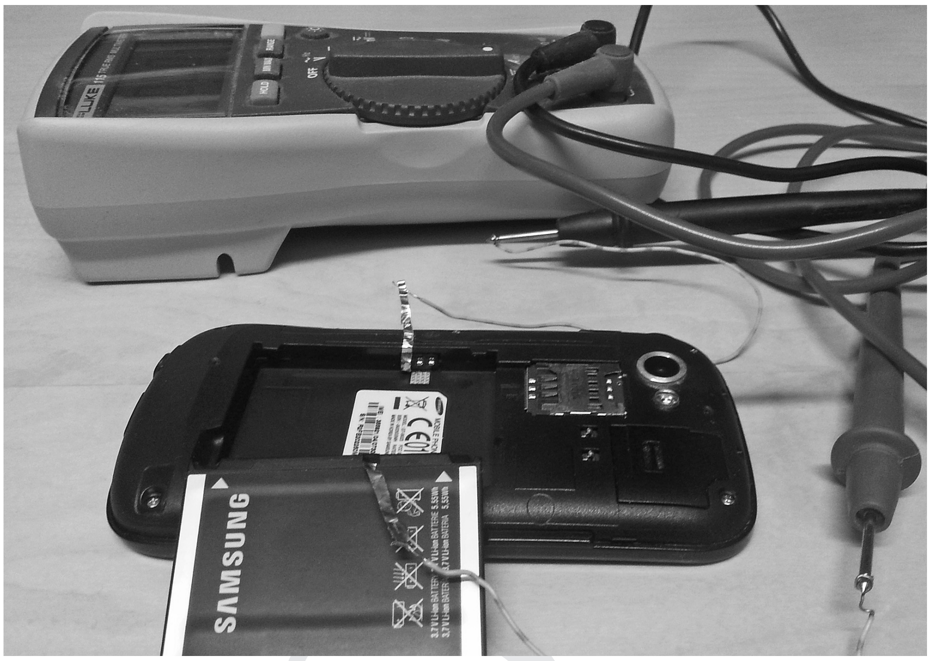


Figure 4.5 The Google Nexus S smartphone with a multimeter connected for measuring current

electrical engineering, and may not provide the smartphone with the temperature and capacity information that the original battery can provide. Finally, fake batteries cannot be used with devices without an easily replaceable battery.

In externally powered hardware power measurement, the voltage given to the phone determines the battery level the phone will report. With a 3.7 V battery, voltages below 3.7 V result in low battery levels, while voltages from 4.0 V up to 4.3 V often result in the phone reporting more than 80% charge. To avoid the phone's power-saving mode, set the voltage to 4.0 V when measuring.

Measuring powered by the original battery

When profiling a battery measurement application, we need to measure battery drain with the application and with hardware. In this case, powering the phone externally is not possible.

Figure 4.5 shows a Google Nexus S smartphone with a multimeter connected to it. The multimeter is placed between the + terminal of the battery and the + terminal of the phone. In the figure, copper tape connects the battery + terminal to one of the multimeter leads, and there is insulation tape on the copper tape, insulating it from the phone's + terminal when the battery is replaced. The + terminal of the phone has copper tape on it in the same way, connecting to the other lead of the multimeter. To start measuring, we replace the battery and make sure the two copper tapes and leads do not touch. We then turn the multimeter on to the DC current measurement position. Finally, we turn the phone on.

Table 4.3. A simple energy profile for screen brightness on Samsung Galaxy S III and S4 smartphones

Brightness level	Average power for S III	Average power for S4
Maximum	874.25 mW	933.48 mW
150 of 255	744.31 mW	766.77 mW
Minimum	598.30 mW	632.21 mW

This approach has some limitations, such as not being able to measure the voltage, a relatively low sample rate, and no easy storage of current values. Better multimeters can eliminate some of these. To measure voltage as well, we can add an oscilloscope to the setup, or replace the multimeter/oscilloscope with the Monsoon Power Monitor.

Simple hardware energy profiling example

To illustrate basic hardware energy profiling, in this section we consider the impact of screen brightness on two Android phones: a Samsung Galaxy S III and a Samsung Galaxy S4.

First, we connect the smartphone in question to the Monsoon Power Monitor as illustrated in Figure 4.3. We then connect the Power Monitor to a Windows computer, and start the PowerTool software. In PowerTool, we set Vout to 4 V and enable Vout, as shown in Figure 4.4. After this the light between the Power Monitor's two inputs should light up green. We turn on the smartphone and wait for it to boot to the main screen. Figure 4.6 shows the correct setup with a Samsung Galaxy S4 smartphone.

Before the experiment, we went to the display settings of the smartphone and made sure that automatic brightness was off, and the screen timeout was set to 10 minutes. We also turned off Wi-Fi, Bluetooth, mobile data, GPS, sound, and screen rotation, and removed any SIM cards or SD cards. We divided the hardware measurement experiment into three scenarios: maximum brightness, 150 of 255 brightness, and minimum brightness. In each of the scenarios, we set the screen brightness to the above-mentioned level and turned off the screen. To set the screen brightness accurately, we can use a screen brightness widget, or a simple Android application. Please see Appendix A for a simple Android application that sets the brightness level to a specified value.

We then simultaneously turn the screen on and press RUN in PowerTool, and unlock the phone. We let the phone stay on until its screen turns off. We then press STOP on PowerTool, and select the measured power data for exactly the first 10 minutes. This gives an average power value on the right side of the screen in PowerTool. We will use this in our model.

Table 4.3 shows the results of the experiment described above.

We have put the source for an Android application that implements this simple energy profile on GitHub.[2] The application uses simple interpolation to estimate energy use for unverified values. For screen brightness values between 0 and 150, on the S4, we used

[2] `https://github.com/lagerspetz/brightness-energy-profile` accessed January 6, 2014.

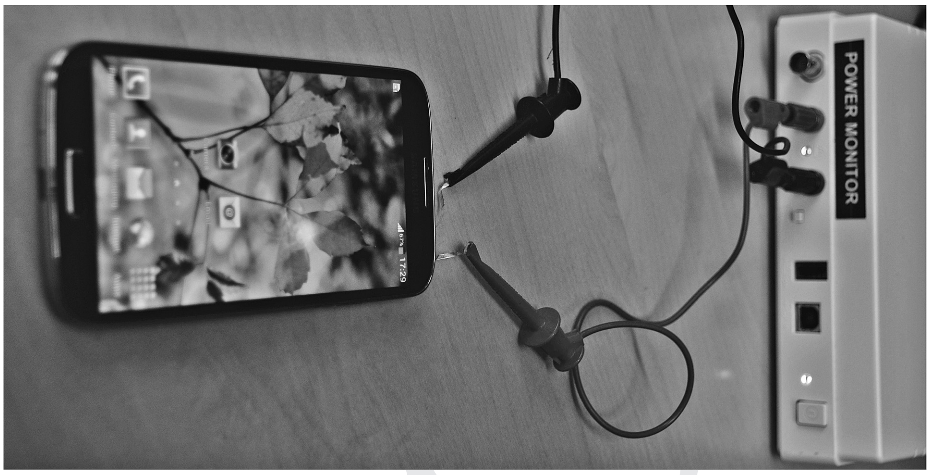


Figure 4.6 A Samsung Galaxy S4 being measured by the Power Monitor

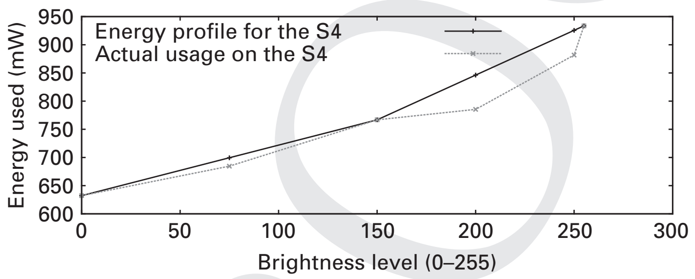


Figure 4.7 A simple energy profile for screen brightness on the S4

$632.21 \times b$, where b is the brightness value. For values between 150 and 255, we used $(v - 150)/(255 - 150) \times 766.77$. The values for the S3 are handled similarly when running on an S3. For other devices, the values for the S3 are used, and results will be inaccurate.

Please see Appendix A for a simple Android application that estimates the energy use of the device based on screen brightness and the model shown in Table 4.3.

This simple model has an error of less than 1 mW when the brightness is set to minimum on a factory-reset S4 phone sitting on the main screen. When brightness is set to 75, a value not included in the profile, the simple interpolation gives us a value of 699.49 mW while the actual usage is 684.57 mW, so the error is 15 mW. For 200, the error is 61 mW, again overestimating the energy use. For 250, close to the measured maximum value of 255, the error is 44 mW.

In Figure 4.7, the energy profile for the S4 is compared against actual energy use. The profile is accurate at the measured brightness levels, but the error grows quickly in other conditions, especially at higher brightness levels. Compensation variables should be calculated if the same profile is to be used with other devices.

4.7 Summary

In this chapter, we discussed the nature of energy consumption and the challenges for measuring it accurately. We introduced software-based methods for energy measurement and profiling, such as the smart battery API. We discussed the process of hardware-based energy measurement. We introduced three ways to measure energy using hardware:

- with an external power source and the original battery,
- with an external power source and a fake battery, and
- powered by the original battery.

We concluded with how hardware-based power measurement can be accomplished for an off-the-shelf modern smartphone, and how the resulting values can be used to build a power profile. Chapter 10 discusses power-profiling software in detail.

References

[1] A. Carroll and G. Heiser, "The systems hacker's guide to the galaxy energy usage in a modern smartphone," in *Proc. 4th Asia-Pacific Workshop on Systems*. New York, NY, USA: ACM, 2013, pp. 5:1–5:7. [Online]. Available: http://doi.acm.org/10.1145/2500727.2500734

[2] ——, "An analysis of power consumption in a smartphone," in *Proc. 2010 USENIX Ann. Technical Conf.* Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855861

[3] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app? Fine grained energy accounting on smartphones with eprof," in *EuroSys*, 2012.

[4] G. Creus and M. Kuulusa, "Optimizing mobile software with built-in power profiling," in *Mobile Phone Programming*, F. H. Fitzek and F. Reichert, Eds. Springer Netherlands, 2007, pp. 449–462. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-5969-8_25

[5] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, "Devscope: a nonintrusive and online power analysis tool for smartphone hardware components," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2012, pp. 353–362. [Online]. Available: http://doi.acm.org/10.1145/2380445.2380502

[6] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2011, pp. 335–348.

[7] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: fast self-constructive power modeling of smartphones based on battery voltage dynamics," in *Proc. 10th USENIX Conf. on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2013, pp. 43–56.

[8] A. Rice and S. Hay, "Decomposing power measurements for mobile devices," in *2010 IEEE Int. Conference on Pervasive Computing and Communications (PerCom)*, 2010, pp. 70–78.

5 On human behavior and energy efficiency

The smartphone is a highly integrated and complex device. Its energy efficiency can be optimized in various ways, but the charging behavior and application use pattern have a large impact on battery life and energy efficiency. This section discusses the impact of human behavior on the energy efficiency and battery life of a smartphone. It continues with the other side of the coin: how battery-awareness applications change human behavior. Finally, it lists some techniques for how to get the most of remaining battery life as a smartphone user.

5.1 Human–battery interaction

The term *human–battery interaction* was coined by Banerjee et al. in [1]. It refers to the different ways that the user interacts with the smartphone battery. The interactions happen in both directions with the user determining the charging patterns and power management related phone configuration, and the phone giving feedback on the battery status and power consumption.

The charging pattern of a particular user is of interest because it can give indications about whether the user in fact ever even runs out of battery. For example, a particular user may always charge the phone overnight regardless of the current battery level. The charging behavior guides the power management of the device so that it is neither too conservative nor too aggressive. Coming back to the example user who charges the phone every night, if the battery would last longer than one day with that user's typical usage, there is no need for further energy savings and, instead, more energy could be spent improving the quality of the mobile services they use. Llama [1] is an example of such a power-management system. It tries to predict the amount of so-called excess energy that would be left over when the user starts recharging the mobile device, and then uses that energy to improve the quality of service (QoS) of the applications being used.

Many studies on the charging behavior have been conducted to date [1, 2, 3, 4, 5]. All of them agree on a couple of salient points. First, recharging is triggered by either the current battery level or the context which derives from time and/or location, which leads to two distinguishable user types. This observation is of notable importance to power-management systems such as the one mentioned above. Specifically, context-driven charging behavior allows power-management solutions to improve the QoS without having any negative influence on the user experience. In contrast, users tend to be sensitive toward changes in the recharging cycle, which means that a user who recharges the phone only when the battery reaches a low level would notice and be annoyed by any significant increase in energy consumption. A second common finding is that the user behavior may vary a lot. Even changes in user type were noticed in one of the studies. Therefore, it is clear that a one-size-fits-all solution does not work and the power-management solutions must adapt to user behavior.

Most of the existing studies sample only a very small number of users, even just a few tens of users, with the exception of one study of about 4000 Android users [6] and another one of about 20 000 BlackBerry users [5]. However, with the rapid development in the mobile industry, these two studies are becoming old and there is clearly room for many more large-scale studies.

User behavior has a significant effect on the workload and energy consumption of a mobile device. For example, users can choose which applications to use and how to use those applications, such as when to send a search request and which video to watch. Moreover, to some extent, they can also decide which access point to connect to and where to store data.

Some user studies [7, 8] have focused on the relationship between the user behavior and the power consumption of the mobile device. A typical methodology is to install a context-monitoring application on users' mobile devices. The application tracks the mobile device settings, hardware resource consumption, network traffic, and/or the user inputs. This information is then used to estimate the power consumption based on power models and to match user activities with the power consumption. For example, Shye et al. [7] developed a logger application for Android G1 mobile phones and used it to collect traces of real user activity. The log included the hardware information, such as the CPU use for each CPU operating mode, the brightness of the screen, and the count of bytes transferred with Wi-Fi during a given interval. The log was entered into a linear regression power model to obtain the power consumption corresponding to user activities.

As shown in the user study by Rahmati et al. [9], most mobile users had no knowledge of the power characteristics of their devices and applications. Moreover, most mobile users underuse the power-saving settings of their mobile devices. Hence, in addition to improving the energy efficiency of hardware/software by taking user behavior into account, we also firmly believe that the tools that can show the predicted battery life [10] and can demonstrate the relationship between power consumption and user activities will help mobile users to extend their device battery lifetimes.

5.2 Human factors in battery-awareness applications

As discussed before, human behavior affects the mobile device and its battery. In this section we consider the reverse: how mobile battery-awareness applications can affect human behavior.

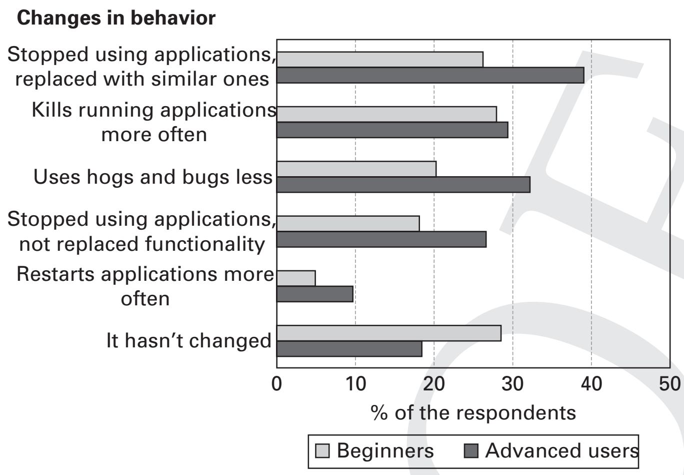


Figure 5.1 Carat changed the application use behavior of users

The primary goal of mobile battery-awareness applications is to make the user aware of what consumes energy. It is natural to consider the effect they have on human behavior. To understand how better energy awareness changes user behavior, Athukorala et al. [11] conducted a survey of 1140 Carat users, and asked them how the application has changed their behavior. The Carat application is discussed in Section 10.7.3. The authors call users who had used Carat for more than three months *Advanced users*, and those who had used it less *Beginners*. Figure 5.1 shows some of the results. Advanced users are much more likely to stop using applications that are identified as high energy consumers by Carat. They also close applications more often than beginners, and use highly energy-consuming applications less. In contrast, beginners think the application has not changed their behavior. The results also showed that advanced users open Carat less frequently than beginners. However, they have gained better battery life, charge their devices less frequently, kill reported hogs and bugs more often, and have learned to better manage their battery without the help of Carat. The authors propose four guidelines for building battery-awareness applications:

1. Building on the observation that further understanding promotes use of a feature, battery-awareness applications should expose to the user not just recommendations, but also the reasoning or data behind them,
2. As long-term users are highly valuable in a community-based system that learns from users, such as Carat, the authors suggest tailoring community-based battery awareness applications to retain long-term users.
3. The audience of an application can be very diverse, and it is crucial to use precise vocabulary. The authors suggest taking into account the audience when formulating feedback to convey precisely what is intended.
4. The authors recommend that battery-awareness applications distinguish system components from third-party applications when making diagnoses and

recommendations. System applications often cannot be stopped or removed by the user, and this should be taken into account when making recommendations.

The guidelines of Athukorala et al. target community-based mobile battery awareness applications. Single-device applications can take advantage of all except the second one.

While the guidelines above are helpful for battery-awareness application developers, smartphone users may be interested in what they can do right now to improve the battery life of their device. To conclude the discussion of human behavior, the next section discusses the challenge of getting the most out of remaining battery life, while still communicating with others.

Box 5.1 General advice for maximizing the smartphone standby time

To maximize the standby time of your smartphone, you can do one or more of the following things:

- Turn off Wi-Fi
- Turn off cellular data connections
- Turn off Bluetooth
- Turn the screen off
- Reduce screen brightness or use automatic screen brightness
- Terminate all applications that you are not actively using
- Turn off GPS
- Turn off gesture and face detection
- Turn off speech recognition or voice activation
- Use SMS instead of internet messaging
- Use cellular calls instead of VoIP

5.3 Experiment: Getting the most of remaining battery life

A challenge that mobile device users face today is getting the battery to last for the entire day. Especially, when the battery is low, it is important to make it last as long as possible while staying connected with email, phone, or VoIP functionality. Box 5.1 lists general advice for maximizing the standby time of a smartphone.

This section is a detailed experiment on how to get the most out of the last 15% of the smartphone battery. We follow the general advice above and consider a few scenarios that include internet-based and cellular communication. The results in this section have been summarized from [12].

5.3.1 Experimental setup

For this experiment we use the Google Nexus S. Table 5.1 shows the specifications of the Google Nexus S smartphone. Note that the device has an AMOLED display, 5.55

Table 5.1. Feature specifications of the Google Nexus S

Feature	Specification
SoC	Samsung Exynos 3 Single
CPU	1 GHz single-core ARM cortex-A8 processor
OS	Android 4.0.4
RAM	512MB
Storage	16GB
Display	Super AMOLED 800 × 480 pixels
Network	2G: GSM 850 / 900 / 1800 / 1900
	3G: HSDPA 900 / 1700 / 2100
Data	SPEED: HSDPA, 7.2 Mbps; HSUPA, 5.76 Mbps
	WLAN: Wi-Fi 802.11 b/g/n, DLNA, Wi-Fi hotspot
Bluetooth	Yes, v2.1 with A2DP
Battery	1500 mAh, 3.7V, Li-ion 5.55Wh

Wh battery capacity, and high-speed connectivity. Energy use is measured using the Monsoon Power Monitor, introduced in Section 4.5. The experiment is run until 15% of 5.55 Wh, that is, 832.5 mWh, is consumed. Then, the duration of the experiment and the average power is recorded.

The experiment examines two communication activities, email and calling. Calling is further divided into VoIP and cellular calls. The experiment aims to minimize the battery drain during the activities, and therefore obtain the longest possible battery life, when only 15% of the battery capacity is remaining. In this experiment, we assume that the phone has no built-in power-saving mode, and that the phone can remain functional for the entire span of 15% to 0%. For phones with automatic power-saving modes, the benefits of the actions here will be smaller, especially the effect of screen brightness. Most mobile devices typically shut down before the battery reaches 0%, but the actions in this section can be used for any 15% of the battery; specifically, for the purposes of this experiment, the span could as well be 20% to 5% or 100% to 85%.

We also compare the battery drain between different communication technologies (Wi-Fi, 3G, and 2G) and display brightness levels (Auto, Minimum, and Maximum).

5.3.2 Email

This section examines how long users can keep sending emails with 15%, or 0.8325 Wh, of the battery capacity available. The email application in this experiment is Gmail. The procedure was to log on to Gmail, create a new email (100 characters) with an attachment (2MB), and send it every 2 minutes. The average time for completing this process is 85 seconds. For completeness, we also conduct the experiment without attachments. For this case, nine emails are sent within 10 minutes, with minimum backlight brightness.

Table 5.2. Expected battery life while using email with different communication methods and brightness levels

Data type	Wi-Fi	Wi-Fi	3G	3G	2G	2G
Brightness level	Auto	Max	Auto	Max	Auto	Max
Avg. power (W)	**0.98**	1.49	1.25	1.76	1.87	2.39
Battery life (min)	**51.12**	33.58	40.11	28.56	26.89	21.06
# Emails sent	15	9	11	8	8	6

Table 5.3. Expected battery life with Skype, screen brightness set to minimum and with the display turned off, on Wi-Fi, 3G, and 2G

Data type	WiFi	WiFi	3G	3G	2G	2G
Brightness Level	Min	Off	Min	Off	Min	Off
Avg. power (W)	1.11	**0.73**	1.59	0.12	1.81	0.14
Battery life (min)	44.99	**68.62**	31.64	41.67	27.64	36.01

5.3.3 Phone and VoIP calls

In this scenario, Skype or the dialer application is used to make a phone call and keep the call running. The experiment is performed in a relatively quiet environment. With Skype, we also examine the case where the screen is locked. Usually Android will turn off the screen during a call, but when users accidentally touch the screen, it will turn on again. Locking the screen guarantees that it uses no energy.

5.3.4 Email scenario results

Table 5.2 shows the results of the email scenario. The longest battery life is achieved with automatic brightness while using Wi-Fi.

With minimum screen brightness, there were improvements of 5% to 13% in expected battery life, when compared to setting brightness to automatic. This amounted to one extra email in all scenarios. However, the difference between maximum and automatic brightness is much larger, up to 52%.

Interestingly, using 3G is more efficient than 2G, even though it is usually the more power-hungry alternative. However, if we send emails without attachments, 2G improves expected battery life by 2.2% over 3G. Wi-Fi is the most effective communication method in this case giving a 11.8% improvement over 3G.

5.3.5 Phone call and VoIP results

Table 5.3 shows that the battery lasts the longest with Skype when the screen is turned off and Wi-Fi is used. With no nearby Wi-Fi access points, the second option is 3G, with about 60% of the battery life, just over 41 minutes.

Table 5.4. Expected battery life on a cellular call, display turned off, on Wi-Fi, 3G, and 2G

Data type	WiFi	3G	2G
Avg. power (W)	0.73	0.73	**0.58**
Battery life (min)	68.91	68.87	**86.11**

The power consumption of cellular calls is noticeably lower than that of Skype, as seen in Table 5.4. Disabling data connectivity or switching to 2G gives the best battery life here. Note that when on a cellular call with 2G only, the data connection cannot be used simultaneously, so switching to 2G and turning off mobile data does not give an additional battery life benefit.

5.3.6 Summary of control points

This section lists the actions for getting the longest battery life while still being able to communicate on a smartphone.

The actions in order of decreasing improvement for using email are:

1. Connect to Wi-Fi, set screen brightness to minimum or automatic.
2. Connect to 3G, set screen brightness to minimum or automatic.
3. Connect to 2G, set screen brightness to minimum or automatic.

Note that 2G gives better battery life with short emails without attachments.

For Skype, the actions are:

1. Connect to Wi-Fi, lock the screen.
2. Connect to 3G, lock the screen.

For cellular calls, the actions are:

1. Disable mobile data or connect to 2G, lock the screen.
2. Connect to 3G, lock the screen.
3. Connect to Wi-Fi, lock the screen.

5.4 Summary

This chapter discussed the human behavior aspect of smartphone battery management, specifically the charging behavior, and the impact on energy consumption of the way users interact with their devices. User studies show that most smartphone users are not aware of the power-saving features of their devices. To show how a battery-awareness application helped change this for the better, we described a user study done with Carat users and concluded it with helpful advice for getting the most out of remaining battery life.

References

[1] N. Banerjee, A. Rahmati, M. D. Corner, S. Rollins, and L. Zhong, “Users and batteries: Interactions and adaptive energy management in mobile systems,” in *UbiComp 2007: Ubiquitous Computing*, ser. Lecture Notes in Computer Science, J. Krumm, G. D. Abowd, A. Seneviratne, and T. Strang, Eds. Springer Berlin Heidelberg, 2007, vol. 4717, pp. 217–234. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74853-3_13

[2] A. Rahmati and L. Zhong, “Human-battery interaction on mobile phones,” *Pervasive and Mobile Computing*, vol. 5, no. 5, pp. 465–477, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119208000746

[3] M. V. J. Heikkinen, J. K. Nurminen, T. Smura, and H. HäMmäInen, “Energy efficiency of mobile handsets: Measuring user attitudes and behavior,” *Telemat. Inf.*, vol. 29, no. 4, pp. 387–399, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.tele.2012.01.005

[4] D. Ferreira, E. Ferreira, J. Goncalves, V. Kostakos, and A. K. Dey, “Revisiting human-battery interaction with an interactive battery interface,” in *Proc. of Ubicomp 2013*. ACM, 2013.

[5] E. A. Oliver and S. Keshav, “An empirical approach to smartphone energy level prediction,” in *Proc. 13th Int. Conf. on Ubiquitous Computing*. New York, NY, USA: ACM, 2011, pp. 345–354. [Online]. Available: http://doi.acm.org/10.1145/2030112.2030159

[6] D. Ferreira, A. Dey, and V. Kostakos, “Understanding human-smartphone concerns: A study of battery life,” in *Pervasive Computing*, ser. Lecture Notes in Computer Science, K. Lyons, J. Hightower, and E. Huang, Eds. Springer Berlin Heidelberg, 2011, vol. 6696, pp. 19–33. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21726-5_2

[7] A. Shye, B. Scholbrock, and G. Memik, “Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures,” in *Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669135

[8] N. Vallina-Rodriguez, P. Hui, J. Crowcroft, and A. Rice, “Exhausting battery statistics: understanding the energy demands on mobile handsets,” in *Proc. 2nd ACM SIGCOMM Workshop on Networking, Systems, and Applications on Mobile Handhelds*. New York, NY, USA: ACM, 2010, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/1851322.1851327

[9] A. Rahmati, A. Qian, and L. Zhong, “Understanding human-battery interaction on mobile phones,” in *Proc. 9th Int. Conf. on Human Computer Interaction with Mobile Devices and Services*. New York, NY, USA: ACM, 2007, pp. 265–272. [Online]. Available: http://doi.acm. org/10.1145/1377999.1378017

[10] N. Ravi, J. Scott, L. Han, and L. Iftode, “Context-aware battery management for mobile phones,” in *Proc. 2008 6th Annual IEEE Int. Conf. on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 224–233. [Online]. Available: http://portal.acm.org/citation.cfm?id=1371610.1372956

[11] K. Athukorala, E. Lagerspetz, M. von Kügelgen, A. Jylhä, A. J. Oliner, G. Jacucci, and S. Tarkoma, “How Carat affects user behavior: Implications for mobile battery awareness applications,” in *CHI ’14: Proc. SIGCHI Conf. on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2014.

[12] C. Shen, “Energy profiling of hardware subsystems and user settings on Android,” Master’s thesis, Department of Computer Science, University of Helsinki, Helsinki, Finland, May 2013. [Online]. Available: http://hdl.handle.net/10138/39542

Part II

Energy management and conservation

6 Overview

A mobile device consists of hardware components, such as microprocessors, wireless network interfaces, display and touchscreen, storage, cameras and sensors; and software including the operating system and applications. To understand and optimize energy consumption, models are needed for the subsystems and components of the smartphone, including the OS and applications.

In this chapter, we give an overview of the smartphone energy consumption and the methodologies that can be used for power modeling and optimization. We aim to answer the question: where is the smartphone energy spent? and its follow-up question: how can we maximize the energy efficiency?

6.1 Methodology

To address the above questions, the first thing we need is a well-defined and rigorous methodology.

6.1.1 Energy profiling terminology

For a thorough treatment of a topic, it is generally important that the core concepts are properly named and that the terminology is consistent and well understood. Otherwise, it is difficult to grasp what exactly is software-based energy profiling and optimization. Furthermore, as we will see especially in the profilers section, the literature is rich with plenty of different kinds of solution which are presented as unique tools but are often combinations of different kinds of underlying technique. Hence, it is necessary to be able to categorize these solutions and to analyze their advantages and disadvantages, which requires a good understanding of the relevance of the different components of an individual energy profiler, such as a specific power-modeling technique. For these reasons, we briefly define here the most important terms that we use in this part.

- **Power model** is a mathematical representation of power draw. It can be a function of variables that quantify the impacting factors of power consumption, such as the use of a hardware component and the number of packets delivered through a wireless network interface, with the desired power draw as output. Usually the values of these variables can be directly obtained from measurement carried out on the smartphone or in the network. A power model can characterize a single subsystem, a combination of them, or even a whole smartphone (system-level model). A simple example of a subsystem power model is a coarse-grained power model of display that is a function of a single variable: brightness level (the example given in Section 4.6.1).

- **Energy model** is the equivalent of power model for energy consumption. Hence, it is a mathematical representation of the amount of energy consumed by the execution of a specific task or a piece of code.
- **Power measurement** is the act of measuring the power draw of a smartphone or its subsystem using an external instrument (e.g. a multimeter) connected to the smartphone or a smart battery interface that directly provides instantaneous power values. The values are usually averaged over a specific time window.
- **Power estimation/prediction** reports the power draw of a smartphone or its subsystem based on power model(s). The accuracy of the power estimation/prediction relies on the accuracy of the power models used. Depending on whether the inputs describe the instantaneous state or the average workload over a specific time window, the report can be the desired instantaneous power draw or the average of the desired power draw over the time window. If the inputs describe the workload that has been executed, the report is an estimate of how much power was consumed by the given workload. However, if the inputs are the predicted workload or state, the report is a prediction of how much power would be consumed to complete the predicted workload or to stay in the predicted state.
- **Power and/or energy profiler/monitor** is a system that characterizes the power and/or energy consumption of a smartphone. Different profilers provide this characterization on different abstraction levels (e.g. system vs. application) and in different levels of granularity (error, sampling rate). Power values are typically reported as averages over a specified time window, whereas energy values can also be reported per task. The profiler is either based on power measurements, that is it is an external instrument, or on power estimation in which case it relies on power and/or energy models and it may require training and calibration. A model-based profiler is usually a piece of software running on the smartphone.

6.1.2 Model-based energy and power profiling

Figure 6.1 gives an overview of the model-based energy and power-profiling process. The figure divides the process into different phases that can be based on human, offline machine, or online machine activity. For example, method selection and parametrization typically involve human activity. The modeling and verification and validation phases can involve both offline and online machine activity. Actual usage of a power model in the estimation phase is typically automatic and performed online.

The process starts with the expert selecting the method and basic modeling tools. After this, the power measurement phase starts first with calibration and then with the power measurements. The power measurements can be combined with system logs and other relevant information to better understand the fine-grained power consumption of

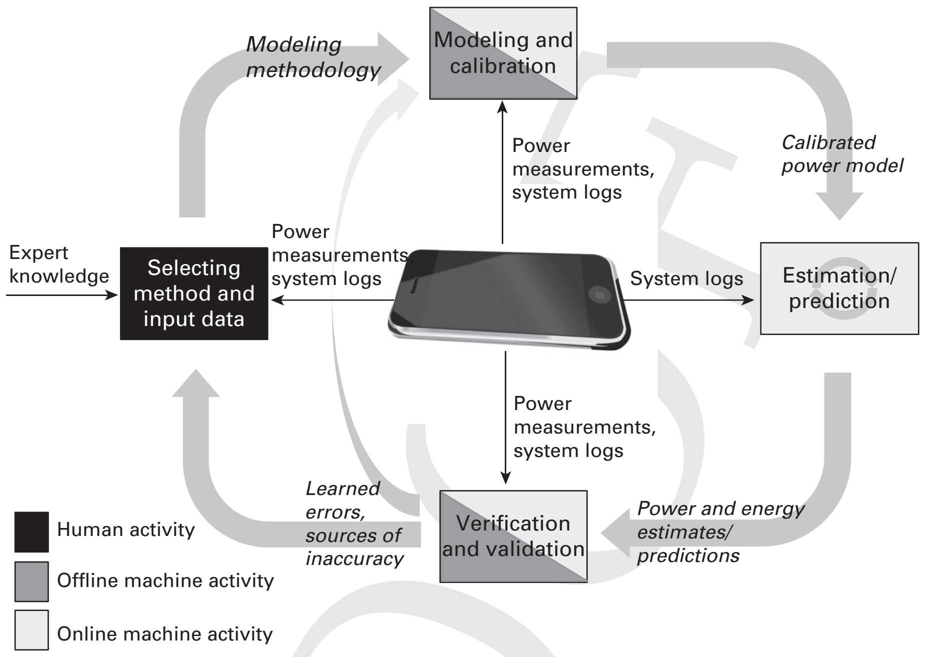

Modeling methodology
Modeling and calibration
Calibrated power model
Power measurements, system logs
Expert knowledge
Selecting method and input data
Power measurements system logs
System logs
Estimation/ prediction
Power measurements, system logs
Learned errors, sources of inaccuracy
Verification and validation
Power and energy estimates/ predictions
Human activity
Offline machine activity
Online machine activity

Figure 6.1 The process of energy and power profiling includes modeling, estimation, and a feedback loop through validation and verification to refine and recalibrate the models

the device. The measurements, expert knowledge, and additional information are then used according to the methodology to create a power model of the device.

The power model is then used to estimate the online power consumption. This phase can use system logs for fine-grained and accurate estimation. The generated power and energy estimates need to be verified and validated through online or offline activity. The estimates are compared with the actual power measurements to understand the accuracy of the estimation phase and the underlying model. This phase typically involves analyzing system logs for sources of inaccuracy. This phase provides valuable input for the expert overseeing the model generation and usage process. The verification and validation phase can also directly provide feedback to the modeling and calibration phase. Thus the feedback loop can be fully automatic without human activity or it can involve the expert user.

6.2 Power modeling

To answer our first question, that is where is the energy spent, we need to model the energy consumption and to understand how the different hardware and software components affect the overall energy consumption of the smartphone. This is a non-trivial task, because the elementary energy consumption of components may exhibit nonlinearity and the individual models may not add up in a linear fashion. For example, hardware components may have dependencies, for example shared circuitry. Let us consider the case where we have two separate and independently derived power models for the Wi-Fi and LTE of a smartphone that has a modem subsystem for the two wireless protocols.The power models work well when only one of the wireless protocols is used, but when Wi-Fi and LTE are used together the linear combination of the models does not work well, because they do not take into account the integrated nature of the hardware implementation. Indeed, while it is important to understand the specific energy models of the components, it is also as important to understand how the various components work together and how to combine the models. In this book, we consider various modeling techniques, including holistic techniques.

6.2.1 Overall process

When modeling power or energy consumption, the smartphone is the test subject that is monitored, as illustrated Figure 6.1. Power measurements are necessary to build and refine a power model. The measured power consumption data and system state are first used to train and build power-consumption models. In this phase, a set of well-defined experiments explore the power consumption with various inputs and settings. In addition, a model obtained from training can be updated based on runtime observations. The power model estimates the power draw of the system based on observations of the device behavior.

The offline learning and calibration part typically uses external power monitoring tools for high accuracy energy-consumption data. With these tools it is possible to measure the energy draw of the device with an external power supply as well as with the regular smartphone battery as the power supply. In many cases it is desirable to have the battery included to factor in its effects that were examined in Chapter 3.

The power monitors report the overall energy draw of the smartphone. In many cases, we are interested in the energy profile of a particular subsystem of the device. Development versions of smartphones typically have hardware debug interfaces that allow the energy consumption of a specific subsystem to be read; however, these interfaces are not available on consumer devices. Therefore, other techniques are needed to build subsystem-specific models.

6.2.2 Power measurements

By now it should be clear that power measurements are an integral part of the power/energy modeling process. Figure 6.2 presents a simple taxonomy of power measurement. Assuming that only device-level power information is available, the methods at hand are to use an external power monitor tool or to use on-board software API interfaces for self-monitoring to read the current, voltage, or battery capacity. The measurement can then focus on the device, its subsystems, or software running on the device. The self-monitoring methods will introduce bias due to the monitoring software running on the device. Moreover, the software APIs typically provide coarse-grained data.

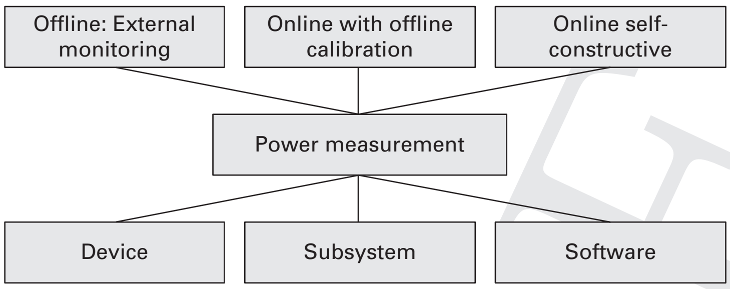


Figure 6.2 A power measurement taxonomy

Instrumentation of the application or OS is needed to be able to map the overall power consumption to subsystems and software processes.

In Chapter 4, we examined external power monitors and considered two important cases: measurement with the original battery and with a fake battery. In the former case, the battery effects are included in the measurement. In the latter case, the battery effects are not involved. The external power monitors are the gold standard for smartphone and mobile device power analysis due to their high precision and accuracy. They are limited by requiring a laboratory setting and they cannot be used to measure large numbers of devices. Therefore, model-based energy profilers are needed for outdoor and mobile experiments as well as for wide-scale energy profiling of thousands of devices.

6.2.3 Subsystem models

A subsystem-specific energy model can be built by carefully designed experiments targeting the subsystem of interest. For example, if Wi-Fi uplink is the target subsystem and use case, all the other features would be turned off to the extent possible. The challenge is that the smartphone is a complex device consisting of many subsystems and countless interactions between them when running software.

Box 6.1 Example energy model

An energy model can be formulated as follows for a component-based system [1]:

$$y(t) = f(x_1(t), x_2(t), \ldots, x_n(t)), \tag{6.1}$$

where $y(t)$ represents the energy draw in the time interval t, and the functions $x_1(t), \ldots, x_n(t)$ represent the component-level behaviors. The function f can be linear or nonlinear. The overall energy drain for a given time period can be obtained by integrating $y(t)$ over the period. The accuracy of the energy estimation is determined by the granularity of the time intervals. Typically time intervals of at least 10 ms (100 Hz) are needed for reliably estimating per-application energy consumption in a multitasking system.

Insights on application and subsystem energy behavior can be gained by complementing the external measurement with data from an instrumented OS and application. The additional instrumentation data provide a fine-grained view to the internals of the device.

6.2.4 Taxonomy

Figure 6.3 outlines a taxonomy for power modeling. The model can be based on the use of resources [2], power triggers [3], or code analysis, to give some frequently used methods. Linear regression is frequently used to estimate the power or energy consumption. A model based on linear regression captures the relationship between an input workload and the power consumption. For example, resource use is frequently employed with linear regression to model the energy consumption of CPU usage, screen brightness, and network activity. The benefit of linear regression is that it is simple to implement and efficient. On the other hand, this method is limited by the linearity assumption, because the relationships typically exhibit nonlinearity. A more sophisticated method is to build finite state machines (FSMs) that capture the power states of the system and transitions between power states. Such a system typically relies on power triggers and can capture more complex energy consumption behaviors. We provide a detailed treatment of different modeling approaches in Chapter 9. Specifically, we will go through three different example approaches that correspond to use-based, system call-based, and event-based modeling.

In practice, models are imperfect and the operating environment changes all the time, making prediction difficult. Models are also limited by the information exposed by the underlying system, the battery sensor, and the granularity of the exposed information varies. The overall accuracy of software profilers varies and depends on the underlying models and their calibration, the granularity and accuracy of the input data given by the smartphone software, and the errors from not isolating the device from the measurement instrument. The software profiler is running in the same device that it is measuring and thus it will introduce error to the output.

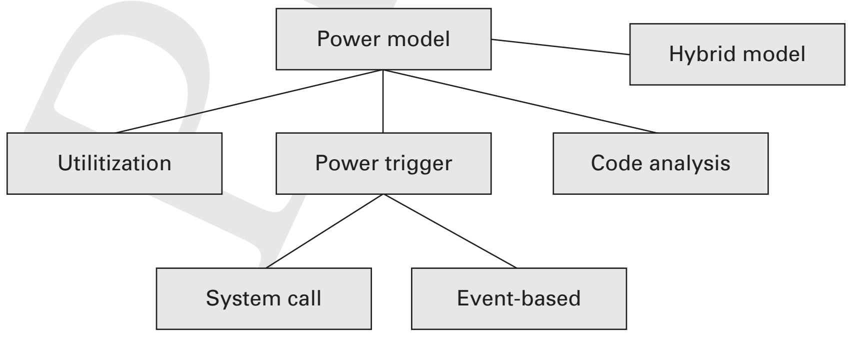


Figure 6.3 A power modeling taxonomy

6.3 Power optimization

To address the second question, that is how to maximize the energy efficiency, we need to have the above energy models of the components as well as knowledge of the control points and options to choose that allow optimization decisions to be made based on the models. An optimization decision is a system configuration change, an option to select that, given the current knowledge, will lead to an improved situation aiming for the optimal energy consumption of the device. Ideally, the models allow the prediction of system behavior into the near future and assessment of the impact of various changes to system parameters. Then it is a matter of selecting the suitable changes for execution.

Figure 6.4 illustrates the energy optimization control loop running on the device, in which the the previously learned model is used to make optimization decisions online. The decisions then in turn affect the energy consumption creating a control loop. The methodology is typically divided into two distinct parts: offline training and calibration, and online prediction and optimization.

The work presented in [4] provides an example of energy-aware optimizations. In that system, the supply and consumption of energy is monitored based on which the system behavior is steered either toward good user experience, if plenty of energy is available, or toward energy conservation, if energy is scarce. The system is based on the PowerScope [5] energy profiler. We discuss this work in more detail in Chapters 8 (the Odyssey OS in Section 8.6) and 10 (PowerScope).

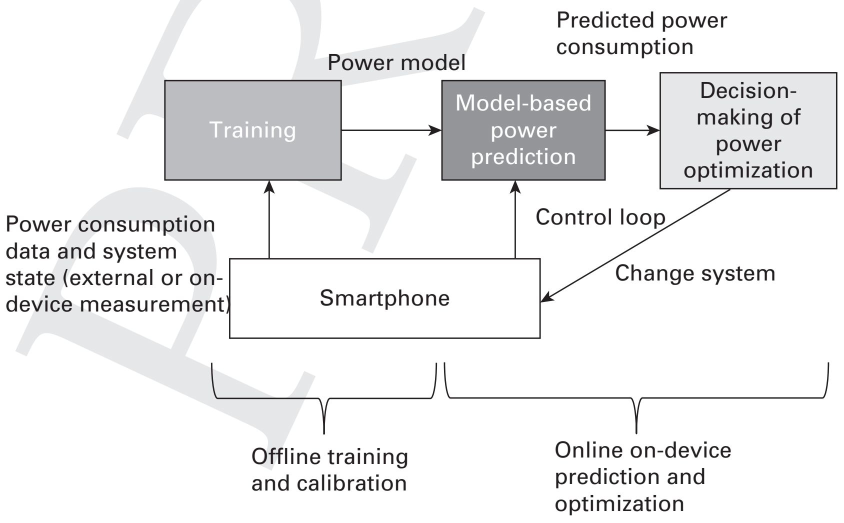

Predicted power consumption
Power model
Training
Model-based power prediction
Decision-making of power optimization
Power consumption data and system state (external or on-device measurement)
Control loop
Change system
Smartphone
Offline training and calibration
Online on-device prediction and optimization

Figure 6.4 Energy optimization control loop

6.4 Summary and organization of Part II

In this chapter, we presented an overview of Part II which focuses on the characterization of the energy consumption of smartphones and their subsystems. The key observations are the following:

- A holistic approach is necessary to understand the overall energy consumption of a complex system, such as a smartphone, but the power draw of a specific subsystem is often characterized separately.
- Power and energy profiling is a key concept in energy-consumption characterization and it always involves power measurements and often also power and energy modeling.
- Power and energy modeling builds on the power measurement that is used to create elementary subsystem models and learn the relationships between subsystems and software processes.
- Power measurement is typically complemented by measurement data from the OS and applications, which are required to correlate software processes and their parts with energy consumption and hardware subsystem use. However, such measurements require instrumentation of the software system.
- The elementary energy consumption of components may exhibit nonlinearity and the individual models may not add up in a linear fashion.
- Power and energy profiling typically has an offline learning and calibration part. Offline power-measurement techniques are very accurate, but they require a laboratory environment. Online measurement can be used anywhere, but it has limited accuracy. Online techniques require either offline or online calibration.
- Energy optimization of smartphones and applications involve optimization decisions. An optimization decision is a system configuration change that is expected to improve a situation aiming for the optimal energy consumption of a device. Estimates and feedback from a power and energy profiler is an essential part of the optimization process.

Part II is organized so that after this overview chapter, we first look inside a smartphone and cover the different hardware subsystems of it in detail and discuss their energy-consumption characteristics in Chapter 7. Next in Chapter 8, we study the most important piece of software residing in the smartphone: the OS. We cover the mainstream OSs and their features relevant to the phone's energy consumption. We also take a look at a few research prototypes of OSs that specifically target energy-efficient operations. After these two chapters, we have covered enough background to look in depth at the main subjects of this part: power modeling and power/energy profilers. Chapters 9 and 10 are devoted to these topics.

References

[1] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proc. 9th Int. Con. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2011, pp. 335–348.

[2] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, “Accurate online power estimation and automatic battery behavior based power model generation for smartphones,” in *Proc. 8th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2010, pp. 105–114.

[3] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, “Fine-grained power modeling for smartphones using system call tracing,” in *Proc. 6th Conf. on Computer Systems*. New York, NY, USA: ACM, 2011, pp. 153–168. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966460

[4] J. Flinn and M. Satyanarayanan, “Energy-aware adaptation for mobile applications,” *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 48–63, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/319344.319155

[5] ——, “Powerscope: A tool for profiling the energy usage of mobile applications,” in *Proc. 2nd IEEE Workshop on Mobile Computer Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=520551.837522

7 Smartphone subsystems

To develop energy-efficient techniques, the first step is to understand how energy is consumed on a mobile device. A mobile device consists of hardware components, such as microprocessors, wireless network interfaces, storage, cameras and a touchscreen, and software running on top of these hardware components. Lower power serial busses facilitate the communication between the internal system components. These hardware components are the actual energy consumers.

Smartphone and mobile device power optimization happens on multiple levels:

- Silicon-level, in which the transistor capacitance and the chip design affect the energy efficiency. Higher capacitance requires the transistors to do more work.
- SoC-level, in which multiple power/voltage/clock domains can be used to support granular power management with the help of software. In addition, dynamic voltage and frequency scaling (DVFS) is used to dynamically adjust both the voltage and frequency to meet the given energy and performance level.
- Software-level, in which various power managers monitor and control the energy and power settings. A high-level framework is needed to perform system-wide tuning and optimization.

There are several choices that contribute critically to the overall efficiency of a mobile device, for instance from an energy-consumption viewpoint. They are the SoC including the CPU, display technology, communications technology, and the OS. The system-level power management is coordinated by the OS. In this chapter, we survey these crucial components and examine their energy consumption.

7.1 CPU and SoC

There are two fundamentally different basic philosophies in designing CPUs: CISC (Complicated Instruction Set Computer) and RISC (Reduced Instruction Set Computer) [1]. These two architectures differ because the instruction sets possible in the CPU are laid out using maximalistic and minimalistic principles, respectively. In this section, we discuss the choice of CPU and focus on the RISC-based ARM processors and SoCs that are frequently used in mobile devices. We then examine power-saving techniques, power states, and multicore and multiprocessor systems.

7.1.1 Choice of CPU

RISC designs are based on the notion of getting higher performance per machine code unit from a simplified (not complex) instruction because the simplicity enables faster execution. Thus, the RISC concept uses a small but optimized set of instructions, rather than a more specialized set of instructions used in many other architectures. Furthermore, RISC systems use memory with the load/store architecture. In RISC processors, memory is accessed through specific instructions and arithmetic/logical instructions do not access memory directly. In a CISC processor, single instructions can execute many low-level operations, for example load from memory, store to memory, and arithmetic operations. CISC processors may execute multi-step operations and addressing within single instructions. Consequently, CISC instruction sets are inherently compact and semantically rich and the work performed per machine code byte is therefore higher than in a RISC processor. This can be advantageous in cache-based implementations. Actually the defining feature of a CISC processor is not the number or complexity of instructions, but the use of memory accesses by the arithmetic instructions themselves. The RISC architecture is characterized by the load/store strategy instead.

Both architectures were used early in the development of computing hardware. For instance, CISC principles were used in the x86 processor family and RISC-type architectures in the ARM family. The resulting differences are profound: in x86 processors the instruction set is built with detailed and more complex instructions aiming to cater for many programming needs straight at the CPU level, whereas ARM processors use a significantly smaller set of simpler instructions.

7.1.2 ARM processors and SoCs

The ARM processor architecture was first developed by the British computer manufacturer Acorn Computers in the 1980s for their personal computers. Since then ARM has become the most important processor architecture for mobile devices. ARM processors are 32-bit RISC processors that have an optimized architecture that minimizes the transistor count and has a very low power consumption [2, 3]. The benefits of this simplified design are improved power usage and heat output and reduced cost as well as integrability with other components into low-power SoCs. The approach also lends itself well to multicore systems in which manufacturers obtain a higher core count with reduced cost and lower power requirements.

The core requirements for mobile processors include energy efficiency (in general that means low power consumption) and – because memory is at premium in small portable devices – small code size. The ARM processor architecture supports the 32-bit ARM and 16-bit Thumb instruction sets. The Thumb instruction set features a subset of the most commonly used 32-bit ARM instructions that have been compressed into 16-bit opcodes. The Thumb extension adds a Thumb decompressor in the instruction pipeline to a regular ARM processor. This generally makes the code size 30% smaller (with some inconsequential reduction in processing times). Low power consumption is achieved through RISC, accurate branch prediction, and optimized processor extensions, such as

DSPs. ARM SoCs use multiple power-saving techniques, such as fine-grained power and clock gating, and DVFS, which is discussed in the next subsection. Comparing the smallest ARM processors with corresponding CPUs in the x86 family we note that the latter are expected to operate at 100 W power levels whereas ARM Cortex-A9, for example, operates between 500 and 2000 mW.

It is therefore unsurprising that ARM processors have captured a sizable share of the combined mobile phone market and that they are used in Android phones and Apple's iOS devices. The CPU choice for mobile devices thus reflects different needs and leads to different hardware in use than, let us say, in desktop or laptop markets and particularly in the server system world where x86 processors prevail.

In Section 1.4.2 we examined a basic SoC design with the key hardware components. Today's SoC designs are more complicated than this basic design with multiple cores and additional co-processors. The ARM processors and SoCs include the ARM Cortex, Qualcomm Snapdragon, TI OMAP, nVidia Tegra, Marvell Xscale, and Apple's SoCs used in iPhones and iPads. The ARM family of processors consists of several different core designs that include the ARM8, ARM9, ARM11, Cortex-A8, Cortex-A9, and Cortex-A15 designs. These designs are licensed by manufacturers from ARM and then integrated with their SoCs with components such as RAM, GPU, and baseband processors. It is also possible for companies to acquire a license from ARM to design their own CPU cores using the ARM instruction set. These licensees include Apple, Qualcomm, Marvell, and Nvidia. The popular ARMv7 is used by the Cortex-A9 and Cortex-A15 designs from ARM as well as Qualcomm's Krait and Apple's A6. ARMv8 is the latest version of the processor architecture and the Apple A7 was the first implementation of the design.

The five popular SoCs used by smartphones today are:

- Qualcomm's Snapdragon which consists of four versions from S1 to S4. The S4 version is the latest and most powerful with Qualcomm's Krait CPU that supports up to four cores, Adreno GPU, LTE modem, and dedicated cores for multimedia processing. Figure 7.1 gives an overview of the Snapdragon SoC architecture that illustrates the modern SoC designs.
- Texas Instrument's OMAP (Open Media Applications Platform). The OMAP 3 series is single core featuring ARM Cortex-A8 and a PowerVR SGX530 GPU. The OMAP 4 series features dual-core ARM Cortex-A9 processors, a PowerVR SG54x GPUs, and two additional Cortex-M3 cores for lightweight tasks. The OMAP 5 is the next extension of the SoC family with two ARM Cortex-A15 cores, improved PowerVR GPU, and several dedicated cores. These SoCs feature the SmartReflex power-saving technology from TI.
- Samsung's Exynos SoCs are used in their smartphones and tablets. The latest version of Exynos is 5 Octa and it features a quad-core Cortex-A15 and a quad-core Cortex-A7, ARM Mali GPU, and auxiliary processors.
- Nvidia's Tegra SoCs are multicore and based on ARM cores with an ultra low-power (ULP) GeForce GPU. The latest version, Tegra 4, supports ARM-A15 cores in quad- or octa-core configurations.

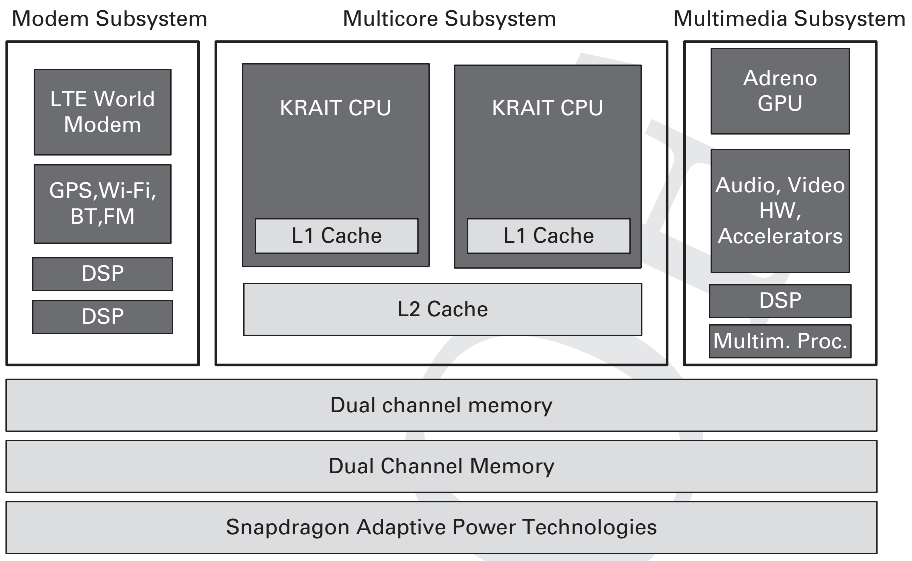


Figure 7.1 Overview of the Snapdragon SoC

- Apple's CPUs and SoCs are designed by Apple based on the ARM architecture. The Apple A6 is based on the ARMv7 dual-core CPU and integrated with a PowerVR GPU and image signal processor. The A7 used is based on an Apple-designed 64-bit ARMv8-A dual-core CPU, a GPU, and auxiliary processors.

Figure 7.2 illustrates the evolution of CPU and SoC performance with the Geekbench2[1] as reported by Nvidia [4]. The performance of mobile processors has improved to match desktop processors, such as the Intel Core-i5. The number of CPU cores has increased to a maximum of eight and at the same time GPUs have evolved. The number of auxiliary co-processors has also increased.

Typically, a state-of-the-art SoC has each core in its own power domain and the voltage and clock speed of a core can be set independently. CPU caches are used to reduce the number of off-chip memory accesses. Caches store most frequently used data in on-chip memory supporting fast access. Typically, each core has its own instruction and data caches (L1 cache). The cores can also share a common larger cache (L2 cache).

In addition to multiple CPU cores, an additional low-power CPU core has been proposed to address low-intensity background tasks [5]. The Nvidias Tegra 4 features such a battery saver core that is optimized for low power. When the quad-core main CPU is not needed, the system switches to the battery saver core and completely switches off the main CPU. The battery saver core has its own L2 cache and the L2 cache used by the other cores can be power gated.

[1] `http://www.primatelabs.com/geekbench/` accessed January 6, 2014.

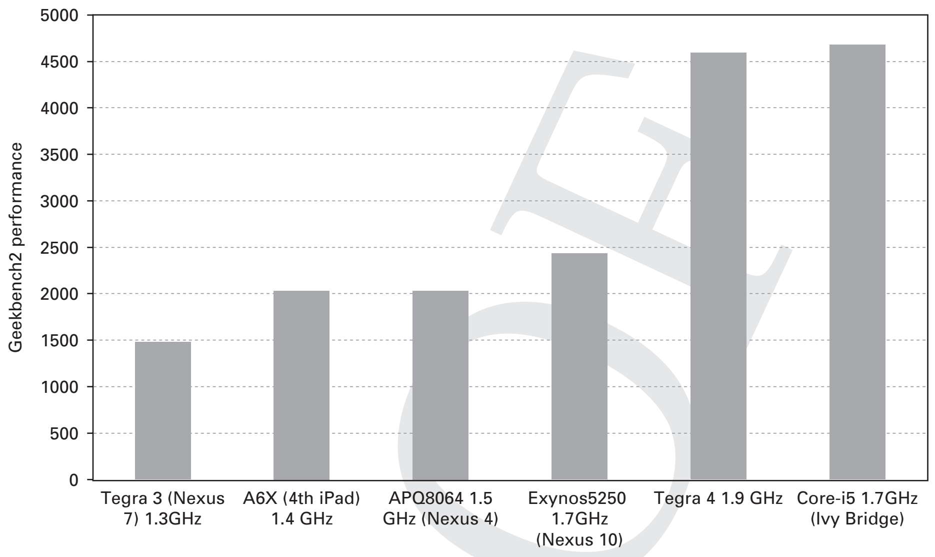


Figure 7.2 Comparison of mobile CPU and SoC performance

The ARM-based SoCs use dedicated controllers for realizing different functions making them very different from the desktop PCs that rely on a single CPU for most of the work. The benefit of dedicated controllers is their efficiency compared to software-based implementations. The dedicated chips and circuits can carry out the tasks more efficiently and with fewer cycles than a software implementation. Indeed, it is not uncommon to first have software-based implementation of a process or algorithm, such as audio processing or graphics generation, that is then implemented in hardware for increasing efficiency.

7.1.3 CPU speed

The internal speed of the CPU depends mainly on two things: the time spent waiting for data or for instructions and the parallelism of the processor work. The less time spent waiting and the more tasks executed simultaneously, the faster the overall processing. However, many processors have been designed with an in-order execution of instruction stream in mind. In SoC architectures the available memory bandwidth is smaller than in CPU's used for example in laptops: the memory is slower, the bus is slower, and parallel access is not possible for a satisfactory number of memory chips. All these features reduce the true memory bandwidth. Furthermore, we must remember that in CPUs using a strongly serial processing model every cache miss will stall the pipeline and halt the thread, possibly forcing the CPU to switch processes. SoC CPUs normally also use the same external memory as the GPUs and this sharing of vital resources tends to harm the speed of both chips.

Alternatively, we could make the instruction stream faster by increasing the CPU clock frequency. The fundamental methods of doing this have their benefits but also their drawbacks. There are three factors that determine the maximum achievable clock frequency and which are feasible in creating faster chips:

1. Increase the density of transistor elements on the chip. Smaller transistors are faster and also more power efficient when active. The disadvantage is that they consume more power when idle.
2. Increase the voltage of the chip: this enables a higher switching speed. Here the drawback is also increased power consumption.
3. Increase the length of the pipeline. With long pipelines, work can be divided into shorter steps and the processing becomes faster. Unfortunately, power consumption is increased.

7.1.4 Power saving techniques

Dynamic Power Management (DPM) is a design methodology for energy and power management of dynamically reconfiguring systems [6]. The goal for a DPM system is to provide the requested services and performance with a minimum power consumption. To minimize power consumption, the system needs to find the minimum number of active components or the parameters of such components that result in minimized power draw. The key insight and fundamental assumption of DPM is that the system workload is not uniform, but varies as a function of time. The more the system knows about the workload and its future behavior, the more effective the power-saving plan that can be designed and implemented. A central component in any system implementing DPM is a power manager that monitors and controls the system. The power manager follows a power-management policy that determines the parameters that are monitored and controlled [7].

Recalling Eq. (2.7) from Chapter 2,

$$P_{switch} = \alpha \times C \times V^2 \times f, \tag{7.1}$$

we observe that it is possible to tune the operating voltage and frequency to achieve energy savings with the tradeoff in task-execution time.

This observation is at the heart of the two well-known active system power optimization techniques: DVFS (Dynamic Voltage and Frequency Scaling) [8, 9, 10] and DPS (Dynamic Power Switching) [11]. DVFS minimizes power needs by trying to reduce the operating frequency and the core voltage to levels sufficient to execute current tasks but with no excess resources left. This is important in situations where the CPU is not idle but not fully occupied either, typically when a background thread tries to force the CPU to keep its resources (frequency and voltage) at their highest level without using them fully. DPS detects the true need of resource use in a CPU and, if there are no current computational tasks, forces the CPU into the minimal power state. This often happens when a CPU is waiting for completion of a DMA (Direct Memory Access) instruction.

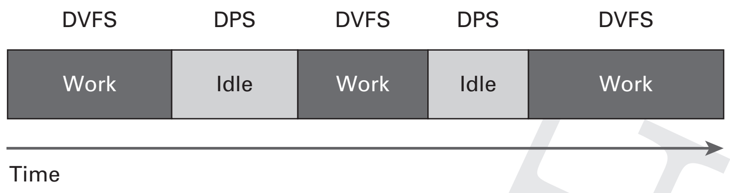


Figure 7.3 Example of DVFS and DPS

DPS and DVFS techniques differ because DPS mainly lowers leakage. DPS may reduce the clock and core voltage to zero, thereby eliminating power needs altogether, but we must note that zero power necessitates saving the CPU state on shutdown and loading it again when power is applied. This requires extra power in itself and therefore the DPS main parameter, the maximum allowed idling time, must reflect the true needs of the instruction stream to guarantee that the mechanism offers real power advantages over the simplest method of plain idling. Figure 7.3 gives an example of these two complementary techniques. In the example, DVFS is used when there is an active load on the system, and DPS is used when the system is idle.

As an example of DPS, we have the Dynamic Power Management (DPM) in Linux and Android. This is typically combined with DVFS to reduce the voltage and clock frequency of components to save energy. System-level power management policies govern the implementation of DPM and DVFS. After investigating the P- and C-states in more detail, we will take the Linux CPU frequency subsystem as an example.

Energy and thermal management at the SoC level requires hardware probes at certain locations to obtain accurate temperature information. In addition to hardware probes, software monitors can also be used to track the hardware sensors and factor in their distance and runtime characteristics. This can help hardware designers to reduce the number of hardware probes and thus save silicon area [12].

Processor performance monitor counters, adaptive feedback controllers, and thermal, delay, or wear out monitors have been considered for SoC-level energy and thermal system management [12]. Performance counters are useful in determining the functional use of components that can then be correlated with temperature and energy usage.

Most of the power- and temperature-aware monitoring approaches involve both hardware and software components. The software running at the OS or Virtual Machine Monitor level then uses thermal and energy consumption data for setting the DVFS and other parameters for triggering task migration.

7.1.5 P- and C-states in CPU

At the processor level we can manage power consumption with two different strategies:

1. Control of the CPU performance states using frequency and core voltage (P-states).
2. Use of processor operating states (C-states).

The ACPI specification [13] is the industry-standard solution for power management used by the desktop and laptop industry. We can take ACPI as an example of a power-management standard that is based on P- and C-states. Current smartphones use these states as well, but they are not based on ACPI.

The processor P-states are for plain power control. Generally there are at least two P-states available:

1. P0 that is the normal state of high voltage and adequate clock frequency.
2. P1 that uses both low core voltage and low clock frequency. This is for situations where the CPU is often idle.

An increasing P-state number means lowering the clock frequency and core voltage in a running processor. The set of P-states is CPU-dependent and differences exist between CPU types. A P-set always starts with P0, which means the highest performance with the highest voltage and core frequency. The combinations of frequency/voltage used for different states vary between implementations but both the power consumption and the performance will always decrease with the growing P-number. Power saving uses the clock frequency and core voltage as parameters but they are not fully independent. If the CPU frequency is lowered, the voltage can be lowered simultaneously without any problems in efficiency. This is fortuitous because a greater performance gain per unit of used power is thus achieved quite easily.

Power saving can be also implemented with several C-states, which are independent of P-states and cumulative. The aim is to turn off all those parts that are not necessary when the CPU is idle. The use of C-states is a recent trend, used originally in laptops but also increasingly in servers.

The set of C-states implemented in a CPU is numbered from 0 upwards and C0 is the active state, that is the CPU is executing an instruction (C1 mode is also always defined). All other states or sleep modes are for when the CPU is idle, the higher the number, the more components have been shut down to conserve power. The drawback in C-states is that the time required to go back to the zero-state grows when the depth of the processor sleep increases (higher latency). The latency level can be controlled in some C-states (depends on the processor) by using submodes, which facilitate choosing a suitable latency time while preserving the power-saving feature.

Figure 7.4 illustrates the P- and C-states. P-states are used to transition between the highest power state to the idle mode typically with the voltage and frequency scaling technique. C-states are used to shutdown idle components to achieve power savings.

In the following we list the typical C-states [14, 13]:

- C0: The processor is unrestricted and all components are on.
- C1: The processor is partially stopped. Software may stop the CPU internal clocks but the bus interface and APIC (Advanced Programmable Interrupt Controller) run normally.
- C2: The processor clock is set unconditionally into the stop clock state. Hardware stops the main internal clocks. All software-visible states are maintained but the waking-up requires interrupts and takes additional time.

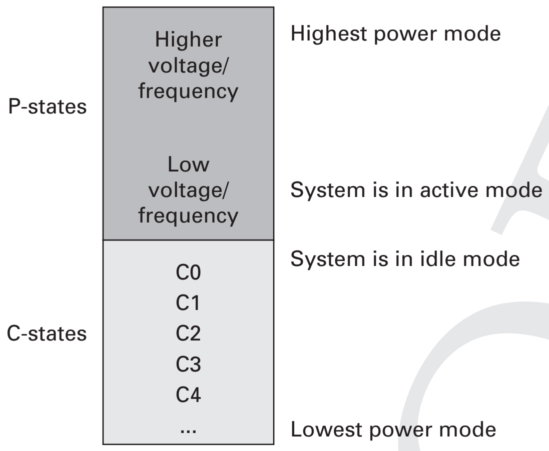


Figure 7.4 P-states and C-states

- C3: CPU is powered down into the sleep state. All internal clocks are unconditionally stopped and the cache is not updated. In different implementations the time of wake-up through interrupts may vary somewhat.
- C4: The processor is disconnected from the system causing deep sleep.

The difference between C-states and P-states is one of operational level and can be summarized as follows:

- In C-states starting from C1 the processor is idle but partially in use.
- P-state is an operational concept and defined solely by the clock frequency and core voltage.

Controlling the CPU frequency dynamically at runtime requires a specific infrastructure in the processor for setting the power policy, be it static or dynamic. This infrastructure typically has the following components:

- A subsystem for connecting the low-level technologies and policies at the higher level.
- A system of in-kernel policy governors. They are essentially pre-configured power schemes with the ability to modify the clock frequency according to required needs. Typically these governors use the P-states to change frequencies for lower power consumption. They will switch between clock frequencies, on the basis of the current CPU use level trying to save power while not unduly losing performance. The governors are tunable allowing some customizing of the frequency scaling.
- Drivers implementing the technology in a CPU-specific way.

As an example of smartphone C-states, we consider the Nexus 4 smartphone based on the Snapdragon S4 SoC. Table 7.1 presents the four CPU C-states of the Nexus 4

Table 7.1. Nexus 4 smartphone C-states

State	Idle Power (mW)
C0	433
C1	390
C2	330
C3	200
Without idle states	1060

and their power draw [15]. In the C0 state for Nexus 4, the CPU clocks are disabled but other components are active. Higher C-states progressively shut down more CPU components resulting in more efficient idle-mode operation. The overall power saving is drastic, from 1060 mW to 433 mW in C0 to 200 mW in C3.

7.1.6 Example: Linux CPU frequency subsystem

As an example of DVFS and DPS, we consider the Linux CPU frequency subsystem that has supported dynamic processor frequencies since the 2.6.0 Linux kernel [16]. The CPU frequency subsystem uses governors and daemons for implementing a static or dynamic power-management policy. In the dynamic setting, the governors tune the CPU frequencies based on the CPU use. The subsystem consists of five governors that set the CPU frequency and change it based on feedback from the system or the user.

The five governors are:

- Performance governor that gives the highest CPU frequency and performance. This governor statically sets the highest frequency value and allows the tuning of this highest value.
- Powersave governor that sets the lowest CPU frequency and system speed. In a similar manner to the performance governor, this governor statically sets the lowest frequency value and allows tuning to this lowest value. This governor does not result in significant power savings, because it causes tasks to run for longer and thus the CPU cannot enter the low-power C-states.
- Userspace governor that allows the CPU frequency to be set manually. This governor works together with the userspace processor frequency daemons that provide the processor frequency values. The component can be used to implement custom power policies.
- Ondemand governor is an in-kernel governor to dynamically set the CPU frequency based on CPU use. The governor monitors the CPU use and when a certain threshold is exceeded, sets the frequency to the highest possible. Similarly, when the use is below the threshold, the CPU frequency is dropped to the level below the current level. Thus the frequency is set based on the CPU use and the threshold value. The frequency range, the threshold, and the usage testing rate can be tuned.
- Conservative governor is similar to the ondemand governor, but allows a more gradual increase of the power consumption. The governor adjusts the frequency based on the

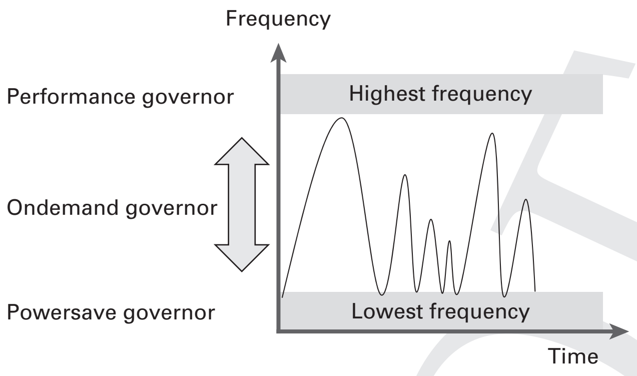


Figure 7.5 Linux frequency governors

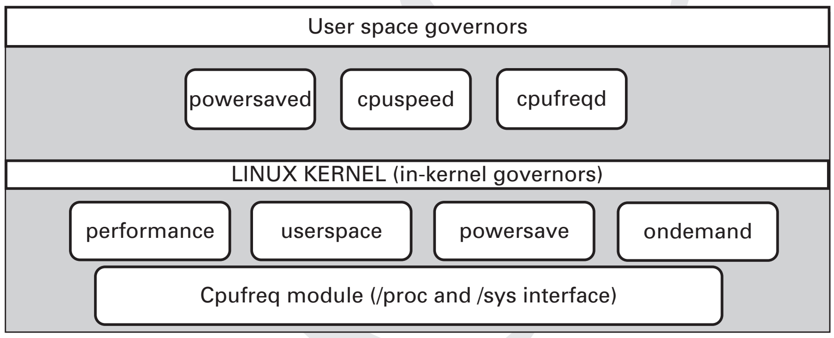


Figure 7.6 Overview of the Linux CPU Frequency subsystem

CPU use step-by-step and does not jump to the highest frequency immediately when a threshold is exceeded as in the ondemand governor.

Figure 7.5 illustrates the governors. The performance governor always sets the highest CPU frequency. The powersave governor, very similarly, always sets the lowest possible CPU frequency. The ondemand governor and its variants set the CPU frequency dynamically based on the CPU use and the thresholds.

Figure 7.6 illustrates the CPU frequency subsystem with the in-kernel and userspace governors. The CPUfreq module provides an interface to the CPU-specific frequency control techniques and policies. The kernel governors are responsible for changing the CPU frequency based on the given policy, such as the CPU use threshold in the ondemand governor. The userspace governor exports the CPUfreq data to the userspace programs through the /sys file system and allows userspace daemons to control the CPU frequency. Userspace programs include the powersave daemon for controlling many aspects of power saving, cpuspeed that monitors the system's idle percentage and sets the CPU frequency and voltage accordingly, and the CPUfreqd that is a small userspace daemon that sets the CPU frequency and voltage based on the battery level, temperature, running programs, CPU, etc.

7.1.7 Hot-plugging

Hot-plugging is the process of adding or removing components on a live system without shutting down the system first. CPU hot-plugging can be used to save power on mobile devices. For example, on an octa-core system some of the cores can be shutdown and then later activated when they are needed. On a running system, this means that any tasks running on the cores that are to be shutdown need to be migrated to cores that are kept active. Hot-plugging requires a scheduler that monitors the load on the cores and then implements power-saving actions and migrates tasks before shutting down cores. This technique saves power; however, when more CPU power is needed it takes some time to active more cores and migrate tasks to them. In addition, when a task is migrated to another core there can be a cache penalty as the target core will not have a cache for the incoming task.

Hot-swapping and task migration are supported by many current multi-processor SoC architectures, such as ARM's big.LITTLE, Nvidia's Tegra, and Qualcomm's Krait. The architectural details, such as cache management, differ between the architectures. In the next section, we examine ARM's big.LITTLE architecture as an example system that allows hot-plugging and task placement over multiple cores.

7.1.8 ARM's big.LITTLE architecture

The ARM big.LITTLE[2] is a computing architecture that combines slow and low-power processor cores with faster and more power demanding cores [17]. The aim of the architecture is to realize a multi-core processor that can dynamically adjust to the computing requirements. This architecture is used, for example, by the Samsung Galaxy Note 3 and S4 smartphones (Exynos 5 Octa).

The architecture supports three ways of arranging cores depending on the Linux kernel scheduler implementation:

- Clustered model, in which the OS scheduler observes one of the two processor clusters, and the scheduler transitions between the clusters based on the observed load.
- In-kernel switcher pairs a more powerful core with a less powerful core with the option of having many identical pairs on a chip. Each pair can be seen to be a virtual core with only one of the constituent cores active. The active core is selected based on the load. More elaborate configurations have also been proposed.
- Heterogeneous multiprocessing (MP) enables the use of all physical cores simultaneously. High priority or computationally demanding threads are run by the powerful cores, while low priority or less demanding threads are run on the less powerful cores.

ARM's big.LITTLE architecture extends DVFS with CPU migration. In this process, the DVFS algorithm monitors the load on each CPU and then migrates tasks between the higher and lower performance CPUs. When the load is low it is handled by the lower

[2] `http://www.thinkbiglittle.com` accessed January 6, 2014.

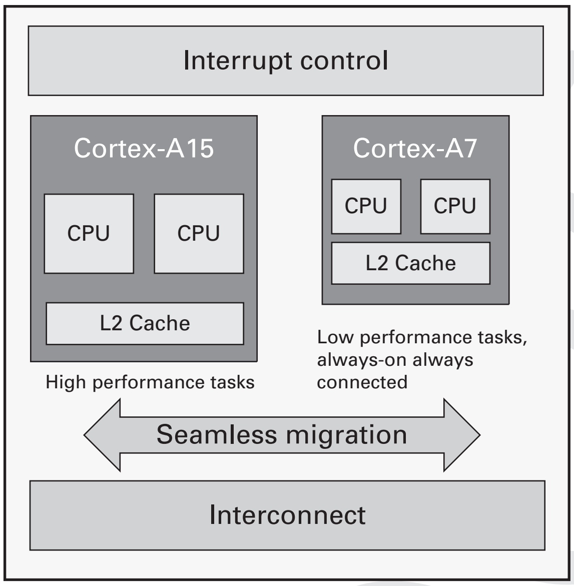


Figure 7.7 Overview of the big.LITTLE architecture

performance CPU. When the load is high it is transferred to the higher performance CPU. If a processor becomes unused, it can be powered down. Moreover, if a cluster of processors becomes inactive, it can be powered down as well.

Figure 7.7 illustrates the system with Cortex-15 and Cortex-A7. The framework allows seamless migration of tasks across the processor cores. The Cortex-A15–Cortex-A7 system is designed to migrate tasks between the processor clusters in less than 20 microseconds with 1 GHz processors [17]. In a larger system, the processor core pair can be seen as a virtual core. Depending on the operating mode, for each virtual core the Linux scheduler can then either choose the active physical core or use all the physical cores simultaneously (MP mode).

An energy-efficiency comparison of Cortex-A15 and Cortex-A7 indicates significant power savings with a variety of benchmarks. For example, the Dhrystone benchmark gives an energy-efficiency benefit of 3.5x for the A7 with a performance benefit of 1.9x for the A15. This encourages the use of the slower processor for lightweight tasks [17].

7.1.9 Graphics processing unit (GPU)

In modern smartphone designs, graphics processing is typically offloaded to the GPU that has a high-performance graphics processing pipeline. An SoC thus consists of one or more CPUs, one or more GPUs, a DSP, and application-specific accelerators. Typically GPUs are used to realize 3D rendering in games and graphics-intensive applications; however, the latest GPUs, such as Nvidia's project Logan, also allow general purpose computing with the GPU. The performance of mobile GPUs has dramatically increased in recent years. The current generation of mobile GPUs is on a par with the graphics power of game consoles, such as Sony's PlayStation 3 (NVidia GF7800) and Microsoft's Xbox 360 (ATI R500) [4].

The four well-known mobile GPUs are:

- Adreno GPU in the Qualcomm's Snapdragon line of SoCs.
- PowerVR GPU used in TI's OMAP line of SoCs.
- Mali in the ARM architecture.
- GeForce ULP (ultra low-power) in the Tegra line of SoCs from Nvidia.

Rather than focusing on performance alone, mobile GPUs emphasize low power consumption. An embedded GPU shares the system bus with the other processors and components, and can access the external memory of the device. The memory access is a bottleneck for mobile GPUs and their desktop counterparts have dedicated memory to ensure a high bandwidth to memory.

Energy-efficient operation requires that the number of memory transactions between the GPU and the external memory is as low as possible. Mobile GPUs use various on-chip caching techniques to reduce the memory traffic [18]. For example, the Qualcomm Snapdragon Adreno GPU has two modes for rendering: deferred and direct. The former breaks the display into smaller tiles and renders them independently. Smaller tiles can be stored in the GPU memory thus reducing GPU memory traffic to the external memory [19]. The latter mode computes the scene and then renders the pixels directly to the screen. The Adreno GPU can decide at runtime whether to use deferred or direct rendering depending on which one gives the best power efficiency. The PowerVR GPU uses a similar tiling technique in which one tile is rendered at a time with all the necessary data for the rendering kept in the on-chip memory. The Tegra GeForce has on-chip caches to reduce the number of memory transactions during rendering. Other advanced optimization techniques used in the Tegra 4 GPU include multiple levels of clock gating, display request grouping, and DVFS [5].

Modern desktop GPUs have hundreds of cores and the number of cores for mobile GPUs will also soon be over several hundred. Some applications can use all the cores and for these applications the performance per watt increases linearly. For applications that cannot use all the cores, the performance per watt becomes saturated. The optimal number of GPU cores is the one that gives the highest performance per watt [20].

An empirical power model has been proposed for GPUs that can be used to predict the optimal number of active processors for a given application [20]. This model is based on execution times. With the predicted number of active cores, up to 22% of runtime GPU energy consumption can be saved.

Modern smartphone operating systems support GPUs for 2D and 3D graphics. The OpenGL ES rendering API [21] is the key graphics programming interface for mobile devices with programmable GPUs. OpenGL ES is a subset of the widely used OpenGL standard for desktop 2D and 3D graphics. To take advantage of the benefits offered by a GPU, the developers need to implement the applications with the API and the primitive graphics operations and write the necessary shader programs.

Some GPUs can be used for generic computations, for example signal processing and face recognition. For example, a Gabor face feature extraction algorithm was implemented with the Tegra GPU and OpenGL ES and the shader language. The resulting GPU-based algorithm achieved a 4.25 times faster speed compared to the CPU-based version [18]. This indicates that a mobile GPU can significantly improve the performance of vision tasks while still requiring low power. The GPU-supported face feature extraction was observed to result in a 3.98 times lower energy consumption than the CPU-based counterpart. While the power draw increased due to the intensive use of the GPU, the significant improvement in the running time of the algorithm resulted in the low energy consumption. The overall face recognition application benefited from the GPU-based algorithm with 1.85 times improvement in the running time and 1.83 times lower energy consumption.

GPUs are an integral part of smartphone SoCs and are essential for graphics-heavy applications, such as games. Certain generic processing tasks, such as face recognition,can also be offloaded to GPUs. Near-future emerging applications include augmented reality that requires intensive image processing for object and face recognition [4].

7.1.10 Modeling the CPU

Next, we outline the development of simple power models for the smartphone SoC and CPU based on the usage and linear regression. The development of our simple model proceeds in the following phases:

- Design of training phase with different CPU loads. The loads should be realistic and reflect the real-life workloads on the device.
- Power measurement of the training loads on the smartphone. An external power monitor tool is typically used for high-accuracy measurements.
- Creation of a power model for the CPU energy consumption.

A power model can be derived in a similar manner for various hardware components including the display, cellular network, Wi-Fi, and GPS. As mentioned, linear regression is frequently used, but it may result in a significant estimation error if the underlying phenomena are nonlinear. A linear model may also abstract important low-level details relating to power states and transitions between them. A power model of the whole system can then be built on top of the component power models.

After the model has been built, it can be used to estimate the power consumption of the CPU and make power-management decisions. Power management requires that we have control points, options to choose, to modify the system behavior.

7.1.11 Processor power model based on counters

Isci and Martanosi have proposed a power model for CPUs based on performance counters [22, 12]. They correlate hardware performance counters and system logs with total power measurements with an external power monitor to obtain a fine-grained view of energy consumption of the CPU. A similar approach can be used to model GPUs [20]. The monitored CPU components include the bus control, L1 and L2 caches, buffers, integer execution, floating point execution, and queues. The performance counter metrics are mapped to the CPU components.

The model collects architectural statistics, such as cycles per instruction, memory references, cache misses, and on-chip communication details, together with the application-level details to optimize performance and meet the desired energy and temperature goals. Given a CPU with N components, the ith component denoted by C_i, the model is based on component access rates given by a performance counter or a combination of performance counters. For CPU component C_i, the maximum power, $MaxPower(C_i)$ and $ArchitecturalScaling(C_i)$ are heuristics and are estimated empirically.

$$\begin{aligned} P(C_i) = {} & AccessRate(C_i) \times ArchitecturalScaling(C_i) \times MaxPower(C_i) \\ & + NonGatedClockPower(C_i). \end{aligned} \tag{7.2}$$

The total power of the CPU is given by the following equation:

$$P_{total} = \sum_{i=1}^{N} P(C_i) + Idle\ power. \tag{7.3}$$

7.1.12 Single core regression model

Assuming a linear relationship between power consumption and CPU load, the linear regression model would be the following:

$$P_{cpu} = a \times U_{cpu} + b, \tag{7.4}$$

where a and b are constants, and U_{cpu} is the CPU use. The latter term indicates the power consumption of the CPU. This simple model does not take the DVFS into account, but it can be extended to include the voltage and frequency scaling.

The power model can then be used in power estimation to predict the power consumption without power measurements. Following our example, given that CPU use is 20% for a give duration, T, we can determine the energy consumption using $E_{total} = (a20 + b)T$. The energy consumption of an application that uses the CPU in a dynamic manner can be determined using

$$E_{total} = \sum_{i} P^{i}_{cpu} \times \Delta T, \tag{7.5}$$

where P^{i}_{cpu} is the i-th measurement of CPU use and ΔT is the time interval of the measurement.

7.1.13 Single core regression model with DVFS

DVFS can significantly improve the energy efficiency of the CPU. The obtained benefit depends on the idle power consumption of the CPU and the execution time of the tasks with different CPU voltages and frequency levels. The effect of DVFS can be examined using the following simple equation [23]:

$$E = P \times t + P_{idle} \times (t_{max} - t), \tag{7.6}$$

where E gives the total energy of the workload, P is the average power over the workload, t is the execution time of the workload, P_{idle} is the idle power of the CPU, and t_{max} is the maximum running time of the workload over all frequencies.

7.1.14 Multicore regression model

Yifan Zhang et al. studied the power modeling of multicore smartphone CPUs and they have identified that the traditional frequency- and use, based regression techniques are prone to errors in the multicore setting [15]. Based on this observation they developed a new regression-based power model for multicore CPUs based on the time spent in C-states. This model builds on a model of a single core working at frequency, f, that is the given by the following equation [15]:

$$P_{core} = \sum_i \beta_{C_i} \times WED_{C_i} + \beta_U \times U + c, \tag{7.7}$$

where WED_{C_i} is the weighted average entry duration for idle state C_i, β_{C_i} and β_U are coefficients of WED_{C_i}, U is the usage rate, and c is a constant. A power model is created by obtaining the coefficients by linear regression on the training data with WED_{C_i} and U, and the associated P_{core}.

The multicore model extends the above single-core model and it is given by [15]:

$$P_{CPU} = P_{BL,N_C} + \sum_i^{N_C} P_{\Delta,core,U_i,f_i}, \tag{7.8}$$

where N_C is the number of cores, P_{BL,N_C} is the baseline CPU power with N_C cores, and $P_{\Delta,core,U_i,f_i}$ is the power increment due to core i when running at frequency, f_i, with usage of U_i. The term $P_{\Delta,core,U_i,f_i}$ can be predicted using the single-core model ($N_C = 1$) and with the measurement of the constant P_{BL,N_C}.

7.2 Display

The display is definitely one of the most energy-consuming components in a modern mobile phone, especially now, when smartphones have started using touch-sensitive full HD screens. After the CPU, the screen is therefore the next robust challenge for energy efficiency. The trend is for users to watch online videos with their phones on the new emerging mobile networks and this increases the screen energy needs, draining the batteries. It has been shown that the display is among the most power-hungry smartphone hardware components with a 400 mW (LCD panel, touchscreen, backlight, and graphics accelerator) power draw, second to the GSM module with a 700–800 mW power draw at full capacity.

7.2.1 Display technologies

There are a few technologies used today for displays of mobile devices: TFT liquid crystal display (LCD), reflective LCD, and organic light-emitting diode (OLED). Both TFT LCDs and OLED displays are used in the current smartphones and tablet devices.

LCD pixels are based on their filtering ability, which can be changed by the panel hardware. Light from the light source is pixel-wise filtered according to the parameter values in the display memory. The opacity of the LCD pixel is adjusted so that smaller values increase the pixel opacity. Thus, the maximum value a pixel can attain is the strength of the backlight. The reflective LCD does not use a backlight, resulting in a relatively small power draw, but the screens do not work well in dimly lit situations. In TFT LCD displays, the LCD panel is enhanced with a backlight, a light source giving illumination to the panel from behind. Modern screens use LED arrays to generate the backlighting, the older method is CCLF (cold cathode fluorescent lamps).

TFT LCD technology comes in three varieties: transmissive, reflective, and transflective.

1. In transmissive displays the backpanel lighting is used for pixel illuminating. Generally this technology gives a very good quality of display and can be applied in widely variable ambient light environments. They are usable in typical room lighting but also in complete darkness, while the contrast remains high and the colors vivid. The worst case for these panels occurs in full sunlight where they may become unreadable because the ambient light chokes the backlight.
2. In reflective LCDs there is no backlight at all, instead the available ambient light is used. Therefore the reflective LCD is much more power-efficient than backlit displays. Furthermore they operate quite reasonably in brightly lit outdoor environments, which gives them an advantage over transmissive displays. The downside is the frontlight normally necessary for dimly lit situations.
3. In a transflective LCD, backlight is used but there is also a reflective mirror. This technique aims to combine the good points of transmissive and reflective panels and guarantee an adequate performance independent of ambient lighting.

OLED displays come in different families, passive-matrix (PMOLED) and active-matrix addressing schemes (AMOLED). AMOLED uses a thin-film transistor backplane to switch each pixel on or off resulting in a higher resolution. In contrast to transmissive LCD displays, OLED displays do not need a backlight. Most of the smartphones currently on the market use AMOLED displays with the exception of iPhones that use the LCD technology.

7.2.2 Power consumption of smartphone displays

All the LCD technologies require an external light source, backlight or frontlight. The light source is by far the greatest power sink in these screens because the LCD filter needs low power when compared to the light sources. Hence, the power consumption of a reflective LCD can be much smaller than a transmissive one, and, for the same reason, a transflective LCD can also demonstrate an improved power efficiency compared to a purely transmissive LCD, especially when exposed to bright ambient light. The power draw of a transmissive LCD display is mostly characterized by the strength of the backlight, which is usually an adjustable parameter in smartphones, and the displayed color scheme has little relevance.

In contrast to LCD displays, the power draw of OLED displays is not just a simple function of the strength of the backlight; it depends on the colors displayed. AMOLED displays have been power modeled as a linear function of a standard RGB color scheme [24], which basically means that the brighter the color, the more power is drawn, and that the white screen draws most power. However, recent research work has shown that a linear relationship may not accurately capture the power-consumption dynamics of AMOLED displays of modern smartphones, suggesting that the exact relationship between power and pixel colors of the display are more complex [25, 26].

Box 7.1 Smartphone screen scrolling and energy

An interesting fact related to the power draw of smartphone displays is that the scrolling operation has been reported to consume a significant amount of energy on smartphones. This phenomenon stems from the observation that the highest possible frame rate is used when scrolling, which may result in up to 50% of the total power consumption of the smartphone. The energy consumption of scrolling can be mitigated by setting the frame rate in an adaptive manner to provide a desirable balance between user experience and power draw. Results with an adaptive scheme indicate that significant power savings, up to 58% of CPU and 34% of the total energy consumption, are possible for the scrolling operation [27].

The dependency of the power draw of an OLED display on the colors shown naturally gives rise to optimization schemes where the color scheme is chosen in such a way that power draw is minimized. An example of such a solution is presented in [24], where the web browser of a smartphone is instrumented to perform color transformations on the fly. These kinds of scheme can be parametrized so that the visible effect of the transformations can be controlled by the user. For example, the user may want to specify a maximal degree of distortion from the original color scheme caused by the transformations, which may of course also limit the achievable energy savings.

7.3 Wireless network interfaces

A mobile device relies on its communication hardware and software, and the trend in recent years has been to intensify and increase all communication-related features in the devices. This means that the cost in power and energy created by communication is growing and this is creating an urgent need to make chips faster and less power-consuming.

Wireless communication differs in many ways from the wired transfer of information. A higher bit rate in a wireless link means generally decreased energy consumption per gigabyte, but communication theory shows that this requires a higher. This again acts contrariwise, increasing the power needs of transmission. So, power-saving measures are also important with wireless communication.

Overall, wireless communication is one of the biggest energy consumers in a smartphone during typical usage. However, the energy consumption depends highly on the type of access network used and the workload offered, that is the specific traffic patterns caused by the applications being used. Indeed, smartphones today have several wireless network interfaces (WNI) that implement different kinds of wireless communication technology. We next take a detailed look at the characteristics of the power draw of the three most used wireless technologies present in every modern smartphone: Wi-Fi, Bluetooth, and cellular network interfaces (3G and LTE).

7.3.1 Wi-Fi

Basics

Wireless local area network (WLAN) equipment based on the IEEE 802.11 standard [28] are prevalent in the market, including practically every smartphone model out there. Wi-Fi is the often more familiar name used for this technology, although strictly speaking it is a trademark of the Wi-Fi Alliance that provides certification for those devices based on the 802.11 standard and having required level of interoperability.

Wi-Fi can be used in two different modes: adhoc and infrastructure. Adhoc is rarely used nowadays because Wi-Fi infrastructure, that is access points (APs), are so extensively deployed for indoor coverage. In the infrastructure mode, the Wi-Fi client, such as a smartphone, first needs to discover an AP and then associate with it before it can begin communicating with the Internet. The discovery part can be done in two basic ways. The first one is passive, where the client scans, that is, listens, to the different Wi-Fi channels for any incoming beacon frames which are sent by APs periodically to advertise their presence. After receiving such a beacon frame, the client device may begin association. The second approach is active, where the client broadcasts special frames to which the APs respond. In this way, the discovery may be performed more quickly because the client does not need to wait for the AP to transmit its beacon frame.

Data is transmitted in frames and it is governed by the Medium Access Control (MAC) protocol in Wi-Fi. The medium access in Wi-Fi is random access, meaning that there is no centralized control and scheduling of the transmission of different clients. The MAC protocol used is called CSMA/CA, which stands for Carrier Sense Multiple Access with Collision Avoidance. The basic idea is that any client can transmit at any time but it needs to first listen for the channel and only transmit if no one else is transmitting at the moment. Even so, collisions, that is simultaneous transmissions that render both signals undecodable, may occur. To detect collisions, Wi-Fi uses acknowledgments, and when there is a collision, each client chooses a random backoff time during which it is not allowed to transmit. The randomness makes it possible to avoid synchronization between transmitting clients, which would lead to a deadlock situation. For a more comprehensive coverage of Wi-Fi, we refer the reader to [29].

Energy consumption

We divide the discussion of the energy consumption of Wi-Fi into two parts. First we look at the energy consumed during the discovery and association stage during which we say that the client is in a non-connected mode. The AP discovery is the most energy-consuming part of this mode because scanning, that is actively listening to a radio channel, draws almost as much power as transmitting. The exact amount of energy spent in the discovery phase depends on many factors but its duration is the key metric. Active discovery is typically faster than passive discovery and, therefore, also more energy efficient. The frequency at which new APs are scanned for is another factor that greatly affects the overall energy consumption. In Chapter 13, we study how the overall discovery energy can be reduced. The key insight in those solutions is to try to deduce in less energy-consuming ways an opportune moment to begin scanning, that is when a new AP is expected to be in range.

Once connected, the actual data transfer takes place. A WNI typically has several operating modes, each of which corresponds to a specific activity. Wi-Fi has three basic modes which are transmit, receive, and idle, which correspond to the transmitting, receiving, and listening for incoming frames. Each of the modes correspond to a specific power state.

The 802.11 standard defines a power-saving mechanism (PSM) [28]. It introduces a possibility for the client to enter sleep mode when it is not actively receiving or transmitting data. In this mode, the radio is mostly powered off and the WNI draws typically an order of magnitude less power than the idle mode. In sleep mode, the radio is only powered on periodically to receive a beacon frame from the AP intervals (e.g. 100 ms) that notifies the client of any incoming packets. The AP with which the client is associated must be notified by the client before it goes to sleep so that the AP knows to buffer any incoming packets that arrive between beacons and are destined for that client. To reduce the negative performance impact of PSM, an adaptive version of it, also known as PSM Adaptive, has been widely adopted in commercial products. In PSM Adaptive, the network interface stays in the idle mode for a fixed period of time, such as 100 ms, before going to sleep. The length of this period is also sometimes called the PSM timeout and its default value varies from device to device. Figure 7.8 illustrates the state diagram of Wi-Fi after it is associated with an AP. CAM is enabled when PSM Adaptive is disabled, and vice versa. P_T, P_R, P_I, and P_S represent the power consumption for transmit, receive, idle, and sleep modes.

Besides the sleep mode, which is designed for reducing the energy wasted in idle mode, some low-power states have been used for improving the energy efficiency in transmit and receive modes. For example, on the Android G1, the transmit mode of the WNI is refined into two sub-states [30]. Each of these sub-states corresponds to a certain level of power consumption. Given a sub-state, the power consumption of the WNI is assumed to be constant. The WNI works in the sub-state with higher power

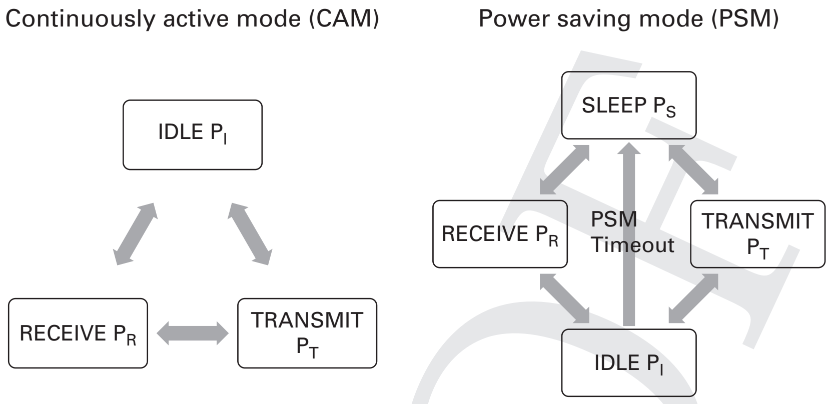


Figure 7.8 Power State machine of the WNI with the PSM Adaptive or the CAM enabled

consumption only if the packet rate is over a certain threshold. This is similar to the DVFS used in the microprocessor, which adapts the clock frequency to the processing workload [31].

Enhancements by 802.11n/ac

The more recent amendments to the 802.11 standard include 802.11n and 802.11ac, which introduce several new features and enhancements compared to their predecessors, 802.11a/b/g. The most important new physical layer enhancements include multiple input, multiple output (MIMO) support and channel bonding. In addition, two new power-saving mechanisms are included: spatial multiplexing power save (SMPS) and power save multi-poll (PSMP).

MIMO enables the use of multiple antennas, each with its own RF chain, simultaneously for data transmission/reception. MIMO can be used in two ways. One is to increase the spatial diversity by simultaneously transmitting redundant data streams encoded in a special way to increase the range and robustness of data transmission. The other way is to do spatial multiplexing by transmitting multiple separate spatial data streams simultaneously to increase the transmission rate. Channel bonding allows adjacent 20 MHz channels to be combined into wider single channels, thereby multiplying the bandwidth and transmission rate. The difference between the two amendments is that 802.11ac, for which the theoretical peak data rates are beyond a gigabit per second, can use wider channels (up to 160 MHz) and up to eight spatial streams for MIMO operations compared to a maximum of 40 MHz channels and four spatial streams in 802.11n.

In [32], the authors measured the energy consumption of 802.11n and discovered that only the number of active RF chains has a significant impact on the power draw. From an energy-consumption perspective, it seems mostly irrelevant whether the multiple RF chains are used for spatial diversity or multiplexing. According to the measurement results, the increase in power is not proportional to the number of RF chains that are active. They found that a two-antenna MIMO consumes roughly 50% more power than a single antenna configuration when transmitting, while a three-antenna configuration only adds another 5% to the power draw. This observation suggests that using more antennas, if available, is energy efficient but only if they are used to transmit/receive at a high rate. In other words, a single antenna configuration is the most energy efficient up to the maximum data rate that it can support. Beyond that data rate, multi-antenna configurations can potentially provide better energy efficiency. However, the configuration that achieves the highest throughput is not the most energy efficient in all cases. Therefore, finding the energy optimal MIMO configurations through rate adaptation, that is slowing down communication to save energy, has been recently under active research. Some example algorithms for energy-aware rate adaptation are proposed in [33, 34].

As for the other enhancements, using channel bonding has a negligible impact on power consumption according to their results, which means that it should be used whenever possible because it increases energy efficiency. SMPS and PSMP do not alter the basic behavior of the power-saving mechanism of 802.11. Both reduce the energy consumed during idle periods. SMPS reduces the power draw when the client is not receiving by switching off all but one RF chain. PSMP effectively makes it possible for the client to sleep for as much of the idle time as possible. Hence, both of these mechanisms impact the power levels in idle and sleep states.

7.3.2 Bluetooth

Bluetooth technology dates back to 1994 and it was created by Ericsson. Since then it has undergone several revisions and the latest major revision of the specification 4.0 was released in 2010 and its latest version is 4.1, released in December 2013.

Bluetooth uses the same frequency range for communication as Wi-Fi does. However, the two differ from each other in several important ways. First, Bluetooth is a device-to-device communication technology and it does not have an equivalent distinction of infrastructure and adhoc modes as Wi-Fi has. The channel access is also different: Bluetooth uses the adaptive frequency-hopping spread spectrum (AFH) as opposed to Wi-Fi's random access (CSMA/CA). In AFH, the device continuously hops from one frequency to another 1600 times per second according to an agreed sequence. AFH is robust against interference from other Bluetooth devices because of different hop sequences. It is also robust against interference from other technologies using the same spectrum through adaptation of the hopping sequence: when interference is detected, occupied frequencies, such as those overlapping with busy Wi-Fi channels, are blacklisted and omitted from the hop sequence.

Bluetooth communication can be divided into activities done in non-connected and connected states. Behavior in the non-connected state includes discovering other Bluetooth-capable devices and establishing a connection with them, while the connected state includes the actual data transfer. We next discuss both in turn.

Device discovery

Discovering other devices using Bluetooth usually consumes a significant amount of energy. A device uses a process called inquiry to discover other devices. During this process, the inquiring device sends an inquiry message at every time slot in two different frequencies and spends the next time slot waiting for responses on the same frequencies. The frequencies used follow pseudo-randomly selected trains of sequences. A discoverable device hops according to a pre-specified hopping sequence, listens for incoming inquiry messages, and responds upon receiving one. Therefore, because of the frequency-hopping channel access, the discovery phase in Bluetooth requires randomly polling for frequencies to find the other device. As a consequence, the discovery takes usually much longer, up to 10 seconds, compared to Wi-Fi using active scanning, for instance, which takes typically less than a second. Inquiry is followed by paging to establish a connection between the devices.

The energy consumption of a smartphone during the discovery phase is relatively large, in large part due to the long duration. The energy consumption also depends on whether the smartphone is scanning for other devices or just being discoverable, meaning that it listens for incoming inquiry messages. The power draw is typically higher when inquiring compared to being discoverable but there is significant variation between different phones as demonstrated by the measurement results presented in [35].

Because of the proportionally large energy demand of Bluetooth discovery, the energy efficiency of the discovery mechanism has been studied so it can be optimized. Bluetooth also allows the discoverable device to only selectively listen for incoming inquiry messages. Two parameters, the scan window and scan interval, control how often and for how long the device listens, which also directly influence how much energy is consumed. In opportunistic discovery scenarios, where both devices can be either inquiring or act as discoverable device, the time spent in each phase also has an important role. Indeed, if two devices never happen to take the opposite roles at the same time, they will never discover each other. It has been shown that tuning these parameters has a significant impact on the overall energy consumption [36]. Dynamically tuning the parameters so that more "aggressive" discovery is performed when new devices could be expected to be in range can be very beneficial in opportunistic discovery scenarios.

Data transmission

Once connected, the two devices hop along the same sequence of frequencies and exchange messages. The power consumption during Bluetooth data transmission is relatively constant regardless of the data rate. Compared to Wi-Fi, the power consumption of Bluetooth is lower, but the throughput achievable is also lower than that of Wi-Fi. Therefore, in most cases Wi-Fi yields a better energy utility, that is, more bits transferred per joule spent. This result is true for bulk data transfers, whereas low bit rate transfers, such as audio streaming, would be more energy efficient to receive or transmit using Bluetooth [35].

Bluetooth low energy

Bluetooth low energy (BLE) was introduced into the standard in version 4.0. It makes low rate and very low-power communications possible using Bluetooth. However, it is not compatible with classic Bluetooth in the sense that a pure classic device cannot communicate with a BLE device. For this reason, dual-mode devices have been introduced, of which a typical example is a smartphone. It can act as both a classic and a BLE device. The original idea of BLE was to be able to support communications for coin cell battery operated devices with up to two years of battery life. Needless to say, the energy consumption must be very low for such scenarios.

BLE communication takes place between a slave and a master device. Compared to classic Bluetooth, the device discovery has been simplified so that there are three dedicated advertisement channels on which a slave device periodically sends advertisement messages. The periodicity is determined by an advertising interval. The master device must listen (or scan) on one or several of those channels for incoming advertisements. Upon receiving one, the master can initiate a connection with the slave.

BLE makes heavy use of duty cycling. All the data transfer in BLE happens through so-called connection events where both the slave and master wake up in synchrony to exchange messages. The two devices can sleep for the remaining time. The frequency of such events is determined by the connection interval parameter.

The way BLE achieves low energy consumption is a combination of two key features: long duty cycles and ultra low power consumption in sleep mode. The energy consumption during the discovery phase is heavily asymmetric. The slave only needs to periodically send an advertisement message and listen for a short while for a reply, whereas the master needs to scan the channels for incoming messages and may need to spend a considerable amount of energy. The energy consumption of the discovery phase can be influenced through setting the corresponding parameters [37].

To illustrate the energy consumption once connected, Figure 7.9 plots the energy utility as a function of packet size. The plot was generated using a power model generated from measurements with a BLE keyfob [38]. There is a certain amount of overhead energy spent for each connection event, so when the size grows the energy utility

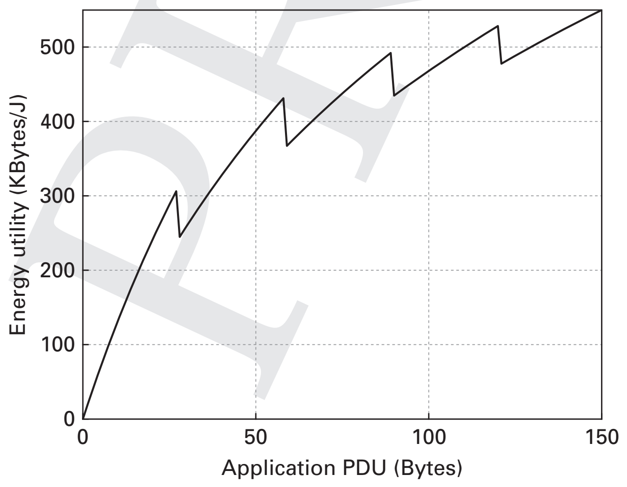


Figure 7.9 BLE energy utility grows with the packet size

increases as well. The curve has a sawtooth shape because the payload size is limited to a maximum of 27 Bytes at the link layer, which corresponds to 23 Bytes of application data when taking protocol headers into account. Therefore, the energy utility steps down each time the 23 Bytes boundary is crossed and an extra packet has to be included to convey the entire application PDU (Protocol Data Unit). To put these numbers into perspective, a classic Bluetooth bulk transfer achieves an energy utility of roughly 15–30 kB/J using RFCOMM and up to 200 kB/J with audio streaming [35], which are rather far from the energy utility of BLE. As for Wi-Fi, the results presented in [39] demonstrate that Wi-Fi's energy utility is highly linear. In the low bit rate range, it is clearly less energy efficient than BLE but, in contrast, the peak rates deliver very high energy utility, up to 2.5 MB/J.

Example power measurements

To get a better idea about how the energy consumption of using classic Bluetooth compares to the low-energy version with a smartphone, we show some measurement results from using a modern Android smartphone. The OS had been upgraded to the Android 4.3 to get BLE support. The phone, similar to all other BLE-capable phones, runs a dual-mode Bluetooth chip where the same radio is used for both classic and low-energy communications. We wrote our own very simple application that sequentially uses both kinds of Bluetooth and recorded a six-minute long power trace when using that application.

Figure 7.10 shows the average power consumption in different modes which we extracted from the power trace. The idle power when running the application and having the Bluetooth interface enabled was subtracted from the results. In fact, whether Bluetooth was enabled or not seemed to have no impact on the device power draw.

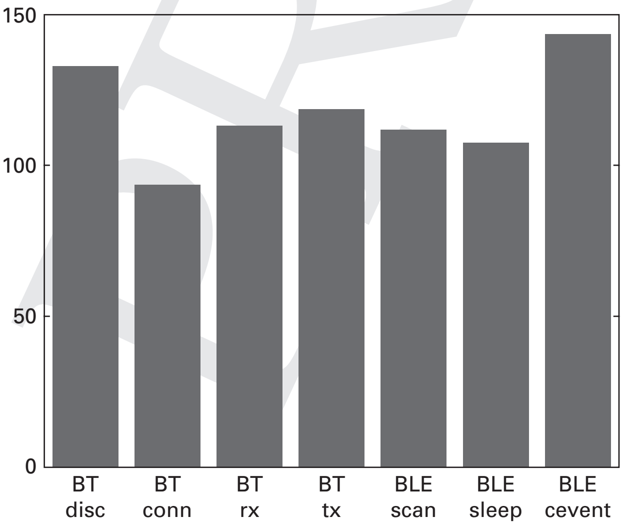


Figure 7.10 Average power of different Bluetooth power modes

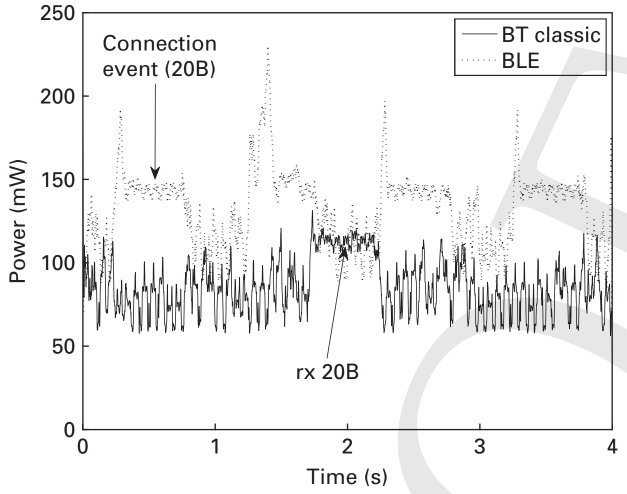


Figure 7.11 Power trace of classic Bluetooth receiving 20 Bytes and BLE receiving 20 Bytes in periodic connection events

Classic Bluetooth's discovery seemed to be slightly more power hungry than the simpler BLE discovery (BT disc vs. BLE scan), as expected. Interestingly, there was no huge difference between the power consumption of the two types of Bluetooth while being connected and idle (BT conn vs. BLE sleep). So, it seems that when using BLE, the Bluetooth radio circuitry is hardly switched into a sleep mode in between the connection events. Surprisingly, BLE connection events draw the highest amount of power (BLE cevent).

To illustrate the difference in energy consumption when receiving 20 Bytes of data over classic Bluetooth and over BLE, we plotted power traces corresponding to the two cases in Figure 7.11. With the used device, it seems more energy efficient to receive even small amounts of data using the classic Bluetooth instead of BLE. These measurements suggest that the current smartphones have not (yet) been optimized for BLE power efficiency. Hence, the low-energy part of BLE mainly applies today to the battery-operated peripherals and small gadgets that connect to the smartphone using BLE.

7.3.3 Cellular networks

HSPA (3G) and LTE (4G) are currently the two dominant types of cellular network that are used for data communication. We next describe the energy-consumption characteristics of each of them in turn.

HSPA (3G)

In the HSPA cellular network, the use of radio resources and power consumption of a mobile phone is controlled by the radio resource control (RRC) protocol. The most relevant part of the RRC from the energy-consumption perspective are the four states out of which three correspond to specific transport channels:

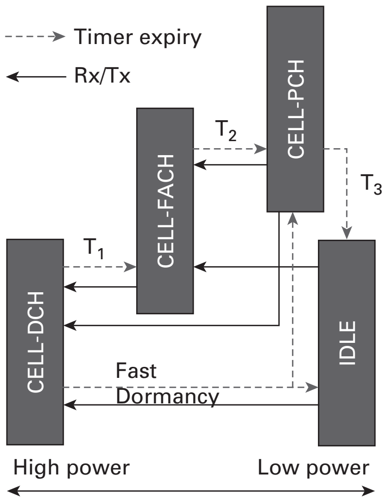


Figure 7.12 3G state transitions and power consumption

CELL_DCH is used for data transmission involving relatively large traffic volumes, such as bulk transfer in streaming applications. It provides the best network throughput, but it also creates the highest power consumption. Therefore, RRC protocol switches the state from CELL_DCH to CELL_FACH for sending/receiving a small volume of traffic with lower power consumption, and to the CELL_PCH state to get the lowest power consumption when there is no active data transmission on the mobile device [40]. The fourth state is idle in which the RRC connection does not exist. The power draw in the idle state is almost the same as in the CELL_PCH state.

Figure 7.12 illustrates the RRC state machine and the inactivity timers that control the transitions among these states. After no data has been received or transmitted for a timer-specified duration, a state transition occurs. The timer values are typically around several seconds and are controlled by the network operator. Some operators may not enable CELL_PCH in their network and in that case the RRC connection is terminated upon the expiry of T2. The figure also shows that operating in different states draws different amounts of power. The figure also depicts a mechanisms called fast dormancy which we discuss in Section 7.3.5.

LTE (4G)

LTE (Long Term Evolution) is the latest generation of cellular network technologies. LTE behavior with respect to energy consumption can also be described in terms of states. However, LTE's state diagram only contains two states: RRC_IDLE and RRC_CONNECTED [41]. Similarly to 3G, there is an inactivity timer with a typical value of 10 s associated with the transition from the connected to the idle state. The power draw in the RRC_CONNECTED state is much higher than in the RRC_IDLE state.

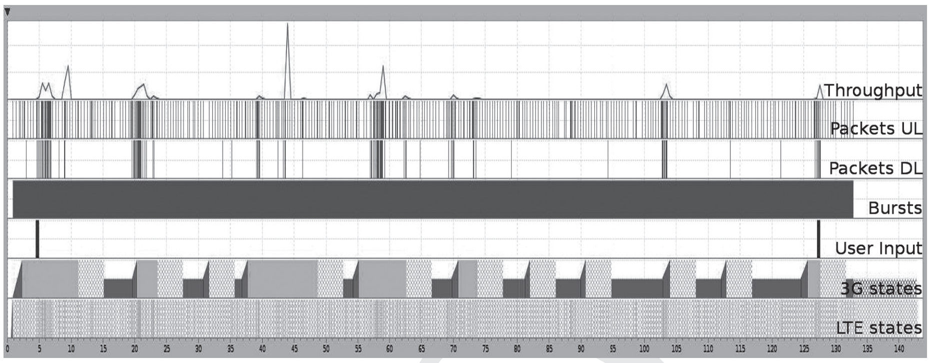


Figure 7.13 Power states of AT&T 3G and LTE when playing Angry Birds on a Samsung Galaxy S2

Monitoring the RRC state

The open-source Application Resource Optimizer (ARO) from AT&T Labs[3] can be used to obtain packet-level smartphone network traces of both Android and iOS phones. We can apply HSPA or LTE profiles of network providers on these traces. ARO estimates the RRC state changes from the traces, which requires a certain number of network-specific parameters, in particular the inactivity timers controlling RRC state demotions, to be known. These parameters can be inferred through a separate profiling process[4] or they can be known a priori.

Figure 7.13 shows a trace of the popular Angry Birds game. The trace is 143 seconds long and contains starting the application, two minutes of playing Angry Birds, and closing it. At the top of the figure, Throughput shows the change in bandwidth requirements across the trace. Packets UL and DL show the density of uploaded and downloaded packets. Bursts shows the placement of data traffic bursts (only one in this figure). User input shows when the user pressed hardware buttons on the device, such as the home and back buttons.

The two rows at the bottom are the 3G and the LTE power states when applying AT&T's network parameter settings with the trace. Focusing on the 3G as an example, the ramp-ups colored with the darkest shade of gray indicate state promotions to a higher power mode, such as from idle to DCH right at the beginning of the trace. The height of the bar indicates the power draw of the state. CELL_FACH, the second lowest power state, is colored with the same dark gray. The next darkest shade of gray indicates active data transmission in the CELL_DCH state. The lightest shade of gray designates time spent in the CELL_DCH state without active data transfer waiting for the inactivity timer to expire, resulting in so-called tail energy, which we discuss more in the next section. While the diagram does not give power values directly, it shows the behavior of the two communication technologies when facing small but continuous traffic.

[3] `https://github.com/attdevsupport/ARO` accessed January 6, 2014.

[4] Detailed description of how the network- specific parameters can be inferred is presented in [42].

Another solution called RILAnalyzer is an on-device tool to monitor the RRC states of a 3G modem [43]. It works on rooted Android phones that have the Intel/Infineon XGold chipset. The tool queries the 3G state information using the radio interface layer (RIL) through which the modem exposes such information to the OS. In addition, the tool collects IP packets through userspace logger leveraging iptables. The fact that RIL-Analyzer directly queries the RRC state is the main difference to ARO which estimates the state and requires the a priori known network parameters.

Tools such as the ones presented above are very useful for application developers who want an easy way to get at least a rough understanding of the energy efficiency of their application from the wireless-communication perspective and, consequently, to optimize it.

7.3.4 Tail energy

All the WNIs have one specific characteristic related to energy consumption in common. As we have just learned, all the different WNIs have mechanisms to ensure that the radio is not fully powered on all the time. Furthermore, the transitions from active to more passive modes, where the radio sleeps at least part of the time, are based on timers. The energy that is consumed because of powered on radio for the inactivity timer specified duration is often called *tail energy* [44]. It is present in all WNIs, but the amount of it may differ substantially.

The timer values are wildly different with the different access technologies. Typical values for an HSPA and LTE networks are up to ten seconds or more, whereas Wi-Fi puts the radio to sleep after only one to two hundred milliseconds. The timer values are not specified in the standards and in cellular networks, they are set and controlled by the network operators. Consequently, the amount of tail energy is vastly different depending on the technology and the specific operator-dependent configuration. Figure 7.14 illustrates how tail energy is consumed with 3G. Timers T1 and T2 control the transitions from CELL_DCH to CELL_FACH and from CELL-FACH further to CELL_PCH or idle, respectively. P1 and P2 are the power drawn in the corresponding states.

Why do such timers exist in the first place if they lead to excess energy consumption? In a sense, the timers and the tail energy caused by them are the price to pay for more efficient radio access network resource consumption and shorter application-level latency. The reason is that switching modes between sleep and active power modes takes a non-negligible amount of time because of hardware limitations. Furthermore, scheduling the shared resources of a radio access network efficiently becomes a difficult task if the duration of resource use can be arbitrarily short. This is true especially with applications that communicate in a sporadic manner, that is transmitting or receiving packets every now and then without a regular and predictable structure in traffic patterns. Note that the issue is not as prominent with Wi-Fi because it has no central scheduler but relies instead on random medium access.

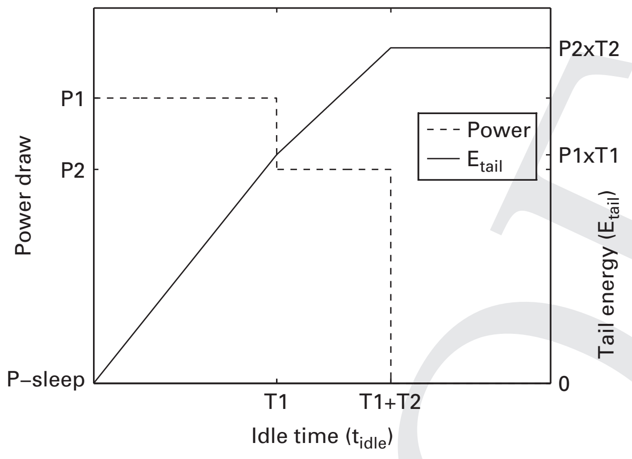


Figure 7.14 Tail energy with 3G having two timers: P1 and P2, corresponding to the power draw in the CELL_DCH and CELL_FACH states, respectively

7.3.5 Mechanisms to reduce tail energy

The impact of tail energy on the overall energy consumption of data communication using 3G or LTE is so dominant that special mechanisms to mitigate it have been devised for the cellular access networks. Table 7.2 summarizes the characteristics of tail energy when using different wireless access networks with a typical modern smartphone and the mechanisms to mitigate it. The delta values for the power draw in different states is simply the power in the idle state after the tail energy has been subtracted from the power in that state, for example, $\Delta P_{DCH} = P_{DCH} - P_{PCH}$.

Fast dormancy in 3G

Fast dormancy (FD) is a technique that allows the 3G WNI to switch directly from CELL_DCH to CELL_PCH or to the idle state. There are two types of FD. Initially, some phones started to support a non-standard mechanism that relies on misusing a specific signaling message: the phone transmits a signaling connection release indication message, normally used to communicate certain error conditions by the phone, to the network which causes a tear down of the PS signaling connection. The WNI may end up either in the CELL_PCH or the IDLE state.

The good news is that power draw is immediately lower. The bad news is that when there is a new communication the phone must re-establish the signaling connection, which introduces additional delay and more signaling traffic into the network than if the WNI had transitioned into the CELL_PCH state without releasing the signaling connection. Especially due to the signaling storms experienced by operators [45], 3GPP Release 8 introduced a new mechanism. This standardized mechanism is a network-controlled FD where the phone indicates to the network that the data session is finished for now and requests the network to transition the device into an appropriate state. The network decides whether the request is granted or not and to which state the device is

Table 7.2. Tail energy and built-in mitigation mechanisms with different wireless access networks, with typical power and timer values; actual values depend on the hardware and cellular network configuration

WNI	Tail timers	Power values	Mitigation mechanisms
Wi-Fi (with PSM)	$T_{PSM} = 0.2$ s	$\Delta P_{idle} = 0.78$ W	Unnecessary
HSPA (3G)	$T_1 = 8$ s	$\Delta P_{DCH} = 0.78$ W	Fast dormancy (FD),
	$T_2 = 3$ s	$\Delta P_{FACH} = 0.59$ W	continuous packet connectivity (CPC)
LTE	$T_{idle} = 10$ s	$\Delta P_{connected} = 1.3$ W	Connected mode DRX/DTS

transitioned. For example, very frequent transitions to CELL_PCH may not be allowed, again due to increased signaling traffic. The problem with the signaling traffic when using legacy FD (referred to as autonomous signaling connection release) and analysis of the effectiveness of the standardized FD to mitigate the problem are explained in detail in [46].

Discontinuous reception and transmission in active states

Discontinuous reception and transmission used specifically in connected states is another set of mechanisms based on duty cycling. It is employed in both LTE and HSPA networks. The idea is that when the device has no data to transmit or receive, it sleeps most of the time and only periodically wakes up to receive control sub-frames indicating whether new incoming packets are buffered at the base station. So, the mechanism is very similar in nature to the PSM of Wi-Fi. Note that in non-active states, such as CELL_PCH or RRC_IDLE, discontinuous reception is automatically applied, which is why these states have naturally low power consumption. The idea is simply that the same mechanism is applied in the active states CELL_DCH and RRC_CONNECTED even during active data transmission.

In LTE networks, these mechanisms are dubbed cDRX/cDTX where the "c" stands for connected mode. The state diagram of LTE with cDRX/cDTX enabled is shown in Figure 7.15. As show in the figure, the activation of discontinuous reception is also controlled with a timer (we refer to it as T_{DRX}). The WNI goes through different DRX cycles short and long until the inactivity timer expires and the interface goes to the RRC_IDLE state. Figure 7.16 illustrates the way that the different cycles work in cDRX: after the T_{DRX} of no packets received, the DRX short cycle is activated during which the radio is awakened periodically for T_{on} amount of time to receive control frames from the base station. After a predefined number of short cycles, the DRX long cycle is activated, which simply means that the radio wakes up less frequently. This goes on until the LTE inactivity timer expires.

Continuous packet connectivity (CPC) is an equivalent solution for HSPA data communication (3G) [47]. It the activates DTX/DRX mechanisms in the CELL_DCH state. It was introduced in the 3GPP Release 7.

The energy savings achieved through discontinuous reception are highly dependent on the configuration. The shorter the DRX inactivity and the on-duration timers, and the

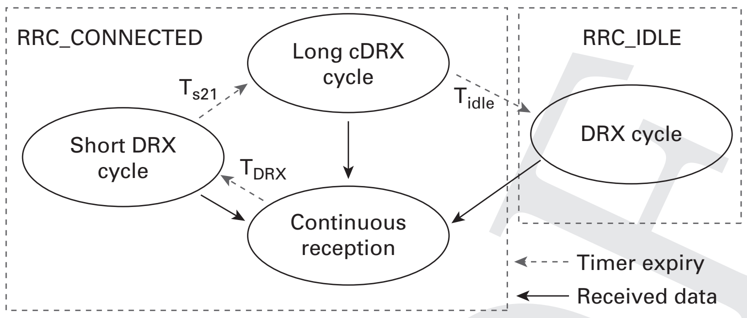
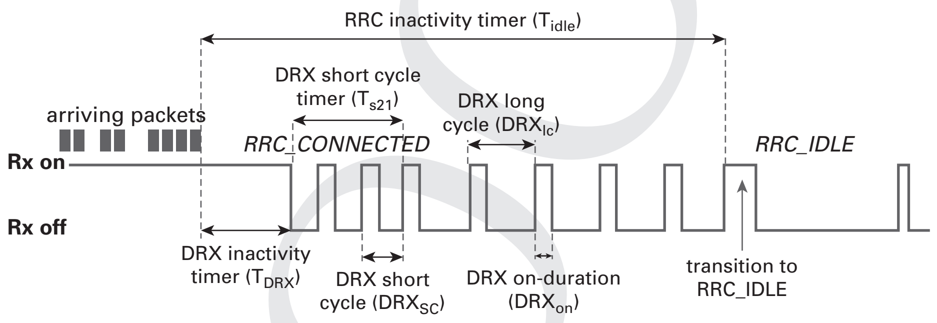


Figure 7.15 LTE state diagram with cDRX/cDTX



RRC inactivity timer (T_{idle})
DRX short cycle timer (T_{s21})
DRX long cycle (DRX_{lc})
arriving packets
RRC_CONNECTED
RRC_IDLE
Rx on
Rx off
DRX inactivity timer (T_{DRX})
DRX short cycle (DRX_{SC})
DRX on-duration (DRX_{on})
transition to RRC_IDLE

Figure 7.16 LTE state diagram with cDRX/cDTX

longer the DRX cycles, the less energy is consumed. However, such "aggressive" timer configuration, that is long DRX cycles and short timers, may increase the delay and reduce the responsiveness of some applications. Consider, for example, web browsing where requests and page loads occur every now and then and the user reads the rendered page in between. With such applications the user experience may worsen due to increased perceived latency in page loading caused by an aggressive timer configuration.

Smartphones also currently put some constraints on the range of effective DRX cycle configurations. It turns out that because of hardware and/or modem software limitations, the energy savings, compared to the case of not having DRX enabled at all, decrease substantially with the current chipsets when reducing the DRX cycle length. Figure 7.17 shows some example measurement results with three modern smartphones from different manufacturers. In these measurements, the on-duration was set to 10 ms. Therefore, if we assume negligible power draw when the radio is in sleep mode, a theoretically optimal DRX implementation would have kept the radio on for 10 ms and allowed it to be off for 70 ms when using an 80 ms DRX cycle, for instance. That calculation yields 87.5% energy savings compared to the case when DRX is disabled. In contrast, we observe roughly 20–40% energy savings, which means that there is still some room for optimization even if there will always be some overhead associated with the switching of the radio. These limitations are crucially important for the energy efficiency of

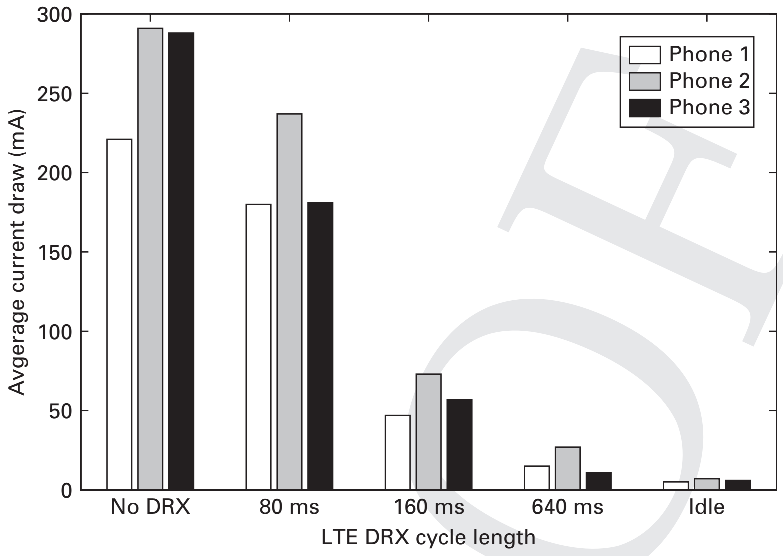


Figure 7.17 The energy efficiency of DRX reduces with current smartphones when the DRX cycle gets shorter

Voice over LTE (VoLTE)[48]. If DRX is used in conjunction with VoLTE, a DRX cycle of only 20 or 40 milliseconds must be applicable.

7.4 Sensors

In recent years, mobile phones have undergone an evolution from simple communication devices into effective multi-sensor platforms, starting a new era of context-aware applications and services. As the increase of new sensing and computational capabilities emerge at an increasing pace, energy-efficient sensor management has become a central concern for mobile sensing. We next look at the different kinds of sensor embedded in typical smartphones and their energy-consumption characteristics.

7.4.1 Types of sensors in smartphones

There can be tens of sensors integrated into modern smartphones and more are being packed into each new generation of phones. Consequently, for clarity and ease of referring we group the sensors into four categories based on the type of context the sensors are used for:

- Motion sensors describe the orientation and movement of the phone and include accelerometers, gyroscopes, and magnetometers. Additionally, many phones include virtual sensors, such as a rotation sensor, a linear acceleration sensor, a gravity sensor, and a significant motion sensor, which capture specific aspects of one or more of the underlying hardware sensors.

- Wireless sensors send and receive information from external sources and include sensors such as GSM, GPS, Wi-Fi, Bluetooth, NFC (Near Field Communication), and infrared.
- Environmental sensors sense various information from the phone's immediate surroundings and include sensors such as a microphone, camera(s), and light, proximity, pressure, humidity, and temperature sensors.
- Internal sensors track the state of the phone's main functions, such as the battery level, voltage and temperature, the current state of the phone's screen, and the current state of the phone calls.

The above categorization is based on encompassing sensors used for sensing similar context. Motions sensors are a natural choice for any activity recognition tasks and can provide an easy, low-energy solution for detecting mobile and stationary periods. The primary uses wireless sensors can be divided into location sensing and sensing for data communication, but they have also found use in other domains, such as detecting crowding in public spaces and estimating the cognitive state of the user. From the wireless sensors, GPS should be distinguished as a special case for energy-efficient sensing, as it is both one of the most employed sensors, as well as one of the most energy consuming ones. Environmental sensors, in addition to the multiple uses of camera and microphone, provide information about the user's immediate surroundings, which can be fused with motion sensors for more accurate activity recognition, or provide hints for positioning from location change. The main use of internal sensors is to filter and adjust the use of other sensors, and to ensure that the phone's hardware is not overly strained. Additionally, motion sensors can greatly benefit from information when the phone movement is caused by user interaction, for example because of an ongoing phone call rather than the user's physical movement.

7.4.2 Characterizing the energy consumption of sensors

When designing a sensor-sampling strategy, it is necessary for the developer to have a clear understanding about the sensor's behavior and what are the key factors for its energy consumption. For this purpose, it is useful to have at least a coarse-grain characterization of the energy consumption, typically in the form of a very simple deterministic power model (we discuss the different types of power model in Chapter 9).

A sensor power model describes the different stages and the associated energy consumption of a given sensor. For example, Table 7.3 lists measured power and energy values for a particular smartphone. Obviously, the absolute values shown in the table do not hold for all smartphones. The phone hardware and OS may have a significant impact on the energy-consumption characteristics. Each sensor typically consumes some energy for triggering the sensor on and off, and draws power while sensing. Also, keeping the sensor powered on but not actively sensing, typically draws some power, that is idle power.

The granularity of the models can be extended to include different stages of the sampling phase. For instance, the sampling phase of a GPS sensor can be divided into

Table 7.3. Sample energy and power measures for sensors of the Samsung Galaxy S2

Sensor	Switch ON	Switch OFF	Sampling	Idle	Pre-sampling
Accelerometer	–	–	21 mW	–	–
Gravity	–	–	25 mW	–	–
L.Acceleration	–	–	25 mW	–	–
Magnetometer	–	–	48 mW	20 mW	–
Orientation	–	–	49 mW	20 mW	–
Rotation	–	–	50 mW	21 mW	–
Gyroscope	–	–	130 mW	22 mW	44 mJ
Microphone	123 mJ	36 mJ	101 mW	–	–
GPS	77 mJ	–	176 mW	–	198 mW

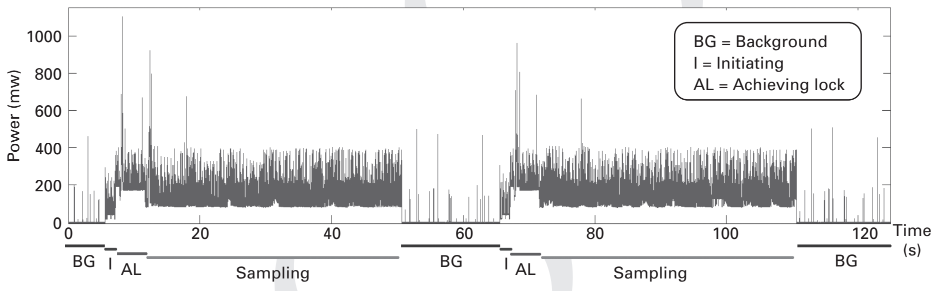


Figure 7.18 Energy measurements from GPS sensor

achieving a GPS lock and tracking position, as illustrated in Figure 7.18. More precise models can be created by adding further information to the models, for example the effect of different sampling rates on the energy consumption, the standard deviation of the energy consumption, and the duration of switching the sensor on/off.

7.4.3 Multiple processors for continuous sensing

Continuous sensing involves the constant monitoring of onboard sensors, such as an accelerometer, microphone, or camera. Thus continuous sensing burdens the processor and uses a lot of energy. Current and forthcoming smartphones support the offloading of sensing tasks to auxiliary processors and low-performance cores to save energy. For example, the Android OS supports batching sensing operations, and as an example we can consider the Samsung Galaxy S4 smartphone that has a hardware chip for aggregating and optimizing sensor data gathering and processing. Figure 7.19 illustrates a sensor hub co-processor that performs always-on monitoring of sensors, allowing the host CPU to sleep. Sensor hubs are an effective way to offload sensing data polling and processing from the main application cores to a more dedicated hardware processor [49].

Most sensing applications consist of a sequence of stages that process sensor data. The typical stages include the following [50]:

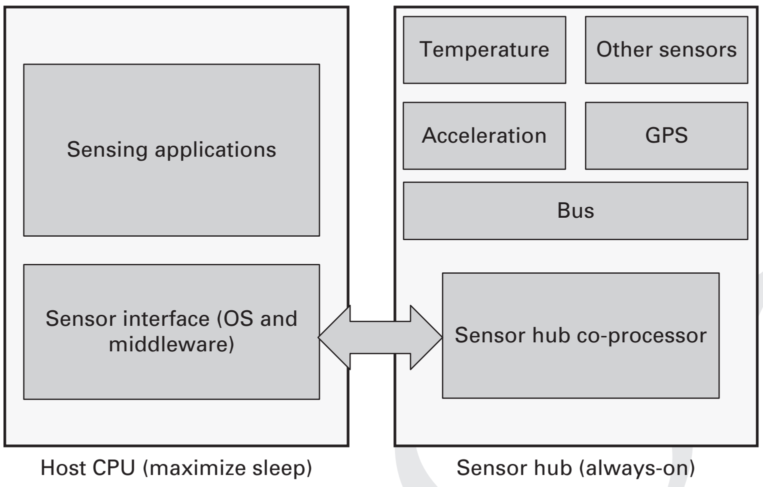


Figure 7.19 Overview of a sensor hub

- Sampling and buffering, in which the sensors are sampled and the data is placed into a buffer.
- Filtering, in which the interesting parts of the data are identified and selected for further processing.
- Feature extraction, in which features are extracted from the data to perform classification.
- Classification, in which the data is classified based on the extracted features by using machine learning or probabilistic methods.
- Post-processing, in which the applications react to the sensing result.

Assuming that an application consists of *N* stages and that we have more than one processor for running the stages, the central question is how to place the stages across the processors. The placement decision needs to take into account the energy characteristics of the processors as well as the processor wakeup and stage/task scheduling cost [50].

Given that we have a high-performance core and a lower performance core for sensing tasks, it is clear that computationally heavy operations should be run on the high-performance core. The low-performance core, on the other hand, would be suitable for reading sensors and then handing over the data to the high-performance core for intensive processing, such as speech recognition.

Assuming that the low-cost processor is the most suitable for a specific computation stage, *i*, we have the following bound for the slow-down of the stage [50]:

$$s_i < \frac{P^M_{active} - P^M_{sleep}}{P^L_{active}} + \frac{E^{trans}/P^L_{active}}{T^M_i}, \tag{7.9}$$

where P^M_{active} and P^M_{sleep} are the power consumptions of the high-performance main core when in active and sleep states, respectively. In a similar fashion, P^L_{active} is the power consumption of the low-performance core, E^{trans} is the transition cost for stage placement on the main core, and T^M_i is the execution time of the stage on the main core.

The slow-down factor is dependent on the hardware and on the computation. For example, the memory size of the processor, data parallel instruction sets, floating point units, bus speed, DMA availability, cache size, processor frequency and frequency scaling, and auxiliary components such as DSP instructions affect the outcome. Experimental results indicate that the transition cost for the main core is several orders of magnitude higher than for the low-performance core. Moreover, the sleep state cost of the main core is significant and comparable to the low-performance core in the active state. Most energy benefits are achieved when placing simple and frequently used sampling and buffering tasks to the low-performance core [50].

In addition to stage scheduling across heterogeneous processors, programming abstractions are also needed to support application development using multiple processors. The Reflex platform aims to make it easier to develop for multi-processor mobile platforms [51]. This approach abstracts low-level development issues by using a shared distributed memory abstraction.

7.5 Camera

Taking images and capturing videos with smartphones and sharing those with others is increasingly popular. When shooting a video, the smartphone uses the display, the camera internal hardware such as the image sensor, the CPU for encoding the video, and it also files IO operations for temporary storage of the encoded content. The total power consumption can be very significant. In this section, we explain how much power and by which hardware component the power is drawn while using a smartphone camera.

7.5.1 Image sensor

The heart of a modern smartphone's camera is usually a CMOS (complementary metal-oxide semiconductor) image sensor. It is cheaper to manufacture than a CCD (charged-coupled device) image sensor which is the other often used sensor type. The sensor is typically integrated into a mobile device, such as a smartphone, so that it is directly connected to the application processor through MIPI interfaces.[5] Applications use the image sensor through the API exposed by the OS. The API allows the camera operations to be controlled and usually also provides options to configure some of its properties, such as those related to the image quality.

When powered on, a CMOS image sensor mostly alternates between active and idle states and sometimes it can go to a standby mode. In the active state, the pixels are read

[5] MIPI (mobile industry processor interface) Alliance defines interfaces for mobile devices in order to have a common set of them used by the mobile device industry.

out from a pixel array, which is a component that consists of a set of photodetectors and transistors, whereas in the idle state the sensor is on but the pixel array is not being read. The sensor operates according to a clock. The clock speed determines the highest rate at which pixels can be read and processed by the sensor, typically one pixel per clock period. Hence, the amount of time spent in active vs. idle modes depends on the frame rate, image resolution, and clock speed.

The work presented in [52] investigated the energy consumption of a CMOS image sensor. It turns out that the image sensors are typically quite far from being energy proportional with respect to image and video quality. Specifically, they consume much more energy per frame with small frame rates than with high frame rates, and similarly, more energy per pixel with low resolutions than with high resolutions. This phenomenon stems mainly from the fact that most of the image sensors draw only a little less power in the idle state compared to the active state, and that the standby mode is not actively used.

The energy proportionality can be improved substantially by using two straightforward mechanisms: clock scaling and the standby mode. As is usually the case with such circuits, the clock speed directly affects the power drawn by the image sensor in both active and idle states in a linear manner: the lower the clock speed, the smaller the power. However, the clock speed also determines the highest resolution and frame rate combination that the sensor can support. Therefore, by scaling the clock speed according to the configured parameters, the energy consumption could be made more proportional to these two parameters. In addition, using the standby mode to reduce the time spent in the idle state can potentially reduce the energy consumption further. Of course, these techniques reduce the energy consumption only when high-quality images and videos are not required.

7.5.2 Dissecting the power drawn by smartphone cameras

To understand how much energy is consumed while recording a video in total and individually by different operations involved, we performed measurements with three different modern smartphones from different manufacturers running different operating systems. We measured different cases to be able to decompose the power drawn by the smartphone into sub-tasks. We specifically quantified the power drawn by the idle phone operated in airplane mode, by the display, by having the camera switched on in focus mode, and finally by recording the video which we call the shoot mode. We show the power consumption break down in Figure 7.20 and we next discuss each part in turn.

Display

Of the tested phones, phone 1 uses in-plane switching liquid crystal display (IPS LCD) and phones 2 and 3 use active-matrix organic light-emitting diode (AMOLED). To evaluate the display power consumption, we set the display to show only white or only black color on the entire screen using a medium brightness level (the brightness level can affect the power draw by up to half a watt). During the measurement there were no background applications running and the device was in airplane mode. We chose white

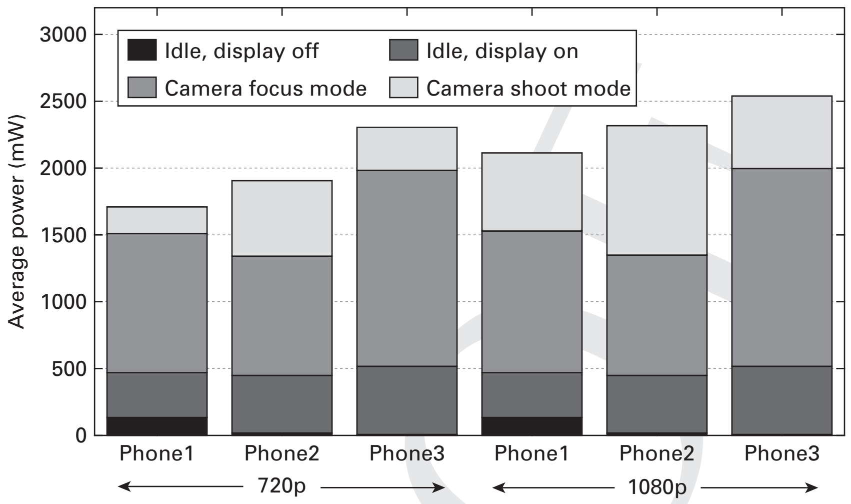


Figure 7.20 Smartphone power consumption during video recording

and black due to the usual dynamics in the power consumption by the display panels for color reproduction; black drawing minimum power and white drawing maximum power. The difference between the two was roughly 250 mW. We then computed their average and show that in Figure 7.20. With these phones, OLED seems to draw about 300 mW more power than IPS LCD to display the same color.

Focus mode

Switching the camera on but not yet recording video causes the camera to focus on the object being shot and display it on the screen. The camera internal hardware is effectively being used in addition to the application that controls camera operations. Using the knowledge we have about the average power consumed by the display, we calculated the added power by this mode of operation and stacked it on top of the display power in the figure. We note that the overall increase in power draw is very substantial, even exceeding one watt. In addition, there are notable differences between the phones but resolution does not yet play a role in this phase, which is at least partially explained by the lack of energy proportionality of the image sensor discussed earlier in this section.

Shoot mode

When recording video, the power drawn by the processing and storing of the video content to the file system is added. Again we measured this part in the same way by subtracting the previous phase (focus mode) power from the total power in shoot mode and stacking the result on top of the two earlier components in the figure. Again, we observe that there are quite large differences between the phones: the extra power drawn by recording the video is relatively small for phone 1 but clearly more significant for phone 2 compared to just the focus mode power. In this last phase, the video resolution also starts to play a role. The increase in power overall is not tremendous and it differs between the phones. For phone 3, which already draws pretty high power in lower resolution, the increase is only about 200 mW, whereas for phone 1 the increase is almost 500 mW.

Taking into account the battery capacities of the tested smartphones, we can compute the operating time. Using a fully charged battery, the operating time varied between two and three hours if continuously shooting video with the phones, which means that the energy consumption rate is very high but the battery capacity may just be sufficient to record an entire live event, such as a concert.

7.6 Summary

In this chapter, we examined the components and features of the smartphone including the CPU and SoC, communications, display, sensor subsystems, and mobile operating systems. The power management and optimization is implemented throughout these components. The OS examined in Chapter 8 has the crucial role of managing the device-level power parameters. We discussed the key components and our observations included the following:

- The modern smartphone hardware consists of multiple physical CPU cores and auxiliary processors. The OS scheduler is responsible for distributing the load across the processors. The system logic controls the CPU speed and furthermore enters/exits C-states at the perceived need. These are the cause of frequent modifications of parameters such as clock speed, number of CPU cores used, speed of the memory bus, and voltages in the wiring system of the chips (CPU and memory). Changes in all these parameters could be avoided if the CPU speed scaling is prevented. This also directly decreases the probability of changes in memory buses (voltage and clock) and in other components of the core. The scheduling system also contributes to the time core components spend in low-power modes.
- Communication activities, such as cellular wireless, Wi-Fi, and Bluetooth consume power when in receiving, transmitting, or scanning states. Thus, it is better to try to use a mode (e.g. the so-called airplane mode) which decreases bias, when not measuring just cell radio power. Also Wi-Fi or Bluetooth can be separately disabled/enabled if need be.
- A notably power-hungry device is the screen and its backlight, both of which can be turned on or off. In certain technologies, the color usage contributes to changing current usage causing bias. Thus, the safe bet is to turn the screen totally off if possible.
- The camera is another component that exhibits a very significant power draw. However, optimization possibilities fortunately exist, as demonstrated by recent research results.

As we have seen, the measurement of current use is a critical feature of the power economy of a mobile device. There exist standard principles and methods to do this. The following principles are important when measuring current use:

- The current should be measured as average instantaneous current and we should use the nominal voltage. Normally this requires using a standardized power source and also special software tools meant for gauging batteries.
- No external charger should be used and no USB connection to the host should be active, has this would cause external current usage and give lower measurement values for the battery.
- An effort should be made to keep the mobile system outside the target component (the component whose power consumption is to be measured) running in a stable state, drawing constant only constant power from the battery. Thence we try to avoid the measurement inaccuracy caused by the possible changing states of the device surrounding the target component.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[2] *ARM Infocenter Website*, Nov. 2013. [Online] Available at: http://infocenter.arm.com

[3] D. Lewis, *Fundamentals of Embedded Software with the ARM Cortex-M3*. Pearson Education, 2012. [Online]. Available: http://books.google.fi/books?id=iaIvAAAAQBAJ

[4] A. Edelsten, "TEGRA: Attacking Mobile Entertainment with Sword and SHIELD," July 2013, SIGGRAPH 2013 Tech Talk. [Online]. Available: http://www.nvidia.com/object/siggraph2013-tech-talks.html

[5] NVIDIA, "Tegra 4 Family GPU Architecture," Feb. 2013, Whitepaper V1.0. [Online]. Available: http://www.nvidia.com/docs/IO/116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf

[6] G. A. Paleologo, L. Benini, A. Bogliolo, and G. De Micheli, "Policy optimization for dynamic power management," in *Proc. 35th Annu. Design Automation Conf.* New York, NY, USA: ACM, 1998, pp. 182–187. [Online]. Available: http://doi.acm.org/10.1145/277044.277094

[7] L. Benini, A. Bogliolo, and G. D. Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.*, vol. 8, no. 3, pp. 299–316, Jun. 2000.

[8] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava, "Synthesis techniques for low-power hard real-time systems on variable voltage processors," in *RTSS*, 1998, pp. 178–187.

[9] E. Bini, G. Buttazzo, and G. Lipari, "Minimizing CPU energy in real-time systems with discrete speed management," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 4, pp. 31:1–31:23, Jul. 2009.

[10] D. C. Snowdon, S. M. Petters, and G. Heiser, "Accurate on-line prediction of processor and memory energy usage under voltage scaling," in *Proc. 7th ACM & IEEE Int. Conf. on Embedded Software*. New York, NY, USA: ACM, 2007, pp. 84–93. [Online]. Available: http://doi.acm.org/10.1145/1289927.1289945

[11] A. Musah and A. Dykstra, *Power-Management Techniques for OMAP35x Applications Processors*, Texas Instruments, Aug. 2008, Whitepaper.

[12] G. Kornaros and D. Pnevmatikatos, "A survey and taxonomy of on-chip monitoring of multicore systems-on-chip," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 2, pp. 17:1–17:38, Apr. 2013. [Online]. Available: http://doi.acm.org/http://dx.doi.org/10.1145/2442087.2442088

[13] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba, "The ACPI specification: revision 5.0," 2011. [Online]. Available: http://www.acpi.info/spec.htm

[14] J. Kurtto, "Mapping and improving the energy efficiency of the Nokia N900," Master's thesis, Department of Computer Science, University of Helsinki, 2011.

[15] Y. Zhang, X. Wang, X. Liu, Y. Liu, L. Zhuang, and F. Zhao, "Towards better CPU power management on multicore smartphones," in *Proc. of the Workshop on Power-Aware Computing and Systems*, ser. HotPower '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:5. [Online]. Available: http://doi.acm.org/10.1145/2525526.2525849

[16] V. Pallipadi and A. Starikovskiy, "The ondemand governor: past, present and future," in *Proc. Linux Symp.*, vol. 2, pp. 223–238, 2006.

[17] P. Greenhalgh, *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7: Improving Energy Efficiency in High-Performance Mobile Platforms*, Sept. 2011. [Online]. Available: http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf

[18] M. Kim and S. W. Chung, "Accurate GPU power estimation for mobile device power profiling," in *2013 IEEE Int. Conf. on Consumer Electronics (ICCE)*, 2013.

[19] Qualcomm, "Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age," Oct. 2011.

[20] S. Hong and H. Kim, "An integrated GPU power and performance model," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 280–289, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815998

[21] K. Pulli, J. Vaarala, V. Miettinen, T. Aarnio, and K. Roimela, *Mobile 3D Graphics: With OpenGL ES and M3G*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[22] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proc. 36th Annu. IEEE/ACM Int. Symp. on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 93–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956567

[23] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. 2010 USENIX Annu. Technical Conf.* Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855861

[24] M. Dong and L. Zhong, "Chameleon: A color-adaptive web browser for mobile oled displays," in *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2011, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000004

[25] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," in *Proc. 18th Annu. Int. Conf. on Mobile Computing and Networking*. New York, NY, USA: ACM, 2012, pp. 317–328. [Online]. Available: http://doi.acm.org/10.1145/2348543.2348583

[26] X. Chen, Y. Chen, Z. Ma, and F. C. A. Fernandes, "How is energy consumed in smartphone display applications?" in *Proc. 14th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '13. New York, NY, USA: ACM, 2013, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/2444776.2444781

[27] H. Han, J. Yu, H. Zhu, Y. Chen, J. Yang, G. Xue, Y. Zhu, and M. Li, "E3: Energy-efficient engine for frame rate adaptation on smartphones," in *Proc. 11th ACM Conf. on Embedded Networked Sensor Systems*. New York, NY, USA: ACM, 2013, pp. 15:1–15:14. [Online]. Available: http://doi.acm.org/10.1145/2517351.2517364

[28] "IEEE Standard for Information Technology – Telecommunications and Information Exchange Between Systems – Local and Metropolitan Area Networks-Specific

Requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, June 2007.

[29] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 6th ed. USA: Addison-Wesley Publishing Company, 2012.

[30] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proc. 6th Conf. on Computer Systems*. New York, NY, USA: ACM, 2011, pp. 153–168. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966460

[31] V. Venkatachalam and M. Franz, "Power reduction techniques for microprocessor systems," *ACM Comput. Surv.*, vol. 37, pp. 195–237, September 2005. [Online]. Available: http://doi.acm.org/10.1145/1108956.1108957

[32] D. Halperin, B. Greenstein, A. Sheth, and D. Wetherall, "Demystifying 802.11n power consumption," in *Proc. 2010 Int. Conf. on Power Aware Computing and Systems*. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924920.1924928

[33] C.-Y. Li, C. Peng, S. Lu, and X. Wang, "Energy-based rate adaptation for 802.11n," in *Proc. 18th Annu. Int. Conf. on Mobile Computing and Networking*. New York, NY, USA: ACM, 2012, pp. 341–352. [Online]. Available: http://doi.acm.org/10.1145/2348543.2348585

[34] M. O. Khan, V. Dave, Y.-C. Chen, O. Jensen, L. Qiu, A. Bhartia, and S. Rallapalli, "Model-driven energy-aware rate adaptation," in *Proc. 14th ACM Int. Symp. on Mobile ad hoc Networking and Computing*. New York, NY, USA: ACM, 2013, pp. 217–226. [Online]. Available: http://doi.acm.org/10.1145/2491288.2491300

[35] R. Friedman, A. Kogan, and Y. Krivolapov, "On power and throughput tradeoffs of wifi and bluetooth in smartphones," *IEEE Trans. Mobile Computing*, vol. 12, no. 7, pp. 1363–1376, 2013.

[36] C. Drula, C. Amza, F. Rousseau, and A. Duda, "Adaptive energy conserving algorithms for neighbor discovery in opportunistic bluetooth networks," *IEEE J. Sel. Areas Commun.*, vol. 25, no. 1, pp. 96–107, Jan. 2007.

[37] J. Liu, C. Chen, and Y. Ma, "Modeling and performance analysis of device discovery in bluetooth low energy networks," in *Global Communications Conference (GLOBECOM), 2012 IEEE*, 2012, pp. 1538–1543.

[38] M. Siekkinen, M. Hiienkari, J. Nurminen, and J. Nieminen, "How low energy is bluetooth low energy? comparative measurements with zigbee/802.15.4," in *Wireless Communications and Networking Conf. Workshops (WCNCW), 2012 IEEE*, 2012, pp. 232–237.

[39] Y. Xiao, Y. Cui, P. Savolainen, M. Siekkinen, A. Wang, L. Yang, A. Yla-Jaaski, and S. Tarkoma, "Modeling energy consumption of data transmission over Wi-Fi," *IEEE Trans. Mobile Computing*, vol. 99, no. PrePrints, 2013.

[40] "3GPP TS 25.331, Radio Resource Control (RRC); Protocol specification," May 1999.

[41] "3GPP TS 36.331, E-UTRA; Radio Resource Control (RRC) Protocol Specification," May 2008.

[42] F. Qian, Z. M. Mao, Z. Wang, S. Sen, A. Gerber, and O. Spatscheck, "Characterizing radio resource allocation for 3g networks," in *Proc. 10th Annu. Conf. on Internet Measurement*. ACM, 2010, pp. 137–150.

[43] N. Vallina-Rodriguez, A. Auçinas, M. Almeida, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft, "RILAnalyzer: A comprehensive 3G monitor on your phone," in *Proc.*

2013 Conf. on Internet Measurement ser. IMC '13. New York, NY, USA: ACM, 2013, pp. 257–264. [Online]. Available: http://doi.acm.org/10.1145/2504730.2504764

[44] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: A measurement study and implications for network applications," in *IMC*, 2009.

[45] "Docomo demands Google's help with signalling storm," [Online]. Available: http://www.rethink-wireless.com/2012/01/30/docomo-demands-googles-signalling-storm.htm, Jan. 2012.

[46] GSM Association, "Fast dormancy best practices. version 1.0," Jul. 2011. [Online]. Available: http://www.gsma.com/newsroom/wp-content/uploads/2013/08/TS18v1-0.pdf

[47] 3GPP, "3GPP specification TR 25.903: Continuous connectivity for packet data users," http://www.3gpp.org/ftp/Specs/html-info/25903.htm, Nov. 2005.

[48] M. Poikselk, H. Holma, J. Hongisto, J. Kallio, and A. Toskala, *Voice over LTE (VoLTE)*, 1st ed. John Wiley and Sons, Inc, 2012.

[49] A. Balasubramanian, A. LaMarca, and D. Wetherall, *Efficiently Running Continuous Monitoring Applications on Mobile Devices using Sensor Hubs*, Nov 2013, technical report. [Online]. Available: http://mobilehub.cs.washington.edu/papers/sensorhub.pdf.

[50] M.-R. Ra, B. Priyantha, A. Kansal, and J. Liu, "Improving energy efficiency of personal sensing applications with heterogeneous multi-processors," in *Proc. 2012 ACM Conf. on Ubiquitous Computing*. New York, NY, USA: ACM, 2012, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2370216.2370218

[51] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong, "Reflex: Using low-power processors in smartphones without knowing them," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 13–24, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2189750.2150979

[52] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl, "Energy characterization and optimization of image sensing toward continuous mobile vision," in *Proc. 11th Annu. Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys '13. ACM, 2013, pp. 69–82.

8 Mobile operating systems

Mobile operating systems are a relatively new development in the computing world. Mostly they are based on some existing older OS, typically written for stationary computers and laptops. For example, Android is based on the Linux kernel, iOS on the Mac OS X kernel, and Windows Phone on the Windows NT kernel. The mobile OS borrows many features from the desktop world; however, it needs to meet the demands of the mobile environment for communication needs, positioning of the device, power challenges caused by mobile operation with rechargeable batteries, and smaller size for the convenience and comfort of users. The OS is responsible for overseeing and managing the hardware and software components on the smartphone and it is responsible for the system-level power management.

In this chapter, we examine four current state-of-the-art smartphone OSs–iOS, Windows Phone, Firefox OS, and Android–as well as a number of energy-conserving research prototypes. We focus on the current generation of mobile OSs given our focus on smartphones. Classical examples of earlier feature phone OSs include the Symbian OS and Windows Mobile [1]. After examining the four smartphone OSs, we compare their features and properties.

8.1 Overview

Power management is an integral part of an OS that operates on multiple levels from drivers to applications. As the functionalities and features of mobile devices are constantly growing so is the demand for power. If the battery capacity is not increased, the software and hardware have to be more power-efficient to keep the same battery life when new features are being introduced to a device. For example, Linux makes use of several power-saving techniques implemented in hardware, such as

- scaling core voltage,
- gating of system clock,
- memory cache disabling,
- use of power and sleep modes.

As a classic example we can consider ACPI [2], which is aimed at personal computers. ACPI replaces the older APM solution and provides an industry-standard for OS-level device configuration and power management. The previous standard, APM, allowed control at the BIOS level, whereas ACPI allows this control at the OS level. In APM, the power management starts when the device becomes idle, and the OS has no control or knowledge over this power state change. In addition, ACPI provides a structured tree for shutting off devices which prevents turning off single components before their subsystem is powered down.

The smartphone OSs do not use ACPI; however, they employ techniques that are very similar. We will observe that iOS, Windows Phone, and Android all have a power-management entity within the OS that monitors and controls driver-level power management and allows component-level power management decisions. The OSs also expose battery information to application developers and employ a number of application-development patterns to save energy.

8.1.1 Mobile OS

Table 8.1 shows a high-level feature comparison of the four major mobile device operating systems: Android, iOS, Firefox, and Windows Phone. Android and iOS application ecosystems flourish and dominate at the moment, as shown in Figure 1.11. Windows Phone has been gaining ground and has been competing with Blackberry,[1] which is a popular platform especially for business users. Firefox OS is a new platform that is exclusively based on HTML5 [3]. In addition to these mobile OSs, the Linux Foundation is developing the Linux-based Tizen[2] for smartphones, in-vehicle infotainment systems, and other devices.

The table examines the typical development languages, network features, background processing support, push notifications, energy and power monitoring, HTML 5, open source nature, and third-party application installation processes. The OSs employ similar principles and patterns, although the implementations and APIs are very different.

The power managers of modern mobile OSs support component-level monitoring and configuration that allows the power state of a component to be changed. Hardware component dependencies are typically modeled with a tree or graph structure.

In general, Android has the least limited API set for gathering information about the network, battery, and system. The network and connectivity APIs of the four OSs support the key wireless standards, such as Bluetooth and Bluetooth LE, Wi-Fi, LTE, and NFC, but typically the APIs are limited to querying the properties of the current network and wireless connection.

Multitasking third-party applications is an important feature; however, it can lead to a significant energy draw due to intensive or frequent background activities. Typically, background tasks poll web resources, play audio content, or track on-board sensors, such as acceleration sensors or GPS. This observation has led to task-based multitasking, in which the developers use a multitasking API to create tasks that are run in the background. The multitasking API typically allows the developer to specify the nature

[1] `http://global.blackberry.com` accessed January 6, 2014.
[2] `http://www.tizen.org` accessed January 6, 2014.

Table 8.1. Mobile OS overview

	Android Linux	iOS	FireFox OS Linux	Windows Phone 8
Development	Java, native code with JNI and C/C++	Objective-C	Javascript and HTML5	C# and NET, various
Network features	Basic APIs: Bluetooth, Wi-Fi, cellular, NFC Network information Enumerate access points Signal strength	Basic APIs: Bluetooth, Wi-Fi, cellular, NFC Network information	No (only for pre-installed applications)	Basic APIs: Bluetooth, Wi-Fi, cellular, NFC Network information Set connection preferences
Background processing	Yes (services)	Task-based multitasking since version 4	No (planned)	Multitasking API
Push notification	Yes (Google Cloud Messaging)	Yes (iOs Push Notification)	Yes (Firefox Push Notification)	Yes (Microsoft Push Notification Service)
Energy and power monitoring	Battery status	Monitoring since 3.0	Battery status	Battery status
HTML 5	Yes	Yes	Yes	Yes
Open Source	Yes	No	Yes	No
3rd party application installation	Certificate, Google Play	Certificate, Apple AppStore	Certificate, app stores and web sites	Certificate, Windows Phone Store

of a task, for example a location-tracking task, polling task, audio playback task, or VoIP task. The OS scheduler can use this task-specific information to perform system-wide optimizations.

All four OSs support push notification that aims to increase application responsiveness and reduce energy consumption due to polling. The idea is to use a single connection for receiving push notification signals for multiple applications. The push services have a similar design, in which the mobile application provides a URI or a token to the web service that wants to send push notifications. This service then gives this information with the push message to the push notification service that is responsible for delivering the message to the proper device. The target device receives the push message through a long-lived connection with the push service. Once the message is received it is then locally routed to the proper application.

All four OSs provide basic battery information that includes the battery level, remaining operating time, and subscription of battery level changes or low battery situation.

Android provides a wealth of information through the BatteryManager and BatteryStats classes.

In addition to task-based multitasking and asynchronous events and push notification, wake locks are a frequently used pattern for preventing unnecessary use of resources. Windows Phone and iOS do not have explicit wake locks, but they allow the application to prevent the display from being turned off. Android, however, builds the power-management model on wake locks. The PowerManager uses wake locks to manage the power states of peripherals, such as the screen display. If a peripheral does not have a lock, it will be powered off and put to a low-power state to save energy. The PowerManager has an API that applications can use to manage these locks.

Next, we investigate the mobile OSs in more detail and then compare their power-management techniques and patterns.

8.2 iOS

iOS[3] is the mobile operating system from Apple used in the iPhone, iPad, iPod Touch, and newer Apple TV products. The OS was originally published on January 9, 2007 simultaneously with the first iPhone device. There have been many versions of iOS since the first release. The developer API has been actively improved by Apple over the years and numerous new features have been added. The first versions of iOS did not support multitasking for third-party applications; however, in 2010, iOS 4 offered multitasking support for application developers. The design aims to offer multitasking-specific APIs for background operations such as background audio play, VoIP, task completion, location services, and fast application switching. The goal was to optimize system performance, for example in VoIP applications the user may now receive calls in the background without causing delays.

The latest version is iOS 7 that extends the multitasking features by observing application usage patterns and allowing applications to update their content in the background before the anticipated usage time. The so-called coalesced update is also a new feature that groups application updates together to save energy by avoiding unnecessary radio usage. As discussed already in this chapter, going from idle mode to active network usage requires a significant amount of energy.

The three key iOS device types are:

- iPad is a tablet computer that runs Apple's iOS. iPad was originally released in April 2010. iPad has a multi-touchscreen and a virtual keyboard, built-in Wi-Fi, and on several models mobile connectivity.
- iPod is a portable media (music) player, originally released in October 2001. It has been modified and redesigned many times since the first version.
- iPhone is Apple's smartphone line. iPhone was originally released in June 2007 and there have been many generations of the product.

[3] `https://www.apple.com/ios/` accessed January 6, 2014.

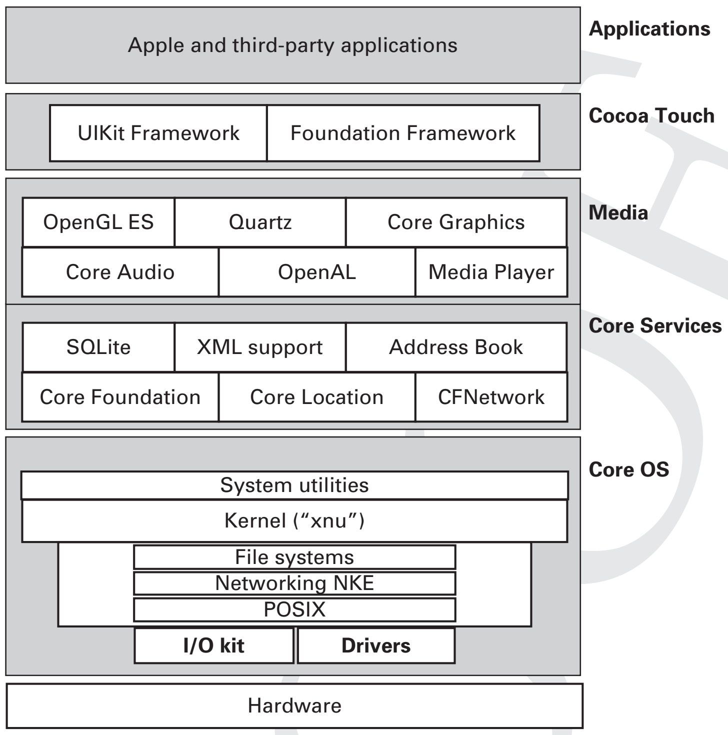


Figure 8.1 Overview of iOS

iOS has its roots in Apple's own operating system software, Mac OS X, which it resembles. The iOS kernel is similar to the Mac OS X kernel, but because it is compiled for the ARM CPU architecture, it has custom modifications. iOS uses the Core OS (Darwin) foundation and its architecture is built around the XNU kernel and system utilities. The general features of the XNU kernel include:

- POSIX support.
- Networking.
- File system support.
- Device drivers.

The iOS kernel is a hybrid which combines the Mach 3 microkernel and some elements from BSD Unix (Berkeley Software Distribution). The device drivers are implemented with an object-oriented API (I/O Kit). The iOS system utilities are the layer above the kernel. Altogether there are four abstraction layers in iOS outlined in Figure 8.1:

1. The Core OS layer which is the innermost layer and includes the kernel, TCP/IP networking stack, sockets interface, power manager, file system management, and security management.

2. The Core Services layer which is necessary for the Mac OS X application interfaces that are below the next two layers, for example APIs for networking, threads, and web. There is also an embedded SQLite database and geolocation tools on this layer.
3. The Media layer handles many I/O interfaces such as OpenGL, audio recording, video playback, and animation support.
4. The Cocoa Touch layer handles and manages multi-touch events and their control. It has also an interface for accelerometer input and supports features for camera and positioning. In modern devices, multi-touch gestures (tapping, swiping, pinching, and reverse pinching) are increasingly used in the user interface and iOS has the necessary tools. Also, in many applications the screen orientation is sensitive to the device y-axis rotation; this is accomplished with internal accelerometers.

The iOS SDK is used for both iPhone and iPad, because essentially iPad uses the same iOS as iPhone. The support given by the SDK also covers universal applications in addition to iPad and iPhone. A universal application uses the features available for the given device type, relying on conditional statements.

8.2.1 Energy-saving patterns

Key iOS energy-saving patterns include task-based multitasking, push notifications, asynchronous events, coalescing work, and coalesced updates.

In iOS, the power behavior of an application depends on its state, which can be foreground inactive, foreground active, background inactive, and background active. When an application is active and in the foreground, the system sends touch events to the application for processing. The UIKit takes care of the event delivery. The key application states are:

- Not running: application has not been launched or it was running and has been terminated.
- Inactive: the application is running in the foreground, but is not receiving events. Typically this state is only used for transitioning into other states.
- Active: the application is running in the foreground and processing events.
- Background: the application is in the background and running code.
- Suspended: the application is in the background and not executing code. The application is still in the memory in this state. When memory becomes low, the system may remove the application from memory.

Early iOS versions did not support multitasking for third-party applications. Since iOS 4, multitasking is supported for certain types of long-running tasks. The background modes for iOS 7 tasks are audio, location update, Voice-over-IP, newsstand downloads, external accessory communication, Bluetooth networking, Bluetooth data sharing, background fetch, and remote notification. Push notifications and asynchronous events are useful in reacting to different kinds of local and remote changes without the need to poll resources.

Work should be coalesced when possible to perform a set of operations as a batch rather than separately over a longer period of time. This allows the idle time of the device to be maximized and reduces unnecessary waking of the device to perform work. Coalesced updates is a new feature in iOS 7 that uses the wireless communication triggered by one application for the background transfers of other applications.

Generic programming guidelines recommend avoiding polling with events and timer-based scheduling. With event-based operation, the guidelines recommend disabling the delivery of events that are not needed and when the events are needed their delivery frequency should be set to the smallest feasible value. This is especially important for the UIAccelerometer class that is used to receive accelerometer events. Display wake locks can be realized through the idleTimerDisabled property of the shared UIApplication object. This property should be set to NO to ensure that the idle timer turns off the display when the timer fires.

8.2.2 I/O kit and power management

Mac OS X and iOS device drivers are developed using the I/O kit framework.[4] This framework consists of the necessary libraries and header files for drivers and userspace code to interact with kernel drivers. The I/O kit is responsible for the system's power management [4]. The power management is realized by kernel-level drivers. The power management is realized through the IOService superclass from which the drivers are derived.

Power management of devices is organized into a tree[5] called the power plane, in which parent nodes provide power for child devices. The power plane captures the power dependencies of the devices. For example, a Wi-Fi modem (a leaf) is connected to a bus (the parent). The IOService monitors and maintains a power hierarchy for the devices so that when a device is transitioning between power states, all the child devices have transitioned before the parent device. Power dependencies are taken into account in the ordering of I/O kit sleep and wake notifications. If a node in the power-management tree, the power plane, receives a sleep notification, it will first notify its child nodes to sleep before entering the sleep state. A wake notification, on the other hand, first wakes the parent before waking the children.

Power can be saved by powering down an idle device. The IOService class provides a timer facility for triggering a lower power state in devices that have not been used recently. A driver can install a timer that expires after a duration has passed. This timer determines how long the device can be idle at full power before transitioning into a low-power state. In addition to the timer, the driver can inform the I/O kit when the hardware was last used. This trigger is used to reset the timer when the hardware is used.

The I/O kit power management is based on devices, power states, and attributes. Each device has at least two power states, on and off, and attributes to these states. The device driver is responsible for setting the attributes that determine the capabilities and

[4] Documentation available at: `https://developer.apple.com/` accessed January 6, 2014.
[5] It is possible for a device to have two parents making this a graph. In this case the ordering is not guaranteed.

requirements of the device. The attributes determine the capability in a power state, the device's power requirements from its parent, and the power characteristics it can provide to its children. These attribute fields are contained in the IOPMPowerState structure and allow the specification of the device's power state budget in milliwatts as well as the time in microseconds required to attain, settle up, lower, and settle down the power state.

The drivers need to respond properly to system sleep and wake events and they may also support other modes, such as an idle state. OS X defines two different types of drivers: passive and active. A passive driver implements the basic power-management features and it carries out custom power-saving actions. An active driver does this and can also carry out advanced power-management features, such as transition between the device's power states.

8.3 Windows Phone OS

Windows Phone[6] was launched in October 2010 as the successor to the Windows Mobile OS. The current version is Windows Phone 8, released in October 2012. The current kernel in Windows Phone 8 is based on the successful NT kernel of older Microsoft operating systems. It is a 32-bit pre-emptive multitasking OS. The Windows Phone kernel has been divided into two parts, one sharing code with Windows and the other part having mobile-specific changes. The former is called the Windows Core System and the latter the Mobile Core.

Only these two components, the core system and mobile core, are allowed to share code between the mobile and static Windows systems. APIs may sometimes be similar but the actual code in the APIs is different in Windows than in Windows Phone. The actual Windows Phone system lies above the core with preset applications like music, video tools, connection management, phone shell, and various platform services. The platform provides services for the application level and is partitioned into four components illustrated in Figure 8.2:

- Package Manager takes care of the application during its lifetime in the Windows Phone system. It handles the installation and uninstallation and stores the metadata related to the particular application, including the registered pinning and extension point information.
- Execution Manager manages the execution of the applications. It also handles all background agents and their logic. Execution Manager creates the required host processes and initiates state messages for applications, for example startup, shutdown, and stopping an application.
- Navigation Server is needed to take care of the moving of focus between foreground applications installed on the phone. This is accomplished by sending commands to the Execution Manager regarding applications to launch or applications to reactivate. Navigation Server also records the status of the navigation stack.

[6] `http://www.windowsphone.com/` accessed January 6, 2014.

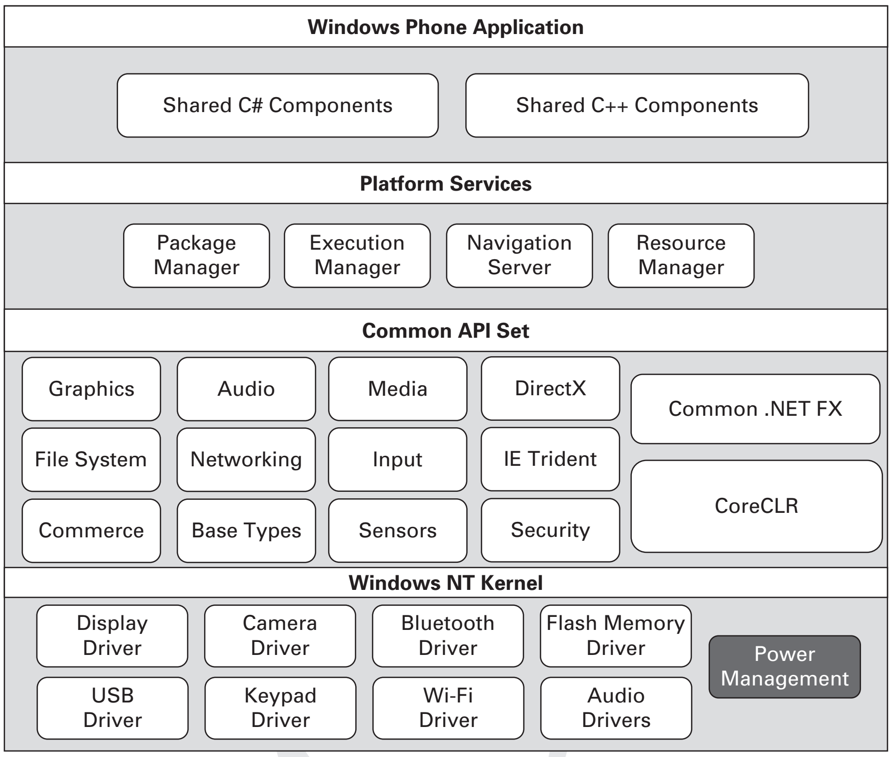


Figure 8.2 Overview of Windows Phone

- Resource Manager is the service that keeps track of the resources of all active system processes. It can enforce constraints on processes while focusing on CPU and memory. Resource Manager is also responsible for terminating misbehaving applications to try to guarantee the stable operation and response of the phone.

The .NET Common Language Runtime (CLR) is responsible for executing applications. This is important because the application's overall safety and specifically the safety against buffer overflow (which can be disastrous) must be handled carefully and the application verified correctly. The runtime executes each Windows Phone application in its own isolated chamber that is based on the capabilities required by the application. A basic set of permissions is given to the application chamber and this basic set can be extended by capabilities that are granted when the application is installed. It is not possible to add permissions after the installation [5].

8.3.1 Energy-saving patterns

The key patterns for application developers include task-based multitasking, push notifications, and display requests to keep the display active. The multitasking API is used to implement background agents that can be periodic or resource intensive. Periodic tasks are useful for short regularly occurring activities, such as sending the device's location to the cloud. Resource-intensive tasks are longer running activities that require certain requirements to be met for the processor activity, power source, and network status. Push notifications allow the asynchronous activation of an application through a short push message sent by a service. Instead of having a push communication channel for each application, the OS and middleware maintain a dedicated connection, typically a TCP connection, for the applications resulting in significant energy savings. An application can also request that the display is kept on when a long video is being watched. This allows application-specific overriding of the default display power-saving policy that would otherwise dim or turn off the display [6].

8.3.2 Windows 8 Power Architecture

The Windows 8 Power Architecture builds on CPU and device power states, through ACPI [2] or drivers, and defines a runtime power-management framework (PoFx) that supports component-level power and clock management. Windows Phone 8 shares codebase with Windows 8 and builds on the same architecture; however, it does not feature ACPI.

The key aim of PoFx is to provide the fine-grained control mechanisms to support battery-powered devices always on always connected devices such as laptops, tablets, and smartphones [7]. A device driver needs to register with PoFx to manage the power usage of a component or subsystem.

The earlier versions of Windows supported power management at the device level. PoFx extends this with component- and subsystem-level power management. Similar to ACPI D- and S-states, on the device-level we have the D-states (D0 is the on state) and on the component-level we have the F-states (F0 is the on state). PoFx maintains system-wide information about the components and their power and clock domains. Device drivers can provide component status and capability information to the PoFX through a device driver interface (DDI). The component-level information includes the state and activity level, state transition time, and the latency tolerance of the clients of the component when waking from a low-power state. Given the system-wide and component-level information, PoFX makes power optimization decisions by controlling the state transitions of the components. These decisions require component dependencies as well as power and clock domains to be taken into account in the decision making [7].

The key components in the Windows 8 power-management architecture are the following:

- Power Manager that is responsible for managing the power usage of the system. This manager maintains the system-wide power policy. The manager interacts with device drivers and uses them to control component power state transitions. The power manager considers the system activity level, battery level, power-related requests (shutdown, hibernate sleep), and control panel settings.
- ACPI driver that is part of the OS (not supported by Windows Phone).
- Drivers provide information to the power manager, respond to power requests, and manage the power states of individual devices.

8.4 Firefox OS

Firefox OS[7] is an operating system for smartphones and tablet computers.[8] In the beginning it was called Boot to Gecko (B2G) and its goal is to augment HTML5 [3] applications in their direct interaction with the device hardware. In other words, HTML5 applications are basically run as native applications by FirefoxOS.

Firefox OS is a Linux-based open-source smartphone platform developed by Mozilla, the non-profit organization that designed the Firefox browser. Firefox OS was released in February 2012 and it offers a community-based alternative OS for mobile devices, because it uses open standards, such as HTML5, JavaScript, and open Web APIs, for direct communication with cellphone hardware. It is a competitor for proprietary mobile OSs, such as iOS and Windows Phone. As an open-source system it is not unique because it also competes with Android and Ubuntu Touch. There are several companies developing mobile phone products based on the OS.

The three main software layers in Firefox OS are:

1. Gonk includes the underlying Linux kernel and hardware abstraction layer (HAL). Open source concept covers the kernel and many of the user libraries. It is notable that there are many common shared modules with the Android project, such as the GPS and camera. Gonk was designed to be fully open to Gecko with all its features and interfaces; Gecko cannot currently access other mobile OSs the same way.
2. Gecko is the runtime engine handling services for applications (actually it is a modified version of the rendering engine of Firefox). Gecko implements open standards, such as HTML, CSS, and JavaScript, and contains several critical components: stacks for networking and graphics, layout engine, JavaScript virtual machine, and porting layers. It provides APIs to access the phone hardware; the interface to other operating systems and browsers is uncomplicated because Gecko implements only standard Open Web APIs.
3. Gaia is the HTML5 layer and user-interface for Firefox OS, managing everything that appears on the screen. Gaia is coded entirely in HTML, CSS, and JavaScript and has a variety of default applications.

8.4.1 Power management

The Firefox OS Power Management Web API consists of tools for managing a device's power settings. This API is only available for certified applications that are pre-installed on Firefox OS devices. The API is non-standard at the moment. The API includes screen power operations including setting the screen brightness, CPU power operations, and advanced power features relating to wake locks. An application can request a wake lock for a screen, CPU, or Wi-Fi resources.

[7] `http://www.mozilla.org/en-US/firefox/os/` accessed January 6, 2014.
[8] `http://developer.mozilla.org/` accessed January 6, 2014.

8.4.2 Battery status API

Firefox OS also supports the battery status API from W3C [8]. This specification defines the API calls for obtaining information about the battery status and condition. The API allows querying of the charging status, charging time, discharging time, and battery level of a device. The API also supports event handlers for tracking changes in the variables.

8.5 Android

Android[9] is a Linux-based operating system and software platform for mobile devices, developed by Google and the Open Handset Alliance[10] in 2007. The Android platform accepts code written in a Java-like language, following the common Java syntax. However, Android does not have a standard Linux kernel and it does not provide the standard class libraries and APIs, accepting only libraries and APIs developed by Google. The two newest versions of Android are 4.4 KitKat and 4.3 Jelly Bean, released in 2013.

Components (layers) of the Android architecture are illustrated in Figure 8.3 and they can be summarized as follows:

- A customized Linux kernel that changes with major versions of Android (meaning that old Android instances run on new kernels but not vice versa).
- An augmented set of hardware drivers (display, keypad, audio, communication). The kernel additions have been incorporated under GNU licence to the open source community.
- The kernel interface was coded in C and C++. It contains a set of open source C/C++ libraries for the components of the system (e.g., WebKit, libpng, and libsqlite)
- The Android application framework (set of APIs) to use the libraries.
- Core Libraries and the Dalvik Virtual Machine in the runtime component. Dalvik is a register-based process virtual machine which executes Dalvik Executable Format (DEF) code.
- Android runtime that executes the custom Java code. Applications for Android are typically written in Java and compiled to a Java Virtual Machine (JVM) bytecode (itself again compiled into DEF when installed on the device).
- Set of application managers on top of the libraries.
- Set of bundled applications on top of application managers. (e.g. browser, email client, SMS program, maps)

Specifically, Linux was enhanced with the following components:

- Alarm driver with timers for waking sleeping devices.
- Memory management with Android shared memory driver (ashmem). This module enhances the ability of applications to share memory and also manages the sharing in the kernel.

[9] `http://www.android.com` accessed January 6, 2014.
[10] `http://www.openhandsetalliance.com` accessed January 6, 2014.

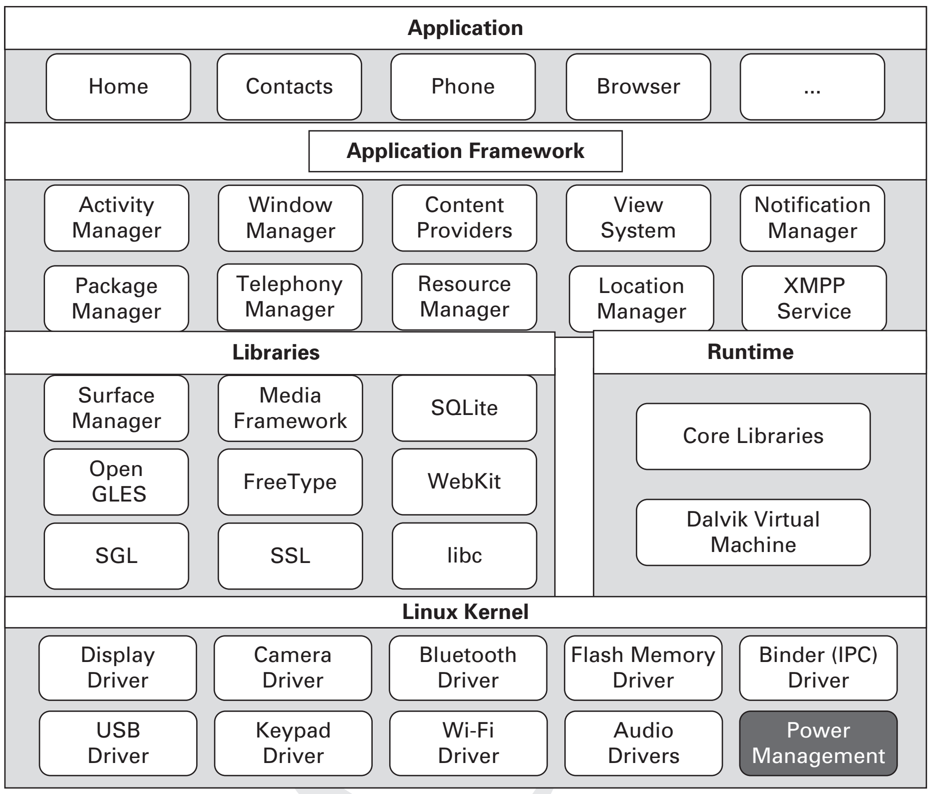


Figure 8.3 Overview of Android

- IPC (Inter-Process Communication Interface). The binder driver included enhances inter-process communication. Data sharing is possible for several applications simultaneously using the shared memory. The binder handles and monitors threads so that services registered as an IPC service can leave managing functions to the IPC module, which also synchronizes processing,
- Standard Linux Power Management has been enhanced with Android Power Management that defines an active policy for managing and saving power.
- Low memory process killer
- Kernel debugger
- System logger.

8.5.1 Android power management

It is notable that many of the Android's kernel components are inherited from the Linux OS and several power-management components derive originally from Linux. But Linux as an OS was mainly designed for stationary computers before the era of ubiquitous mobile devices and therefore it is not optimized for phones or tablets in any way. Yet the mobile devices are notoriously both resource-hungry and limitation-prone at the same time, making it critically important to optimize the OS for mobile usage. Power saving is perhaps the central point of importance for users and therefore for designers of applications, hardware, or the OS.

Suspend and hibernation as power-saving schemes inherited from the laptop world are not sufficient for smartphones or for embedded systems. Mobile phones have a limited power supply in the form of their battery capacity, which creates a hard constraint for the design process. Battery technology needs to develop, but also the other system components should be able to help conserve power irrespective of the battery technology. Android's power-saving methodology aims just to make the usable life of a mobile device between battery charges longer and richer.

PowerManager

Contrary to desktop Linux systems, Android does not use APM or ACPI for power management, but instead has its own Linux power-management extension called PowerManager[11] that conserves battery life by turning off components while allowing developers to selectively prevent these actions. Figure 8.4 illustrates the Android Power Management framework. The PowerManager is implemented in Java and it interfaces the OS and kernel-level power management features using Java Native Interface (JNI) calls.

PowerManager uses wake locks to manage the power states of peripherals, such as screen display, backlight, and keyboard backlight. The PowerManager provides an API for requesting and releasing wake locks relating to peripherals. If a there is no lock for a given peripheral, it will be powered off and kept in a low-power state to conserve energy. The wake lock API has two parts: driver and userspace.

Applications can request that a certain peripheral is kept on through the API. Applications can also set partial wake locks that ensure that the CPU is running; however, the screen display and the keyboard backlight are turned off according to the current policy. A partial wake lock ensures that the CPU is on even if the user presses the power button. With full wake locks, pressing the power button places the device into a sleep mode.

To make use of this facility, the Linux kernel was extended with a power driver module that contains a set of low-level drivers specifically for controlling the power state of supported peripherals. In addition to supporting wake locks, any driver can register its own early suspend and late resume handlers. This allows drivers to handle power mode configuration to the device before the kernel is suspended. In addition to the CPU, the default peripherals that support wake locks are:[12]

- backlight of display panel,
- backlight of keyboard,
- backlight of buttons.

[11] `http://developer.android.com/reference/android/os/PowerManager.html` accessed January 6, 2014.

[12] Recent Android API recommends using the FLAG_KEEP_SCREEN_ON flag instead of wake locks for the display.

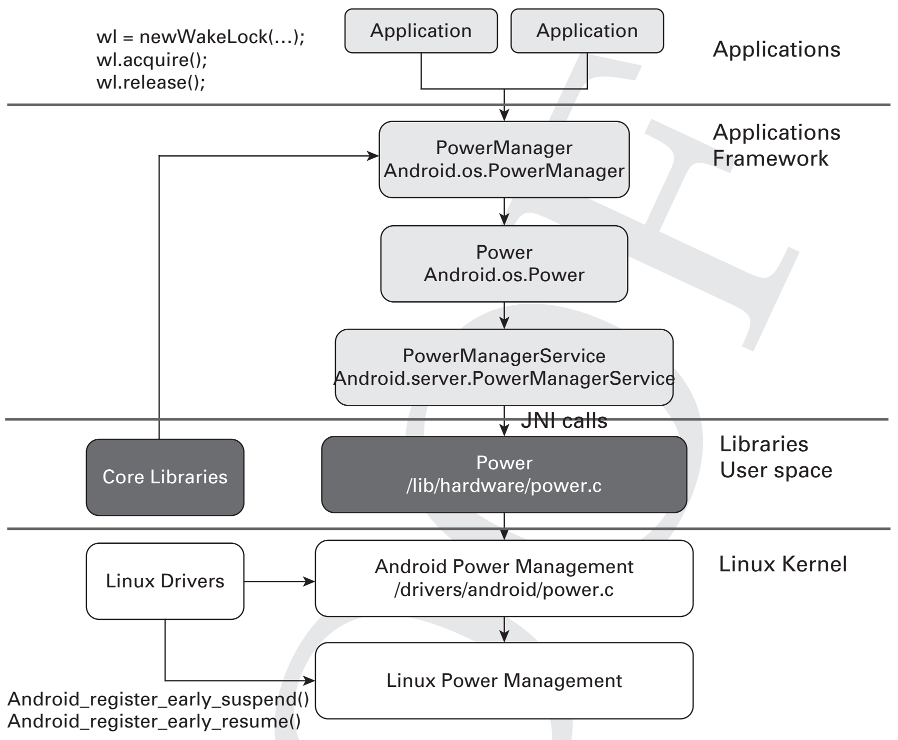


Figure 8.4 The Android power-management system

WakeLocks

Android controls power for the peripherals using locks (WakeLocks). A lock is set up when an application wants a certain peripheral to remain active. Android checks the states of locks and turns off all devices which are not locked in the on-state (unlocked devices). PowerManager also monitors the device status, the battery condition, and the circuitry charging the battery, so it can decide whether the system must be powered down at some critical preset battery charge threshold.

All application calls relating to power management are routed through the Android runtime PowerManager API, as illustrated in Figure 8.4. The native libraries supporting the Android Runtime module are not allowed to call the power management services, such as WakeLocks, directly because this would make the system unstable. The Android PowerManager allows kernel drivers to register to receive notifications of power up/down requests from the use space.

From the application perspective, wake locks are used as follows[13]:

- The application must get a handle to an instance of a PowerManager service. This is accomplished by calling Context.getSystemService().

[13] `http://www.kandroid.org/online-pdk/guide/power_management.html` accessed January 6, 2014.

- Application creates a wake lock and specifies the power-management flags for devices.
- Application activates the wake lock.
- Application performs the desired operation.
- Application releases the wake lock.

Care must be taken when using wake locks to ensure that energy is not spent unnecessarily in high-power states. Wake lock bugs have been observed frequently in Android applications [9, 10]. According to Pathak et al., a no sleep bug occur when an application becomes active, sets a wake lock, and then never releases the wake lock preventing the system from sleeping. The bug can occur because the application never releasing the lock, a race condition where the acquisition and releases happen in the wrong order due to the race condition of threads, or when the release of the lock is delayed [9].

BatteryStats

The Android power-management system also monitors the battery life of the device and implements a power-saving policy [11]. The PowerManager keeps track of the energy consumption with a component called BatteryStats that is examined in Chapter 10. BatteryStats performs time-based book-keeping of applications that is used to estimate their energy consumption.

8.5.2 New sensing features

The latest KitKat version has new energy-saving mechanisms, such as the sensor batching and step detector and counter features. The sensor batching allows Android to use the device hardware to obtain and deliver sensor events in batches. This allows the application processor to spend longer periods in lower power states and then react to sensor batches. The API uses a standard event listener and the batching interval can be set by the developer. Immediate delivery of certain requested events between batches is also supported. The sensor batching supports various sensing applications relating to location and context. The step detector and step counter features are composite sensors that allow applications to track steps when the user is walking, running, or climbing. Android devices will implement these sensors in hardware for low power consumption.

8.6 Energy-aware OS research prototypes

The current mobile operating systems employ many techniques for power saving, such as scheduling, sleeping, and resource-management techniques applied on multiple levels. Various dynamic power-management schemes have been proposed that are used to control the power consumption of system components [12]. The idea of an energy-aware operating system has been studied in the research community since the late 1990s. The Odyssey proposal was one of the first to investigate energy efficiency. At the time, energy was seen as a key limiting resource that should be minimized without sacrificing performance. This research waned during the next ten years, but interest was rekindled with the advent of smartphones and mobile applications. Energy efficiency has become a key requirement during this decade with many new proposals also in the operating system area [13].

8.6.1 Odyssey OS

The Odyssey OS and framework enable applications to gracefully degrade the data fidelity to save energy. The key insight of this system is to dynamically modify application behavior to conserve energy. This is achieved through the OS that monitors energy consumption and supply, and sets the application parameters according to an energy policy. The Odyssey OS uses an offline approach to energy management [14]. This approach is based on the PowerScope profiler that creates a process-based energy consumption profile [15]. The PowerScope approach employs an external power monitor tool for accurate device-specific energy consumption data that is then correlated with the process lists to understand the relations between software processes and energy consumption.

Figure 8.5 illustrates the Odyssey architecture. Concurrent adaptation of diverse applications is supported through the OS and kernel. Odyssey is conceptually an OS-level system; however, it was implemented in the userspace with an interceptor module in the kernel. The viceroy component is responsible for resource monitoring and management. The wardens encapsulate various data types, such as video, image, and audio content. Odyssey uses a data-specific warden to decide at what level of fidelity applications should be executed. Odyssey does not require applications to extrapolate future power requirements, but instead uses smoothed observations of present and historical power usage to predict near-future behavior. The empirical results indicate a 30% energy saving when degrading the image and video quality of various workloads.

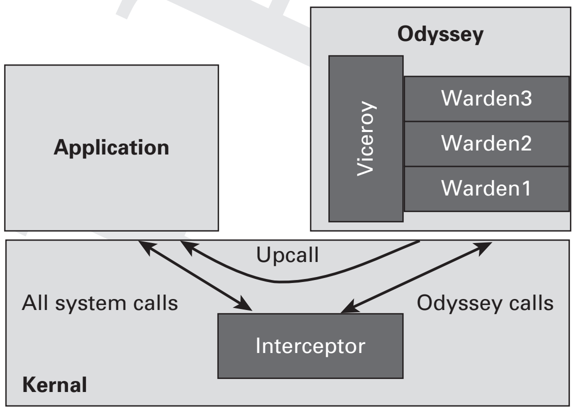


Figure 8.5 Overview of the Odyssey architecture

8.6.2 ECOSystem

ECOSystem is an early example of an energy-aware OS [16]. This system allows users to control per-application energy expenditure with energy allocation and accounting. The key abstraction is called currentcy. This empowers applications to spend a certain amount of energy up to a fixed limit. This enables application-specific energy budgets that the OS monitors and enforces.

The overall goal is to achieve a user-specified battery lifetime through limiting energy consumption. The system divides time into epochs and for each epoch a total amount of currentcy is allocated, determined by the discharge rate needed to achieve the target battery lifetime. If the amount of currentcy is less than 100% some components are throttled. The available currentcy is split between the competing applications according user-specified proportions. For each epoch, an application receives an allowance of currentcy based on its proportional share. The currentcy energy budgets are implemented with resource containers that are given currentcy allocations, and that are charged by the hardware resources that they use. The hardware resources have their own charging policies.

The currentcy budget is determined based on the battery lifetime target and application-specific proportional currentcy allocations. User preferences are reflected in the application specific allocations, for example preference for interactive performance instead of background performance. The limitation of this approach is that modern mobile applications are complex and use system features and components in complex ways.

8.6.3 Cinder OS

The Cinder OS is an example of a more recent proposal of an energy-efficient mobile OS [17]. This system builds on the HiStar exokernel and introduces device-level energy budgeting and accounting. The energy consumption is estimated using standard device-level techniques. The explicit flow control of HiStar allows fine-grained tracking of resources use even across interprocess communication calls.

The system builds an offline energy model that is used to estimate the energy consumption of system resources. The Cinder system monitors how applications use system resources and correlates this usage to energy consumption. Cinder can amortize costs across principals so that, for example, the costs due to multiple applications using internet services can be determined even when they share some of the hardware resources. This approach lends itself well to supporting energy budgets and limits for applications. Cinder uses an abstraction called reserve for modeling the energy consumption of the device and its components. The system forms a resource consumption graph with the battery as the root reserve of the graph. When applications use resources, the corresponding reserves are consumed. The scheduler only schedules those threads for running that have sufficient reserves. A second abstraction called the tap is used to specify the rate of reserve consumption from one reserve to another.

The three key energy-management principles in Cinder are: isolation, delegation, and subdivision. Isolation is a fundamental part of an OS and for Cinder this relate

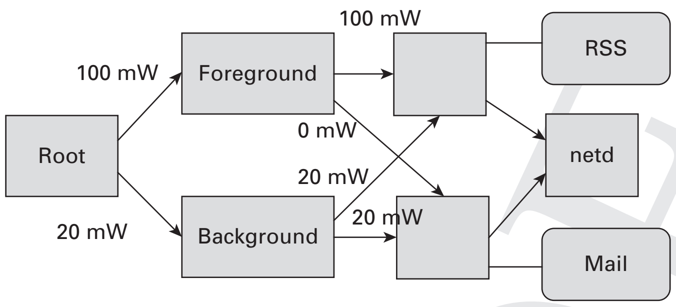


Figure 8.6 Example of Cinder reserves and taps

to application- and resource-specific energy budgets. Delegation allows applications to reallocate their energy budgets to other applications and components. Subdivision allows the budget to be divided into smaller parts. Subdivision combined with delegation allows an application to share its energy while maintaining an internal energy budget.

Cinder's CPU scheduler is energy-aware and throttles threads that have empty energy reserves. Energy in reserves can be delegated and subdivided. For example, a thread can delegate energy to another thread. An application can also subdivide its reserve to multiple subreserves allowing other threads to connect to these. Threads typically manage energy with taps that allow reserve-to-reserve energy transfers.

Figure 8.6 illustrates the reserves and taps with two applications: an RSS application running in the foreground and an email application running in the background. Both applications have been given energy reserves from the root reserve. The foreground applications has a budget of 100 mW and the email application has a budget of 20 mW. The task manager has given the foreground application an energy tap of 20 mW from the background application, to ensure that the foreground application has sufficient energy to meet the user's expectations and to be responsive. Both applications use the network so the netd network daemon transfers energy from their reserves into its own reserve. When the combined reserves of the requesting application and the netd reserve have enough energy, the radio subsystem will be started. Thus the applications' network access is synchronized resulting in less energy waste.

8.7 Summary

In this chapter, we have surveyed four well-known operating systems for mobile devices: iOS, Android, Windows Phone, and Firefox OS. The systems are based on three different underlying kernels: MacOS, Linux[14], and Windows NT. Android and Firefox OS are examples of smartphone platforms based on open source, whereas iOS and Windows Phone are closed source. Each kernel and OS has its own power-saving

[14] `kernel.org` accessed January 6, 2014.

schemes based on well-known patterns and solutions. All of the above systems use dynamic power-management techniques, such as DVFS and DPS.

The OS monitors and controls the power parameters of the device and implements an energy-optimization policy. The OS provides the necessary API for drivers to interface with the OS scheduler and power manager as well as APIs for application developers. The power managers found in modern mobile OSs support component-level monitoring and configuration, and typically model the component dependencies through a tree or graph structure. Task-based multitasking, push notification, and the wake lock pattern frequently use developer API-level solutions for reducing power consumption. Additional techniques include sensor data batching and asynchronous notification of status changes.

Table 8.2 presents a comparison of the well-known mobile operating systems and their power-management features and patterns. We examine the four OSs discussed in this chapter as well as the Symbian Series 60 that is a classic example of a mobile energy-efficient OS [18].

The table examines the low-level and high-level power management, energy-conservation patterns, policies, and battery information. The kernel is responsible for the low-level power management. This is realized by a power-management component within the OS that interfaces with the device drivers. As discussed in this chapter, the modern OSs support a tree-or graph-based component model that allowing fine-grained power management.

The Symbian is a classic example of an OS designed for phones that have limited batteries. This OS has a kernel-side framework with a power manager that monitors and controls system components. The manager is responsible for the management of the transitions between processor and peripheral power states as well as managing the components' power requirements. The key power states are off, standby, and active [19].

The energy conservation patterns include wake locks, multitasking APIs, push APIs, coalesced updates, sensor batching, and asynchronous events including wakeup events. In addition, the Symbian OS has specific design patterns for coping with limited devices, such as the active object pattern that uses a single thread of execution for multiple objects to conserve threads.

All the OSs provide the basic battery information through an API. Android and Symbian provide the most information; however, none of the battery APIs guarantee to provide reliable current and voltage data. We examine Android BatteryStats that performs rudimentary time-based book-keeping of applications that is used to estimate their energy consumption, later in Chapter 10. The Nokia Energy Profiler was one of the first tools for on-device power profiling [20]. We survey smartphone energy-profiling techniques in Chapter 10.

A number of energy-aware OS research prototypes have been proposed. These systems include techniques such as data fidelity tuning based on energy targets, and fine-grained energy accounting and budgeting of how applications use system resources. Table 8.3 compares the three examined energy-aware OSs: Odyssey, ECOSystem, and

Table 8.2. Mobile OS feature comparison

	Android Linux	**iOS**	**Windows Phone 8**	**Firefox OS**	**Symbian Series 60**
Low-level power Management	Linux Power Management	iOS kernel	Windows NT	Linux Power Management (Gonk)	Kernel-side framework with power API (Power Manager), peripheral power on/off
High-level power Management	Java class PowerManager, JNI binding to OS. Key methods: goToSleep(long), newWakeLock(...), userActivity(long...) BatteryStats monitory energy consumption and uses device specific subsystem models.	I/O Framework	Runtime power management framework	Gaia (OS Shell) Gecko runtime: Power Management Web API is non-standard and reserved for pre-installed applications	Application use domain manager that follows system-wide power-state policies. Nokia Energy Profiler API
Energy conservation patterns	Wake lock (partial, full) is used to ensure that device stays on. Methods: Create, acquire, release, sensor batching	Coding patterns, multitasking API (since iOS 4), push API, coalesced updates (since iOS 7)	Multitasking API (tasks and push notification), asynchronous events	Asynchronous events (system messages) Resource lock in Power Management API	Active object, wakeup events, resource and domain manager

(*cont.*)

Table 8.2 (*cont.*)

	Android	**iOS**	**Windows Phone 8**	**Firefox OS**	**Symbian Series 60**
Policies	Wake lock specific flags and policies, system-wide power setting	Internal, multitasking API (since iOS 4)	Internal	Gonk and Gecko level	Domain manager for system-wide and domain-wide policy. Domain-specific policies are possible
Battery information	The BatteryManager class contains strings and constants for different battery-related notifications that applications can subscribe to, includes: battery level, temperature, voltage	iOS 3 and later: UIDevice Class can query/subscribe battery info	Battery class provides the battery level, remaining operating time, and an event when battery is below 1%	W3C Battery Status API	Battery API (charge level, external power). Nokia Energy Profiler

Table 8.3. Comparison of energy-aware OSs

OS	Description	Accounting and budgets	Proportionality	Application support
Odyssey	Change fidelity to achieve target lifetime	Energy supply and demand are monitored for triggering adaptation	User preference based (priority)	Wardens encapsulate data types
ECOSystem	Target lifetime by limiting discharge rate	Epoch based	User preference based	User preferences as input for competing applications, Kernel implementation
Cinder	Fine-grained resource accounting	Reserve, taps, and energy consumption graphs.	Resource sharing based (graphs)	Reserves, taps, and policies for developers. Command line tools

Cinder. Odyssey uses data fidelity adaptation to meet an energy target. ECOSystem and Cinder aim to achieve OS-level accounting and budgeting of energy. Compared to ECOSystem, the newer Cinder proposal has a more advanced energy-budgeting model based on reserves and taps organized as a graph. This model allows fine-grained modeling of inter-process calls and resource sharing within a mobile device.

The main benefit of Odyssey is that the device lifetime can be increased by adjusting data fidelity while taking user experience into account. ECOSystem's benefit is that it allows resources that exceed their energy budgets to be stopped, thus making it possible to set the budgets to meet the desired battery lifetime. Cinder's main benefit is having a more sophisticated accounting and budgeting model that can cope with shared resources and more complex dependencies.

References

[1] S. Tarkoma and E. Lagerspetz, "Arching over the mobile computing chasm: Platforms and runtimes," *Computer*, vol. 44, no. 4, pp. 22–28, Apr. 2011. [Online]. Available: http://dx.doi.org/10.1109/MC.2010.272

[2] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba, "The ACPI specification: revision 5.0," 2011. [Online]. Available: http://www.acpi.info/spec.htm

[3] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer, and I. Hickson, "HTML5: A vocabulary and associated APIs, for HTML and XHTML," W3C, Tech. Rep., Aug. 2013, Candidate recommendation. [Online]. Available: http://www.w3.org/TR/html5/

[4] O. Halvorsen and D. Clarke, *Power Management*. Apress, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-1-4302-3537-8_10

[5] *Windows Phone 8 Security Overview*, Microsoft, Oct. 2012 [Online]. Available: http://blogs.msdn.com/cfs-filesystemfile.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-01-55-06/8272.20_2C00_206.01_5F00_WP-8_5F00_SecurityOverview_5F00_102912_5F00_CR.pdf.

[6] *Windows Phone Dev Center: DisplayRequest Class*, Microsoft, Oct. 2013 [Online]. Available: http://msdn.microsoft.com/en-us/library/windowsphone/develop/windows.system.display.displayrequest.aspx

[7] Microsoft, *Overview of the Power Management Framework*, Nov. 2013 [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/hh406637(v=vs.85).aspx

[8] A. Kostiainen and M. Lamouri, *W3C Battery Status API*, Apr. 2013, W3C Editor's Draft April 19, 2013.

[9] A. Pathak, A. Jindal, Y. C. Hu, and S. Midkiff, "What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps," *Mobisys*, 2012.

[10] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards verifying android apps for the absence of no-sleep energy bugs," in *Proc. 2012 USENIX Conf. on Power-Aware Computing and Systems*, ser. HotPower'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387869.2387872

[11] *Power Profiles for Android*, Android Open Source Project, Nov. 2013 [Online]. Available: https://source.android.com/devices/tech/power.html.

[12] G. A. Paleologo, L. Benini, A. Bogliolo, and G. De Micheli, "Policy optimization for dynamic power management," in *Proc. 35th Annu Design Automation Conf.* New York, NY, USA: ACM, 1998, pp. 182–187. [Online]. Available: http://doi.acm.org/10.1145/277044.277094

[13] N. Vallina-Rodriguez and J. Crowcroft, "Energy management techniques in modern mobile handsets," *IEEE Communications Surveys Tutorials*, vol. 15, no. 1, pp. 179–198, 2013.

[14] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 48–63, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/319344.319155

[15] ——, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proc. 2nd IEEE Workshop on Mobile Computer Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=520551.837522

[16] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, "Ecosystem: managing energy as a first class operating system resource," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 123–132, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/635508.605411

[17] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich, "Energy management in mobile devices with the cinder operating system," in *Proc. 6th Conf. on Computer Systems*. New York, NY, USA: ACM, 2011, pp. 139–152. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966459

[18] A. J. Issott, *Common Design Patterns for Symbian OS: The Foundations of Smartphone Software*. Wiley Publishing, 2008.

[19] J. Sales, *Symbian OS Internals: Real-time Kernel Programming*. Wiley Publishing, December 2005.

[20] G. Creus and M. Kuulusa, "Optimizing mobile software with built-in power profiling," in *Mobile Phone Programming*, F. H. Fitzek and F. Reichert, Eds. Springer Netherlands, 2007, pp. 449–462. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-5969-8_25

9 Power modeling

Our studies of power measurement have revealed that the power consumption of a smartphone varies with the applications in use, the way the device user uses the applications, and the environment where the smartphone is operated. As power measurement only tells us how much energy is being consumed by the whole smartphone or a hardware component such as the display, more information is required for analyzing the factors that affect the power consumption.

Power modeling is a technique that has been developed for quantifying the impact of different factors using mathematical models. A power model can be specified for a certain hardware component, a certain smartphone, or a certain piece of software. The information used for defining the model variables can be provided by hardware, OS, and/or applications, while the coefficients of these variables can be derived from power measurement using deterministic and/or statistical methods. The methodology of deterministic and statistical power modeling is introduced in Section 9.1, followed by three case studies: a deterministic power model of a Wi-Fi network interface (Section 9.2), a statistical model of the overall power consumption of a smartphone (Section 9.3), and a fine-grained energy profiler using system call traces (Section 9.4).

Power models can be used for estimating the energy consumption of the hardware/-software component. Examples of model-based energy profilers can be found in Chapter 10. More importantly, power models provide hints on improving the energy efficiency of smartphones and applications. For example, the power models of Wi-Fi network interfaces presented in Section 9.2 describe the impact of traffic burstiness on the Wi-Fi transmission cost. According to these power models, it is more energy efficient to send data in big chunks instead of small bursts. This provides the theoretical basis for the traffic-aware adaptation of streaming applications that shape the traffic into bursts to reduce the transmission cost (see Section 15.1). More examples are given in Part III.

9.1 Methodology

A hardware component can work in several power states, corresponding to different levels of power consumption. The power state a hardware component should be in any particular time depends on the workload generated by the software running on it. As

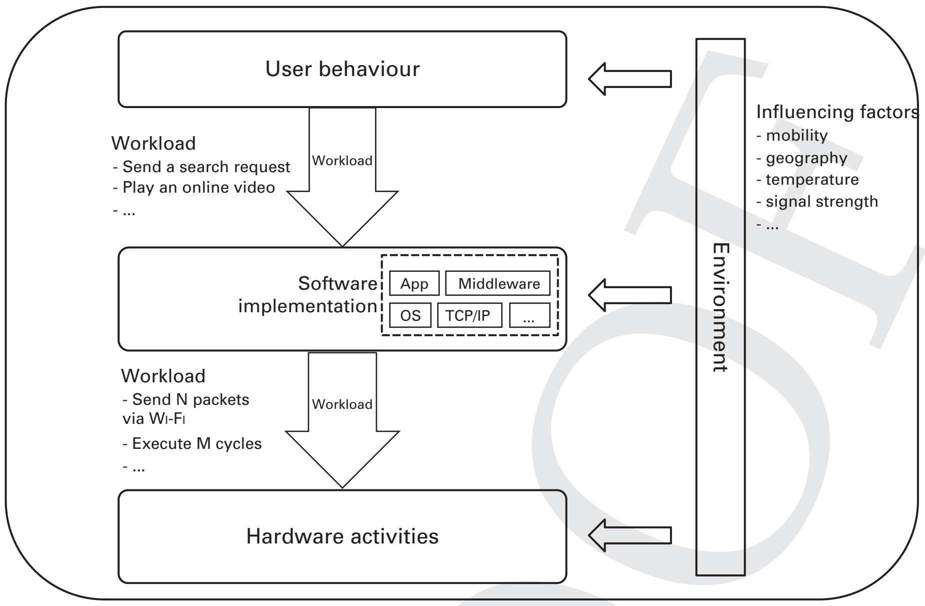


Figure 9.1 Overview of hardware activity causes

shown in Figure 9.1, software including the OS and applications generate the workload of computing, I/O access, encoding/decoding, and other hardware operations. The work is transformed into a set of circuit activities on the corresponding hardware components, and the circuit activities consume energy. Software running on the same smartphone by default share the underlying hardware resources. For example, multiple applications that require internet connectivity can run simultaneously on a smartphone and access the Internet through the same WNI. Therefore, the power state of a hardware component should be determined by the aggregated workload.

From a software perspective, each hardware component has several operating modes. For example, a Wi-Fi network interface has at least three operating modes, corresponding to the operations of sending, receiving, and waiting for traffic. I n most cases where one operating mode corresponds to exactly one power state, the power consumption of the hardware component can be derived from the operating mode, and vice versa. There are also exceptions where the hardware components automatically adapt their operation to their current workload, and thus something the software sees as one single operating mode can in fact include several hardware power states. For example, Pathak et al. [1] observed that on HTC Tytn2 running Windows Mobile 6, the Wi-Fi network interface can switch to a power state with a higher power consumption when the packet rate exceeds 50 packets per second.

In this section we introduce two methods of power modeling: deterministic and statistical. The basic idea of deterministic power modeling is to map software operations to hardware activities based on expert knowledge and to estimate the power consumed by the hardware components involved based on their activities. Statistical power modeling aims to find out the relationship between power consumption and the model variables based on statistical models like linear regression. The variables of statistical power models can be application-specific parameters, hardware performance metrics, and other variables that are related to power consumption.

9.1.1 Deterministic power modeling based on operating mode

Deterministic power modeling can be used for studying the energy consumption of hardware components, such as WNIs and displays. Each deterministic power model is based on a power state machine which describes the transition between power states of the hardware component in question. The total energy cost of a hardware component over time is composed of the energy that the component spends in each of its power states and of the energy spent during the transitions between the power states.

A deterministic power model can be formally presented as follows.

$$E(t) = \sum_j E_j(t_j) + \sum_j \sum_k E_{j,k} \times C_{j,k}(t), \tag{9.1}$$

where $E(t)$ is the total energy consumed by the hardware component over the duration t, $t = \sum_j t_j$, t_j is the duration spent in power state j, and $E_j(t_j)$ is the energy spent during t_j. Assuming that P_j, the rate of energy consumption in power state j, is constant during t_j, $E_j(t_j)$ can be calculated as the product of t_j and P_j. When the assumption fails, $\mathrm{E}_j(t_j)$ can be a function of the power consumption with the operating mode, workload description, and duration as variables.

$E_{j,k}$ is the overhead caused by the transition from power state j to k, while $C_{j,k}(t)$ shows how many times this transition has occurred during t. $E_{j,k}$ depends on the physical characteristics and is usually assumed to be constant. The energy wasted in waiting for the transition into a lower power state is often called tail energy. Sometimes, the transition overhead is not counted in the energy consumption, because the transition overhead is small enough to be safely ignored, or the monitoring of the transition is not feasible.

Building a deterministic power model includes three steps:

1. Defining the operating modes of the hardware component.
2. Discovering the potential power state transition within each operating mode. This type of state transition is usually related to the level of hardware usage. The state transition may be triggered immediately when the usage crosses certain thresholds, or after a predefined timer expires. To discover the potential thresholds and timers, a feasible approach is to define the experiments in the way that the hardware usage increases/decreases at various rates.
3. Measuring the power consumption of the hardware component in each power state and the transition overhead if applicable.

The operating modes of hardware components can be tracked using three methods, depending on how much information is available.

First is to directly read the information about the operating mode from the hardware component via a device driver of the OS. For instance, Quanto [2], a network-wide energy profiler for embedded network devices, adopts this method. However, as standard device drivers do not usually expose the operating mode information, Quanto requires modifications to device drivers.

Second is to estimate the operating mode based on system call traces, as proposed by Pathak et al. [1]. System call traces can show which components are being requested as well as the level of use being requested, as applications always access hardware (I/O) components via system calls. By tracking the calling subroutine of each system call, the hardware access can be related to the calling subroutine, which makes it possible to estimate the energy consumption on a per-subroutine basis. An example of fine-grained energy accounting using system call traces can be found in Section 9.4.

Third is to derive the operating mode from the measured workload. For example, the workload of network transmission can be described with libpcap[1] packet traces. These traces can tell if the wireless network interface is sending, receiving, or waiting for packets. Moreover, they can provide traffic statistics, such as throughput and packet rate, which are useful for detecting workload-driven power state transitions. In practice, a power state machine of the WNI can be built by empirically correlating changes in the packet traces to physically measured changes in power levels. With the help of such a state machine, t_j and $C_{j,k}(t)$ could be derived from a libpcap packet trace. In Section 9.2, we show an example of deriving the operating mode of a Wi-Fi network interface from different kinds of traffic statistics.

The accuracy of a deterministic power model depends on not only the accuracy of the power measurement involved, but also how influential the undiscovered or uncontrollable factors are in estimating the power consumption. For example, if certain sub-states within an operating mode are not discovered, the power model cannot give an accurate power estimation when the transition between these sub-states occurs. In addition, environmental factors such as temperature may affect the energy efficiency of hardware components. These factors are not always controllable in the practice of power modeling.

9.1.2 Deterministic function-level power modeling for software component

Hardware function-level power estimation was earlier used for predicting the power dissipation of the microprocessor [3, 4] at the design stage. This method assumes that the energy required to execute a functional unit is approximately constant, and calculates the total energy consumption as the aggregate of the per-functional-unit cost. As this assumption could also hold for some software functional units, this method has been extended to the power modeling of mobile OSs and applications.

[1] libpcap is a portable C/C++ library for network traffic capture. It is available on www.tcpdump.org accessed March 3, 2014.

Mobile OSs and applications can be decomposed into functional units. Building a function-level power model of a piece of software includes three steps.

1. Measuring the amount of energy (in joules) consumed by each functional unit.
2. Profiling the execution of the target software and counting the functional units invoked in the execution.
3. Calculating the total energy consumption as the aggregation of the energy cost of each functional unit.

In practice the functional unit can be defined with various granularities, depending on the software structure. For instance, Feeney et al. [5] proposed a collection of linear equations for calculating the energy consumption of the Wi-Fi network interface in adhoc mode. Each linear equation corresponds to a software activity, such as sending a point-to-point data packet. A similar method has been applied for analyzing the processing overhead of protocols such as TCP [6] and Secure Sockets Layer (SSL) protocol [7].

The study in [8] identified the energy components of an embedded OS by studying its internal operations and classified them into system functions. It proposed to obtain the base energy of each system function from the power measurement, and to calculate the energy consumption of an embedded OS based on the base energy per system function. Similarly, Li et al. [9] profiled the execution of the mobile OS as a set of kernel service routines, and calculated its energy consumption based on the energy per kernel service routine.

In mobile cloud offloading (see Chapter 14), function-level power models have been used for estimating the computational cost of offloadable methods. Whether to offload the methods at runtime is determined by the tradeoff between the estimated reduction in computational cost and the increase in wireless transmission cost.

9.1.3 Statistical power modeling

Statistical power modeling employs statistical methods, such as linear regression, to estimate the relationship between the power consumption and some measured variable, such as transmission rate or processor clock speed. These methods have been applied in analyzing the power consumption of software components as implemented in PowerScope [10], as well as in modeling the system-level power consumption of the smartphone hardware. Examples of the latter include PowerTutor [11], Sesame [12], and the work presented in [13].

Taking linear regression (refer to Section 2.6.2) as an example, based on the linear dependency of the output on the values of the predictor variables, the values of the coefficients in linear regression models can reflect which variables have relatively more effect on the power consumption. The bigger the coefficient value is compared to others, the more effect the corresponding variable has on the power consumption.

A linear regression based power model can be built in five steps:

1. Select the regression variables. For hardware components, the regression variables can be the metrics that reflect the activity levels of the hardware resources, such as the CPU cycle rate and network data rate.
2. Design the energy benchmark that stresses each regression variable and explores their cross product. The benchmarks generate workloads in ways that correspond to different activity levels of the hardware resources.
3. Run the energy benchmark to collect a set of observations. Each observation consists of the values of each regression variable and the corresponding result of power measurement. These observations are used for model fitting (Step 4) and validation (Step 5). To test the predictive efficacy of the model, one of the core principles is that the data used for building the model should be separated from the one used for final validation. In practice, we can select, for example, two-thirds of the collected observations by random into a training data set. The remaining observations are collected into a testing data set.
4. Use the training data set to form a linear regression model based on the least square method [14]. The model can be used for runtime power estimation.
5. Validate the power model with the testing data set. The prediction accuracy is evaluated by the prediction percentage error and the minimum square error.

More detailed descriptions of each step are given in the case study presented in Section 9.3. In practice, the above method can be applied for both hardware components and software components. For example, the power consumption of microprocessors was modeled using the least square regression method, with parameters defined from hardware performance counters [15], while that of an H.323 video encoder was modeled as a function of the bit rate [16].

9.2 Deterministic power models of Wi-Fi network interface

On a mobile device, the energy consumption of wireless data transmission is mainly caused by the operations on WNIs. These operations, such as sending/receiving a unit of data through a WNI, are controlled by the software running on top of the WNIs, including the hardware drivers, the network protocol stack, and the network applications. Instead of going through the source code of the software in use, an alternative way of tracking the operations on WNIs is to monitor the traffic going through them. Packet-level traffic traces can be collected using packet analyzers, like tcpdump[2] and the open-source Application Resource Optimizer (ARO) from AT&T Labs[3], which have been ported to many mobile OSs, including Android and iOS. These packet analyzers can capture packets and analyze the packet size, packet arrival time, source/destination IP address, and other information indicated in the header of each packet.

We use the Wi-Fi interface as an example to explain how to obtain the values of the parameters listed in Eq. (9.1) from packet-level traffic traces. We assume that the Wi-Fi

[2] `http://www.tcpdump.org` accessed January 6, 2014.
[3] `https://github.com/attdevsupport/ARO` accessed January 6, 2014.

interface adopts the adaptive PSM (described in Section 7.3.1) and its state machine can be illustrated as shown in Figure 7.8. The modeling process consists of two steps.

First, detect the transmit and receive modes based on the transmission direction of the packets. The transmission direction can be worked out by comparing the source/destination IP address with the IP address assigned to the smartphone. After that, the time spent in the transmit and receive modes is determined by the packet sizes and the processing capacity of the Wi-Fi interface in the corresponding operating mode. The processing capacity indicates how fast the Wi-Fi interface can handle packets locally. It is different from the end-to-end network throughput. Given an operating mode, either transmit or receive, its processing capacity can be assumed to be fixed. If the transmit and/or receive mode includes sub-states, each of which corresponds to a certain processing capacity, each sub-state is treated as an individual operating mode.

Second, detect idle and sleep modes based on packet intervals and the PSM timeout, and counting the transitions that occur during the data transmission. Only if the packet interval is bigger than the PSM timeout, does the transition from idle to sleep mode occur and the packet interval is divided into two parts. The one equal to the PSM timeout is spent in the idle mode, while the rest of the interval is spent in the sleep mode. Depending on whether the network interface stays in idle or sleep mode at the end of each packet interval, the transition from idle/sleep mode to transmit/receive mode can also be derived.

The above method can also be applied for the power analysis of the 3G WCDMA network interface [17], whose power state is defined by the RRC protocol [18]. Similar to the adaptive PSM for Wi-Fi, the operating mode when sending/receiving data can be detected by comparing the traffic volume with the ones that can be handled by CELL_PCH and CELL_FACH. In addition, T1, T2, and T3 are the inactivity timers used in CELL_DCH, CELL_FACH, and CELL_PCH, respectively. Each inactivity timer can last for seconds. They work in the same manner as the PSM timeout in adaptive PSM. When the values of these inactivity timers are known, comparing the traffic interval with the timer values, we can figure out the duration spent in each operating mode.

Packets captured over a certain duration may belong to several applications and network flows, as smartphones can run multiple network applications at the same time, and each application may communicate using several network flows. Analyzing network traffic using the concepts of flows and applications is intuitive, and many kinds of traffic statistics at the flow and application levels can be derived from the packet-level information. Power models based on the flow- and application-level traffic statistics can bring insight to mobile application developers on how to improve the energy efficiency of their applications. This is because the traffic statistics at the flow and application levels are more related to the parameters of the mobile applications and network protocols. On the other hand, it is not always feasible for packet-level traffic profiling to be run on mobile platforms, because it usually requires root access and causes more overhead than the traffic profiling with coarser granularity. Hence, it is worth developing power models based on the statistics at the flow or application level, although their accuracy might be lower than that of the packet-level power models. In the rest of this section, we showcase deterministic power modeling of the Wi-Fi interface based on traffic burstiness.

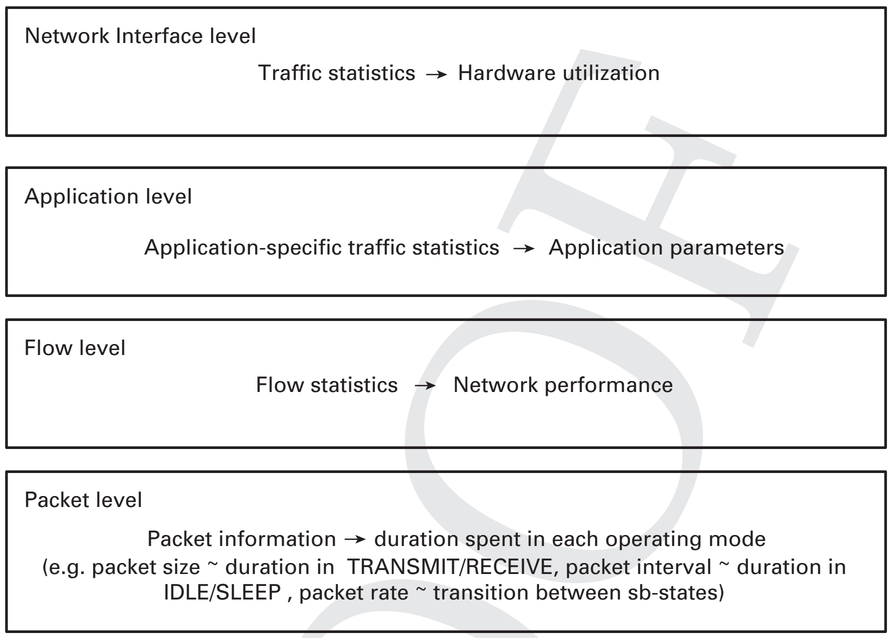

Network Interface level
Traffic statistics → Hardware utilization
Application level
Application-specific traffic statistics → Application parameters
Flow level
Flow statistics → Network performance
Packet level
Packet information → duration spent in each operating mode
(e.g. packet size ~ duration in TRANSMIT/RECEIVE, packet interval ~ duration in IDLE/SLEEP , packet rate ~ transition between sb-states)

Figure 9.2 Workload description of data transmission

9.2.1 Definition of train burstiness

An internet flow can be considered as a train of packets. According to the definition of “train burstiness” in [19], “a burst can be defined as a train of packets with a packet interval less than a threshold θ”. An internet flow can then be divided into bins with one burst in each bin. One burst includes one or more packets, depending on the distribution of packet intervals and the value of θ.

Due to the difference in power between the transmit and receive modes, we add one constraint to the definition of “train burstiness” [19]. We define a burst as a train of packets with the same transmission direction and with each packet interval smaller than the threshold θ. As shown in Figure 9.3, burst duration, T_B, is “the time elapsed between the first and the last packets of a burst” [19], while burst size, S_B, is the amount of data sent or received during T_B. Burst interval, T_I, is the time elapsed between the last packet of a burst and the first packet of the following burst. Bin duration, T, includes the burst duration and the burst interval.

Given an internet flow, we can detect all the bursts and then use the burst information to calculate the average network throughput, $\overline{r}$, over the internet flow using

$$\overline{r} = \frac{\sum S_B}{\sum T} = \frac{\sum S_B}{\sum T_B + \sum T_I}. \tag{9.2}$$

From Eq. (9.2), we can see that given a fixed amount of data and a fixed data rate limit, the data can be delivered in different traffic patterns in terms of distributions of burst

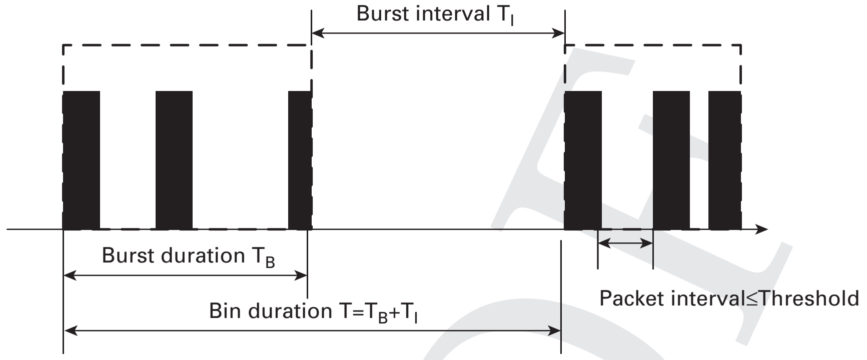


Figure 9.3 Burst definition

size and interval. We use the standard deviation of burst interval and that of burst size to describe the regularity of the bursts. If internet flows consist of bursts with small standard deviations, such as those generated by audio streaming, we consider these flows to be regularly bursty traffic and to be randomly bursty traffic otherwise. We describe the power models that fit these two kinds of traffic in Section 9.2.2.

9.2.2 Downlink/uplink power consumption

According to our definition of train burstiness, a downlink or an uplink flow can be divided into bins. We aggregate the energy spent in each bin into the transmission cost of a flow.

We assume that the threshold value, θ, is always smaller than the PSM timeout, $T_{timeout}$, when the PSM is enabled. This means that the transition from the idle to the sleep mode may only happen during burst intervals. Let T_{sleep} be the duration spent in the sleep mode during a burst interval. As described in Eq. (9.3), only when the value of T_I is greater than that of $T_{timeout}$ can the Wi-Fi network interface switch to the sleep mode. Let r be the bin data rate. In Eq. (9.4) we define a threshold r_c as the bin data rate when T_I is equal to $T_{timeout}$.

$$T_{sleep} = T_I - T_{timeout}, \quad \text{when } T_I > T_{timeout}. \tag{9.3}$$

$$r_c = \frac{S_B}{T_B + T_{timeout}}. \tag{9.4}$$

To evaluate the effect of the PSM, we define the following two scenarios. The Wi-Fi network interface is expected to always stay in the idle mode during burst intervals in Scenario 1. Thus only in Scenario 2 can the PSM save energy.

- Scenario 1: PSM is disabled, or r is not smaller than r_c with PSM enabled.
- Scenario 2: r is smaller than r_c with PSM enabled.

Energy per bin

We denote by P_T, P_R, P_I, and P_S the power when the Wi-Fi network interface stays in the transmit, receive, idle, and sleep modes, respectively. As some modern smartphones may support transmit power control, we make the simplifying assumption that the transmit power stays the same within one burst and only can change between the bursts. We estimate the power within one burst to be fixed to either P_T or P_R, depending on the transmission direction. Our estimation ignores the transition into the idle mode during the packet intervals smaller than the threshold value, θ.

Downlink power consumption is the power consumed when receiving data. Let E_B denote the transmission cost of a bin in joules, and $P_d(r)$ denote the average downlink power in watts. In Scenario 1, the Wi-Fi network interface operates in the receive mode when receiving data and in the idle mode otherwise. Thus E_B includes the energy spent in the receive and idle modes. In Scenario 1, the value of $P_d(r)$ can increase linearly with the bin data rate, r, as shown in Eq. (9.5).

$$P_d(r) = \frac{E_B}{T} = \frac{P_R T_B + P_I T_I}{\frac{S_B}{r}} = P_I + r\frac{T_B}{S_B}(P_R - P_I). \tag{9.5}$$

In Scenario 2, T_I is divided into two parts, $T_{timeout}$ and T_{sleep}. The Wi-Fi network interface can be in the idle mode for a duration of $T_{timeout}$ after receiving the last packet of data, and in the sleep mode after this until the end of the bin. E_B can then be divided into three parts as shown in Eq. (9.6). Accordingly, the definition of $P_d(r)$ is refined into Eq. (9.7).

$$E_B = P_R T_B + P_I T_{timeout} + P_S T_{sleep}. \tag{9.6}$$

$$P_d(r) = P_s + r\left[\frac{T_B}{S_B}(P_R - P_S) + \frac{T_{timeout}}{S_B}(P_I - P_S)\right]. \tag{9.7}$$

Power over an internet flow

If the bursts included in an internet flow are regularly repeated, S_B and T_B can be considered to be fixed, while the length of the burst interval, T_I, varies with the bin rate, r, for example, T_I increases when r decreases.[4] In that case, the internet flow can be compared to one single bin that repeats itself over and over again for the whole duration of the flow. Thus Eqs (9.5) and (9.7) can be used for estimating the average power over the internet flow by replacing r with the $\overline{r}$ defined in Eq. (9.2).

According to Eqs (9.5) and (9.7), power increases linearly with the data rate for regularly bursty traffic. We denote the energy utility of the internet flow by $E_0(\overline{r})$ and define it in Eq. (9.8). Similarly with power, we can see that $E_0(\overline{r})$ increases with $\overline{r}$, which means

[4] Keeping the burst size and burst duration constant and varying the length of the burst interval according to the desired network throughput is a data-rate-limiting mechanism used in many traffic-shaping utilities, such as Trickle [20]

it is more energy efficient to transfer regularly bursty traffic at a higher rate.

$$E_0(\overline{r}) = \frac{\overline{r}}{P_d(\overline{r})}. \tag{9.8}$$

If the bursts included in an internet flow are not regularly repeated, which means the burst sizes and intervals vary over time, the total energy consumption, E, can be aggregated from the energy spent in each bin. When the PSM is disabled, E and $P_d(\overline{r})$ can be calculated using Eqs (9.9) and (9.10).

$$E = \sum E_B = \sum T_B P_R + \sum T_I P_I. \tag{9.9}$$

$$P_d(\overline{r}) = \frac{E}{\sum T} = P_R - \frac{\sum T_I}{\sum T}(P_R - P_I). \tag{9.10}$$

When the PSM is enabled, let T_S denote the total duration spent in the sleep mode and c be the total number of burst intervals. $P_d(\overline{r})$ in this case can be obtained from Eq. (9.11).

$$P_d(\overline{r}) = P_R - \frac{\sum T_I - T_S}{\sum T}(P_R - P_I) - \frac{T_S}{\sum T}(P_R - P_S), \tag{9.11}$$

where,

$$T_S = \left(\sum_{T_I > T_{timeout}} T_I \right) - cT_{timeout} \left(1 - \sum_{T_I \leq T_{timeout}} T_I \right). \tag{9.12}$$

The above Eqs (9.9)–(9.11) can be applied to estimate the average uplink power, $P_u(\overline{r})$, by replacing $P_d(\overline{r})$ with $P_u(\overline{r})$, and P_R with P_T.

9.2.3 TCP download/upload power consumption

We model TCP transmission as a combination of separate downlink and uplink transmissions. Let r_d be the downlink data rate and r_u be the uplink data rate. Taking TCP download as an example, r_d is the data rate of downloading the files, while r_u is the data rate of sending ACKs.

We first discuss the power consumption of a TCP download. We assume that a downlink burst includes n packets, and is followed by uplink bursts that consist of m ACKs in total.[5] Let the downlink burst size be S_{db} and the size of one ACK be S_{ack}. The uplink data rate, r_u, can be obtained from Eq. (9.13).

$$r_u = \frac{mS_{ack}r_d}{S_{db}}. \tag{9.13}$$

[5] Depending on the TCP version, there may be one ACK for each received packet or one ACK for multiple received packets. Depending on the intervals between ACKs and the threshold value in the burst definition, the ACKs may be divided into more than one uplink burst.

We extend the definition of a bin here to have a bin including one downlink burst and all the uplink bursts sent before the beginning of the next downlink burst. The bin duration is the duration from the first packet in the downlink burst until the first packet of the next downlink burst. We assume that the downlink and uplink bursts do not overlap.

We denote the downlink burst duration by T_d, and the uplink burst duration by T_u. For TCP download/upload, we redefine the threshold of network throughput, r_c, in Eq. (9.14). Having the data rate smaller than r_c is a necessary condition for the Wi-Fi network interface to go to sleep during a bin. Whether the interface will go to sleep and how many times the interface will switch into the sleep mode within a bin depends on each value of the burst intervals within the bin.

$$r_c = \frac{S_{db}}{T_d + T_u + T_{timeout}}. \tag{9.14}$$

Let the average power during the TCP download be $P(r_d)$. It consists of both downlink and uplink power. In Scenario 1 defined in Section 9.2.2, $P(r_d)$ can be calculated based on Eq. (9.5), as follows.

$$\begin{aligned} P(r_d) &= P_d(r_d) + P_u(r_u) - P_I \\ &= P_I + \frac{r_d T_d}{S_{db}}(P_R - P_I) + \frac{r_u T_u}{mS_{ack}}(P_T - P_I) \\ &= P_I + \frac{r_d}{S_{db}}[T_d(P_R - P_I) + T_u(P_T - P_I)]. \end{aligned} \tag{9.15}$$

In Scenario 2 defined in Section 9.2.2, $P(r_d)$ can be estimated using Eq. (9.16).

$$\begin{aligned} P(r_d) = P_s + \frac{r_d}{S_{db}}[&T_d(P_R - P_S) \\ &+ T_u(P_T - P_S) + \alpha T_{timeout}(P_I - P_S)], \end{aligned} \tag{9.16}$$

where $\alpha T_{timeout}$ is the total duration the interface will spend in the idle mode during all the burst intervals within a bin. The factor, α, is calculated as follows. Assume that there are X burst intervals within the bin, out of which Y intervals are longer than $T_{timeout}$. At the beginning of each of these Y intervals the interface stays in the idle mode for the duration of $T_{timeout}$ before going to sleep. Additionally, the interface stays in the idle mode during the complete duration of the $X-Y$ intervals that are shorter than $T_{timeout}$. The factor α is thus:

$$\alpha = Y + \frac{1}{T_{timeout}} \sum_{i=1}^{X-Y} T_i : T_i < T_{timeout}. \tag{9.17}$$

Similar to the TCP download, the power consumption of the TCP upload can be calculated as presented in Eqs (9.15) and (9.16) by replacing r_d with r_u, and S_{db} with the data size of the uplink data burst. When considering the power consumption of multiple TCP connections, the aggregate network data rate has to be taken into consideration. In

Table 9.1. Different forms of power models

Information required	Parameters	Eq.	Accuracy
Packet size, arrival time, transmission direction	Burst size, duration, interval, transmission direction	(9.15) (9.16)	High
Packet arrival time, transmission direction	Burst duration, transmission direction	(9.19)	High
Throughput	Throughput	(9.20)	Low

practice, we replace the r_d and r_u in Eqs (9.15) and (9.16) with the aggregate data rate in each direction. The extra protocol processing cost of multiple TCP connections can be ignored when compared to the uplink and downlink transmission cost.

9.2.4 Simplified power models

As listed in Table 9.1, the models presented in Section 9.2.3 require information including packet size, arrival time, and transmission direction. In this section we provide two simplified power models that require less information for Scenario 1 defined in Section 9.2.2.

The first model estimates the average power over the internet flow based on the average power in active mode. Here we define active mode as the operating mode of the Wi-Fi network interface when it stays in either the transmit or the receive mode. We denote the average power in active mode by P_{active}. It can be calculated based on the durations of uplink and downlink bursts as shown in Eq. (9.18). The average power over the internet flow can then be transformed from Eq. (9.10) into Eq. (9.19) by replacing P_R with P_{active}. Equation (9.19) can be applied to any traffic pattern and can be applied for both TCP/UDP download and upload.

$$P_{active} = \frac{\sum T_u \times P_T + \sum T_d \times P_R}{\sum T_u + \sum T_d}. \tag{9.18}$$

$$P = P_{active} - \frac{\sum T_I}{\sum T} \times (P_{active} - P_I). \tag{9.19}$$

The second model simplifies the power models by ignoring ACKs. Due to the small sizes of ACKs, receiving/sending an ACK in a modern smartphone usually costs less than 1 ms. The energy cost of sending ACKs is so small compared to the cost of transmitting data packets. Thus the energy cost of ACKs can be dropped from Eqs (9.15) and (9.16) for practical usage if a higher error rate is acceptable. In addition, the packet intervals in each burst are limited by the threshold, θ. If we assume that the packet intervals can be ignored, the data rate of a downlink burst can be considered to be equal to the maximum processing capacity of the downlink traffic of the Wi-Fi network interface. We denote it by r_{max}. When the PSM is disabled, Eq. (9.15) can be simplified to

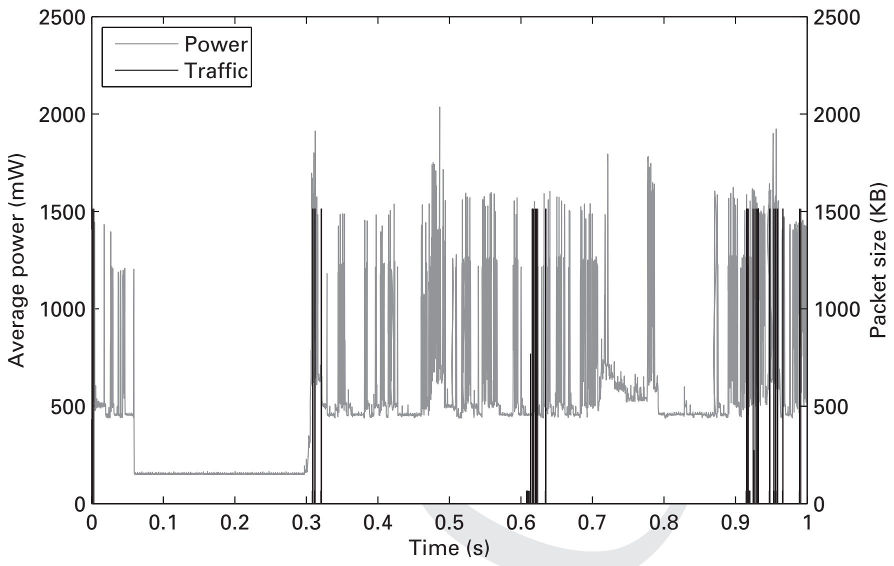


Figure 9.4 Synchronized I/O graph and power consumption on Nexus S

Eq. (9.20).

$$P(r_d) = P_I + \frac{r_d}{r_{max}}(P_R - P_I). \tag{9.20}$$

To calculate the energy consumption of the TCP upload, replace P_R with P_T, r_d with r_u, and r_{max} with the maximum processing capacity of the uplink traffic in Eq. (9.20).

9.2.5 MAC layer retransmission

From the energy perspective, retransmitting a packet is no different to transmitting a "fresh" packet. We ran Wireshark[6] on a Samsung Nexus S while sending packets in a congested network, and then synchronized the traffic trace with the power-measurement trace. As shown in Figure 9.4, the black line represents the I/O graph with each spike corresponding to one IP packet captured by Wireshark running on the phone.[7] The gray line shows the power consumption of the phone during data transmission. We can find a lot more spikes in the gray line, each of which corresponds to a retransmitted packet.

The overhead caused by MAC layer retransmission includes two parts. One part is the energy spent in retransmitting packets on the sender side. According to the retransmission mechanisms used in 802.11 [21], the sender may retransmit a packet several times until the transmission succeeds or until the retransmission limit is reached. Let T_{ir} be

[6] www.wireshark.org accessed March 3, 2014.

[7] The black line does not include MAC layer retransmission, since monitoring of retransmission at the MAC layer requires the Wi-Fi network interface to run in monitor mode, but the interface cannot be used for transmitting or receiving data while operating in monitor mode.

the interval between a retransmitted packet and its previous packet. If the value of T_{ir} is not greater than the threshold value, θ, and the previous packet is an uplink packet, the retransmitted packet can be considered to be part of the uplink burst that the previous packet belonged to. In other words, the uplink burst duration is increased by T_{ir}, due to retransmission. We define $E(T_{ir})$ as the expected value of T_{ir}, and #(*retransmit*) as the total number of retransmitted packets. The cost of retransmitting packets $E_{retransmit}$ can be calculated as follows.

$$E_{retransmit} = \#(retransmit) \times E(T_{ir}) \times (P_T - P_I). \tag{9.21}$$

Given a packet trace captured on the network layer, we denote its packet count by #(*packet*). The value of #(*retransmit*) can be calculated as follows:

$$\#(retransmit) = \#(packet) \times \frac{R_r}{1 - R_r}, \tag{9.22}$$

where R_r is the retransmission ratio calculated from the MAC layer traffic information. For example, if we capture N packets on the MAC layer including M retransmission attempts, the value of R_r is equal to $\frac{M}{N}$.

The other part of the retransmission overhead is caused by the increase in the baseline cost, due to the DVFS mechanism of CPUs. The basic idea of DVFS is to adapt the CPU frequency to the processing workload. The extra workload caused by retransmission may lead to an increase in the CPU frequency, with the result that the baseline cost represented by the values P_T, P_R, P_I, and P_S increases accordingly. For example, in Figure 9.4, the power in the idle mode during 0.1 s and 0.3 s is only 0.177 W, whereas it gets close to 0.5 W during the interval between 0.8 s and 0.85 s. The device backlight was turned off. Meanwhile, the power while sending packets increases by around 0.3 W when the retransmission starts. This change in power is consistent with the change in the CPU frequency from 100 MHz to 200 MHz. Hence, to estimate the transmission cost in congested networks, fine-grained CPU frequency measurement is necessary to provide the right inputs for the power models.

9.2.6 Model evaluation

We built models of Wi-Fi transmission cost for the Samsung Nexus S following the steps described in the above sections. In this section we briefly describe the experimental setup and the results of the model evaluation. More detailed description can be found in [22].

We first obtained the values of P_T, P_R, and P_I from power measurement. Note that the phone we used did not provide any interface for adjusting the PSM parameters. As the measurement results with the default settings seemed to fit the “PSM disabled” version of our power models, we did not provide the value of P_S for this phone. Additionally, we observed that the CPU frequency of the Nexus S varied with the transmission rate. For instance, when the display was turned off, the CPU frequency increased from 100 MHz to 200 MHz whenever the data sending rate of the phone increased from 256 KBps to

Table 9.2. Parameter values for Nexus S

CPU frequency (MHz)	Display	$P_T(W)$	$P_R(W)$	$P_I(W)$	Data rate (KB/s)
100	Off	1.094	0.867	0.177	≤ 128
100	Off	1.130	0.903	0.213	$160 \sim 256$
200	Off	1.245	1.021	0.435	≤ 512
100	On	1.217	0.887	0.742	512
200	On	1.376	1.050	0.800	1024
400	On	1.549	1.208	0.890	≤ 1536

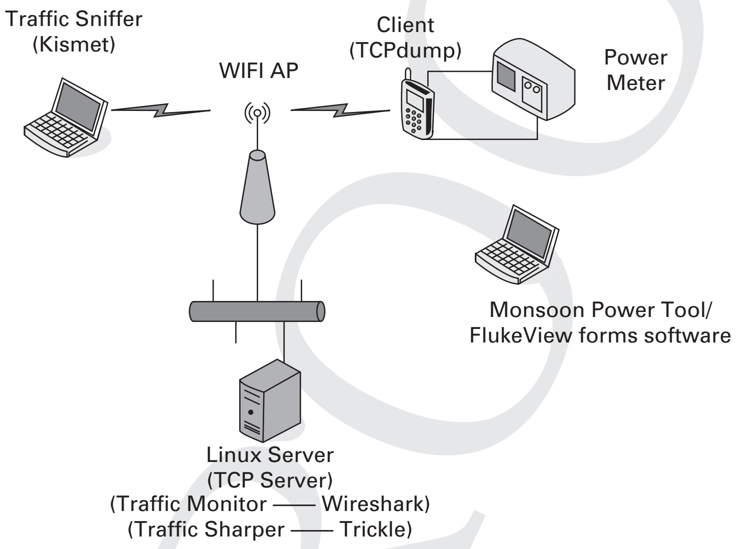


Figure 9.5 Experimental setup

512 KBps.[8]. Table 9.2 lists the parameter values for each CPU frequency. The listed data rate information is to show which values were used in our calculations. They are not necessarily the exact thresholds used in DVFS. In addition, the values of P_T and P_R include the cost of network protocol processing.

We evaluated our models in TCP transmission scenarios at various data rates with various traffic patterns and in different network environments. Our experimental setup is illustrated in Figure 9.5, including the TCP download/upload setup, and power measurement and traffic capturing tools. Our test cases are listed in Table 9.3.

We first conducted the experiments in an ideal network environment where the processing latency and packet loss could be ignored. We applied the model presented in

[8] Due to the partial wake up mechanism in Android, the CPU worked at a reduced frequency when the display was turned off.

Table 9.3. List of test cases

No.	Description	No. flows	Data rate limit
1)	TCP download/upload	1	Enabled/Disabled
2)	Concurrent TCP download/upload	2/4/8	Enabled/Disabled
3)	TCP download/upload in congested network	1	Disabled
4)	Web browser, Fengxing (video streaming), Dropbox (file upload), QQ(Instant messenger)	1	Disabled

Table 9.4. MAPE of power models for Nexus S

No. downlink	No. uplink	MAPE (%)	No. downlink	No. uplink	MAPE(%)
1	0	$3.4 \pm 2.0^*$	0	1	$2.6 \pm 1.5^*$
		4.2 ± 1.7			4.1 ± 4.4
2	0	$3.3 \pm 1.7^*$	0	2	$2.9 \pm 2.0^*$
		2.4 ± 1.1	0	2	5.1 ± 4.0
4	0	2.9 ± 2.0	0	4	5.1 ± 1.5
8	0	2.2 ± 1.4	0	8	5.9 ± 1.8
1	1	$2.1 \pm 1.8^*$	1	1	2.7 ± 1.9
2	2	5.0 ± 2.8	4	4	4.8 ± 2.1

*The display was turned off.

Eq. (9.19) to estimate the power consumption and compared the estimated power with the physical power measurement. As listed in Table 9.4, the average MAPE of the TCP download/upload with a single flow was at most 5.7%.

After that, we conducted TCP download/upload experiments in a congested network environment, with the retransmission ratio varying between 10% and 30%. Due to the interference caused by the neighboring APs, MAC layer retransmissions could not be left ignored. Based on the collected MAC layer traffic traces, we calculated the retransmission ratio, R_r, and the expected value of the retransmitted packet interval, $E(T_{ir})$. The overhead of retransmitting packets was computed following Eq. (9.21). As listed in Table 9.5, in upload cases, taking into account the retransmission overhead can improve the power estimation accuracy by almost 50%.

We also evaluated the accuracy of the complete power models (Eqs (9.15) and (9.16)), and the simplified one (Eq. (9.20)) with four real-life applications running on a Nexus S. The descriptions of the tested applications are listed in Table 9.6 along with experiment parameters and results. The traffic generated by these Android applications had different characteristics, in terms of both traffic size and pattern.

According to our measurements, the phone does not seem to implement the traditional 802.11 PSM/PSM adaptive with the sleep mode, but does instead have a DVFS-induced low-power state that is entered when an inactivity timer expires. This

Table 9.5. MAPE of power estimation in congested networks

Test case	$R_r(\%)$	$E(T_l r)(ms)$	MAPE(%)
Download	11.2 ± 1.5	–	2.4 ± 1.2
Upload	20.6 ± 2.3	0.8 ± 0.2	$5.2 \pm 4(11.0 \pm 1.4^*)$

* The retransmission overhead is not counted.

Table 9.6. Description of the experiments with Android applications

Application	Description	Display	Throughput (KB/s)	Duration (s)	Overhead (mW)*	MAPE (%)
Fengxing video player	Stream videos from youku.com	On	7.9	3054	103	5.4
Dropbox Android app	Upload files to dropbox.com	On	18.9	1578	112	5.3
Web browser	Open web pages on dalong.net	Off	13.9	509	41	4.3
QQ instant messenger	Receive random text messages	Off	0.04	1068	127	8.6

mechanism works in a manner similar enough to the 802.11 PSM that Eq. (9.16) can be applied with good results by replacing P_S with the measured power of the DVFS-induced low-power state, and the value of the PSM timeout with the length of the inactivity timer. We determined empirically the length of the inactivity timer to be about 1.35 s when the display was off, and to be 400 ms when the display was on.

As shown in Figure 9.6, all the models gave reasonable results, except for the QQ messenger where only model Eq. (9.16) was able to estimate the power with good accuracy. This was caused by the exceptionally long burst intervals in the QQ messenger traffic, during which the DVFS put the phone into the low-power state, which was only accounted for in Eq. (9.16).

9.3 Statistical system-level power models of smartphones

We show an example of building a statistical power model for a Nokia N810 following the steps described in Section 9.1.3.

9.3.1 Variable selection

The first step is to define the variables. As we would like to model the power consumption of the processor, the Wi-Fi network interface, and the display, our variables

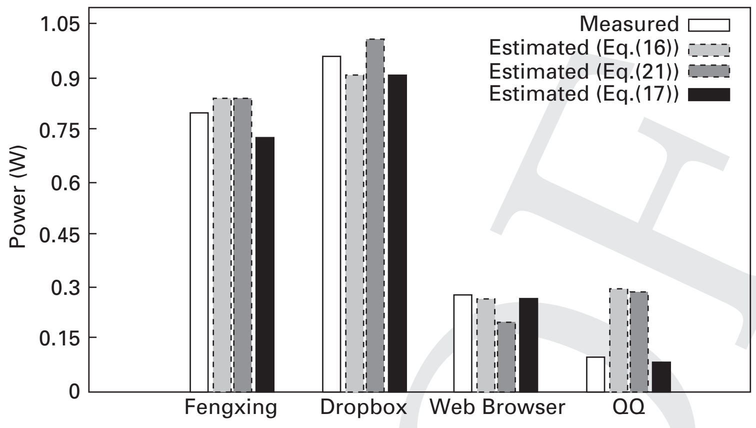


Figure 9.6 Power consumption of 4 Android applications on a Nexus S

are supposed to describe the workload of all three of these components. We initially selected 21 regression variables, which reflected the power characteristics of the processor, the Wi-Fi network interface, and the display. The variables and their preprocessing functions are described in Table 9.7.

We used hardware performance counters (HPCs) to estimate the power consumed by the CPU processing and the memory access, as HPCs reflect the activity levels of the hardware components in a processor such as the DATA cache andinstruction cache. In addition, monitoring some HPCs such as the L3 cache miss counter allows us to track the use of the off-chip memory. In practice, there are 17 HPCs available on our experimental processor, ARM 1136, as listed in Table 9.7, but only the CPU cycle counter CPU_CYCLES and any other two HPCs can be monitored simultaneously during runtime. We defined an event rate for each HPC as shown in Table 9.7. HPCs are aggregate counters. For CPU_CYCLES, we defined the event rate as the consumption rate of CPU cycles, which can be calculated as the number of CPU cycles elapsed in a unit of time. For the other HPCs, we defined the event rate of each HPC as the increment in the HPC during a CPU cycle. It can be calculated as a ratio of the increment in the HPC to the corresponding increment in CPU_CYCLES during the same monitoring period. The event rates are normalized. The statistics used for normalization are calculated based on the data set used for model fitting.

For the cost of the Wi-Fi network interface, we selected three network parameters based on the knowledge gained from a previous study about power modeling of data transmission. The power consumption of the data transmission through Wi-Fi linearly increases with the upload/download data rate. Hence, we selected the upload and download data rates as regression variables because they reflect the workload of the Wi-Fi network interface. Moreover, the 802.11 power-saving mode has an impact on the power consumption. For instance, when the data rate is lower than a threshold such as 32KB/s, the power consumption is higher if the continuously active mode (CAM) is enabled. For the requirement of nonnegative coefficients, we chose the CAM switch as a regression

Table 9.7. Summary of regression variables and their preprocessing functions

Hardware resource	Regression variable $x_{i,j}$	Preprocessing function $g_j(x_{i,j})$	Description
Processor	17 HPC-based event rates: $x_{i,j}(j \in [0..16])$ $= \begin{cases} \frac{c_{i,0} for\ CPU\_CYCLES}{d_i} \\ \frac{c_{i,j}}{c_{i,9}}, for\ other\ HPCs, \end{cases}$ Where is the increment in CPU_CYCLES, and $c_{i,j}(j \in [1..16], i \in [1..n])$ is the increment in any other HPC during the same monitoring period d_i.	Normalization function: $g_j(x_{i,j}) = \frac{x_{i,j} - mean(x_{i,j})}{stdev(x_{i,j})}$ The mean and the standard deviation are calculated from the data set used for model fitting.	HPCs available on ARM 1136: CPU_CYCLES, DCACHE_MISS, TLB_MISS, ITLB_MISS, CYCLES_DATA_STALL, INSN_EXECUTED, DTLB_MISS, DCACHE_ACCESS, DCACHE_MISS, EXP_EXTERNAL, DCACHE_ACCESS_ALL, IFU_IFETCH_MISS, BR_INST_MISS_PRED, CYCLES_IFU_MEM_STALL, LSU_STALL, PC_CHANGE, BR_INST_EXECUTED.
Wi-Fi network interface	Download data rate (KB/s) $x_{i,17}$ Upload data rate (Kb/s) $x_{i,18}$ CAM switch $x_{i,19}(x_{i,19} \in [0..1])$	$g_j(x_{i,j}) = x_{i,j}$	$x_{i,j} = 1$: CAM enabled AND network data rate is lower than a threshold; $x_{i,j} = 0$: Otherwise.
Display	Brightness level $x_{i,20}(x_{i,20} \in [0..5])$	$g_j(x_{i,j}) = x_{i,j}$	Six Brightness levels on a Nokia N810: 0: off; 1..5: brightness from low to high.

variable. When the power-saving mode is disabled, the value of the CAM switch is set to 1. Otherwise, it is set to 0.

We used the brightness level to model the power consumption of the display. Assuming that the resolutions of the displays are fixed during runtime, we divided the workload of the display into six brightness levels, compatible with the screen configuration on a Nokia N810.

9.3.2 Benchmark

The benchmark captures the values of the regression variables at a certain sampling frequency when running the workloads. Each run of the workload lasts for a certain monitoring period, such as 60 seconds.

For example, the workloads we used are categorized into five types: idle with different brightness levels, audio/video players, audio/video recorders, file download/upload at different network data rates, and streaming. In each category there are multiple test cases as described in Table 9.8. They are chosen based on the following three principles.

First, similar to the micro-benchmarks used in [23], our benchmark stresses the selected variables and explores the space of their cross product. The resource consumption of a real-life mobile application can be described in terms of four elements: CPU processing intensity, memory access rate, network data rates, and the brightness level of the display. Each element can be represented by one or multiple regression variables. We define the values of the first three elements to be low, medium, and high, and the value of the fourth element to be equal to the brightness level of the display set in the screen configuration. We chose the test cases in which the resource consumption corresponded to as many different combinations of the values of the four elements as possible, which made it possible for our model to be independent of usage scenarios.

Second, we chose workloads with a fixed demand on hardware resources over a sampling interval of the HPC event rates. Accordingly, the power consumption was considered to be stable during the sampling interval. A monitoring period included at least one sampling interval. For workloads with a fixed demand on the hardware resources over a given monitoring period, the values of the corresponding HPCs were increasing at a constant rate. For example, in a case where a video playback with a fixed frame rate is a workload with a fixed demand on the hardware resources over the monitoring period, the increment in the HPCs such as CPU_CYCLES was stable in each sampling interval. In this case, we calculated the average value of all the samples and considered it to be one observation. For workloads with a varying demand on the hardware resources over the monitoring period, we divided the monitoring period into several smaller periods in each of which the demand can be considered to be stable. For example, an internet radio test case can be divided into two periods: downloading only and downloading together with playback. To simplify the synchronization of the HPCs, the network parameters and the power consumption during data collection, we defined the test cases in the

Table 9.8. Description of the workload used in the benchmark

Category	Description	Test case
Idle with different brightness levels	CPU and memory workload: Low wireless connection: No Brightness level: 0∼5	Keep the system idle without running any applications and set the brightness level of the display to different values.
Audio/Video Players	CPU and memory workload: Low∼High Wireless connection: No Brightness level: 0 for audio player; 5 for video player.	Media player on N810: mplayer Media file storage: Phone memory Audio format: MP3, OGG, RM Number of audio players in parallel: 1, 2, 3 Video format: AVI, MPEG Number of video players in parallel: 1, 2
Audio/Video Recorders	CPU and memory workload: Medium Wireless connection: No Brightness level: 5	Run an embedded audio recorder to record an audio file played on a machine close to the experimental device. Use the embedded camera to record a video.
File Download/Upload at Different Data Rates	CPU and memory workload: Low∼High Wi-Fi connection: On Network data rate (KB/s): 16∼400 Brightness level: 0	N810: netcat Linux Server: netcat, Trickle (bandwidth limiting utility) Data rate limit: 16, 32, 128, 256, and 400KB/s. CAM: On/off (data rate $<$ 32KB/s); Off (data rate $\geq$ 32KB/s) Download storage: phone memory, /dev/dull Upload storage: phone memory
Streaming	CPU and memory workload: High Wi-Fi connection: On Wi-Fi PSM: Enabled Brightness level: 5	Watch online TV programs transferred from www.itv.com. Encoding rate: 16 ∼ 72KB/s Listen to radio programs from three different radio websites. Download date rate: around 24KB/s Use web browser to watch YouTube videos online. Download data rate: 46∼136KB/s depending on the network conditions.

way that the demand of the hardware resources can be stable for a relatively long time.

Third, we chose the applications that are typical on the mobile devices in question. All the applications we used are either embedded in our experimental device, or easy to download and install from the support website for the device.

9.3.3 Data collection

Our benchmark ran Oprofile[9] to access the HPCs from the userspace, and logged the readings of the HPCs every second. The sampling interval of the HPCs was set to 100000 CPU cycles during the initialization of Oprofile. During runtime, three HPCs at most can be accessed simultaneously on an ARM 1136, and one counter is reserved to count the CPU cycles. Similarly with the multiplexing technique presented in [24], to get a full observation including all 17 event rates, our benchmark repeated the same test case eight times with two different HPCs monitored each time. We used the CPU cycles as timers to multiplex the event rates. In other words, the eight samples from the different runs can be merged into one full observation only when the event rate of CPU_CYCLES in each sample is equal to each other.

In practice, we collected 60 samples continuously from the beginning of a test case in each run, except for the streaming cases where we started the monitoring when the playback started. After eight runs, we had 60 full observations. According to our observation, in most of the test cases, the difference among the 60 samples in a line is close to zero. In these cases, instead of importing 60 similar observations into our data sets, we only imported one observation, which included the average value of each HPC in the 60 observations.

We use different test cases to generate two different data sets for model fitting and evaluation, respectively. For example, we used the data collected from three types of workload for model fitting, including idle with different brightness levels, audio/video players, and file download/upload at different network data rates. For model evaluation, we used the data collected from streaming, audio/video recorders, and file download/upload at different network data rates. Even for the same type of workload, the test cases used in model evaluation were different from those in model fitting.

9.3.4 Model fitting

We used a function called lsqnonneg[10] in Matlab, which is meant for optimizing the least-square objectives with a nonnegativity constraint. The nonnegativity constraint reflects the fact that the rate of power consumption increases with the use of hardware resources. Given a linear regression model with nonnegative variable coefficients, the regression variables with relatively large coefficients would contribute more to the output of the model, which is the overall power consumption of the smartphone in our case.

During model fitting, we ran this function twice. In the first round, each observation included seventeen HPC-based variables, three network parameters, and one display parameter. After executing lsqnonneg, we chose three HPC-based variables with the biggest coefficient values. They were the event rates of DCACHE_WB, TLB_MISS, and CPU_CYCLES. In the second round, we fitted the observations including the three selected event rates, the three network parameters, and the one display parameter to

[9] `http://oprofile.sourceforge.net` accessed January 6, 2014.

[10] `http://www.mathworks.se/help/optim/ug/lsqnonneg.html` accessed January 6, 2014.

a regression model by running lsqnonneg again. The final power model is presented below.

$$\begin{aligned} Power(W) = {} & 0.7655 + 0.2474 \times g_0(x_0) + 0.0815 \times g_1(x_1) \\ & + 0.0606 \times g_2(x_2) + 0.0011 \times g_{17}(x_{17}) \\ & + 0.0015 \times g_{18}(x_{18}) + 0.3822 \times g_{19}(x_{19}) \\ & + 0.125 \times g_{20}(x_{20}) \end{aligned} \tag{9.23}$$

where $g_j(x_j)(j \in [0..2, 17..20])$ is the preprocessing function as described in Table 9.7. Let c_0 , c_1, and c_2 be the increment in CPU_CYCLES, DCACHE_WB, and TLB_MISS in the monitoring period d, respectively.

$$\begin{aligned} g_0(x_0) &= \frac{x_0 - 1316.84}{1349.423},\ x_0 = \frac{c_0}{d}, \\ g_1(x_1) &= \frac{x_1 - 0.000901}{0.00045},\ x_1 = \frac{c_1}{c_0}, \\ g_2(x_2) &= \frac{x_2 - 0.000513}{0.000365},\ x_2 = \frac{c_2}{c_0}, \\ g_{17}(x_{17}) &= x_{17}, x_{17} : \textit{download data rate in KBps}, \\ g_{18}(x_{18}) &= x_{18}, x_{18} : \textit{upload data rate in KBps}, \\ g_{19}(x_{19}) &= x_{19}, x_{19} : \textit{CAM switch}, \\ g_{20}(x_{20}) &= x_{20}, x_{20} : \textit{brightness level}. \end{aligned} \tag{9.24}$$

We define the idle mode of a mobile device as the status when there is no application running. The value of the intercept, 0.7655, is close to the power consumption in the idle mode with the display turned off. Among the three HPC-based variables, the event rate of CPU_CYCLES, which describes the general workload of the processor, takes a large part of the total power consumption. The event rates of TLB_MISS and DCACHE_WB reflect the memory access efficiency in a CPU cycle. The coefficient values of the non-HPC variable show an increase in the power consumption of the Wi-Fi network interface or the display when the corresponding variable increases by one unit. For example, an increase of 1 KBps in the upload data rate costs on average 1.5 mW more power, and the network transmission with CAM-enabled costs on average 0.3822 W more when the network data rate is less than 32 KBps on the experimental device.

9.3.5 Model validation

We used the testing data set to validate the regression model obtained in the previous step. To analyze the prediction accuracy for different workloads, we can use the median error for each category of workload as the metric. For the data set used for model evaluation, the median percentage error in power estimation is 2.62%, and the standard deviation of the error rate is 0.0376.

9.3.6 Discussion

When a new hardware component is installed into the mobile device, there are two ways to update the system-level power model. One way is to add regression variables, which describe the activity levels of the new hardware component, define new test cases to stress the new variables, and fit the new data sets to a regression model. The other way is to directly add an analytical power model of the new hardware component to the existing system-level model. For the latter method, an analytical power modeling of the hardware component must be built and validated beforehand.

Because the HPCs can be monitored for each process, we could also estimate the computational power consumption of each process based on the per-process HPC values. Assume that there are N processes contributing to the HPCs during a monitoring period, d. For the process $i(i \in [0..N-1])$, the increments in DCACHE_WB, TLB_MISS, and CPU_CYCLES are defined as w_i, m_i, and u_i, respectively. The total number of CPU cycles elapsed in d is defined as $c_0, c_0 = \sum_{i=0}^{N-1} u_i$. We reform the pre-processing functions of the HPC-based regression variables defined in Eq. (9.24) as below.

$$\begin{aligned} g_0(x_0) &= \sum_{i=0}^{N-1} \frac{\frac{u_i}{d} - 1316.84}{1349.423}, \\ g_1(x_1) &= \sum_{i=0}^{N-1} \frac{\frac{w_i}{c_0} - 0.000901}{0.00045}, \\ g_2(x_2) &= \sum_{i=0}^{N-1} \frac{\frac{m_i}{c_0} - 0.000513}{0.000365}. \end{aligned} \tag{9.25}$$

It is possible to estimate the computational power consumption of process i as shown in Eq. (9.26). However, because there is no power meter available for measuring the power consumption of each process, we have not been able to validate our power breakdown of the processes.

$$\begin{aligned} \mathit{Computational\ Power}(W) = 0.7655 + 0.2474 \times g_0(x_0) + 0.0815 \times g_1(x_1) \\ + 0.0606 \times g_2(x_2) \end{aligned} \tag{9.26}$$

The power consumption of a Wi-Fi network interface is estimated based on the aggregate data rates in our model. When there are multiple flows sharing the network interface, the traffic intervals can be inside a flow or between flows, and there is no common rule for assigning traffic intervals to each flow. In Section 9.4, we show an example of tracking the per-process transmission cost using system-call-based power models.

9.4 Eprof: fine-grained system call tracing energy profiler

Eprof [25] is a fine-grained energy profiler for applications written for Android and Windows platforms. It profiles the energy consumption of each entity based on system-call-based power models [1]. Here an entity can be a process, a thread, a subroutine, or a system call. The procedure is to first track all the caller–callee invocations in the code, then estimate the energy consumption based on predefined power models, and at the end map the estimated energy consumption to entities based on the call graph. This procedure is implemented with three components: code instrumentation and logging, power modeling and energy accounting, and profile presentation.

9.4.1 Code instrumentation and logging

Applications on Android can be written using Android SDK in Java, or in native C/C++, or in both using Google's Native Development Kit (NDK). Java code runs within the Dalvik VM, while the native C/C++ code runs out of the Dalvik VM. Eprof for Android provides SDK routine, NDK routine, and system-call tracing.

The SDK routine tracing logs routing invocations and the time spent per invocation. It is implemented with a modified version of the Android routine profiling framework.[11] For performance purposes, the modified version only counts all caller–callee invocations, and periodically snapshots the routine call stack.

The NDK routine tracing was done by the gprof port of the NDK profiler.[12] Pathak et al. [1] used SystemTap to log CPU scheduling events in the sched.switch function in the kernel. In addition, they used the ADB (Android Debugger) logging APIs to log different sensors, GPS, accelerometer, and camera accesses at the application framework level, and to log disk/network at the Dalvik VM level.

To enable the above tracing functionalities, customized kernel images are required. In addition, source code is needed for tracing the NDK part of applications. Regarding the overhead of logging, according to the measurement results presented in [25], the logging incurs overheads on the CPU and memory. The energy overhead varies between 0.40% and 7.35% for the application on Android, while the logging rate for the application varies between 60–70 KB/s.

9.4.2 Power modeling and energy accounting

The traces collected during an application run are postprocessed for model-based accounting. Eprof uses both deterministic and statistical power modeling.

First, the power consumption behavior of each hardware component or the whole mobile device is described with a finite state machine (FSM). Each state in the FSM represents a power state of a component, or of a set of all components when extended to

[11] Android routine profiling framework marks routine boundaries with timestamps at runtime and calculates the runtime of each routine. Android debug class:`http://developer.android.com/reference/android/os/Debug.html` accessed January 6, 2014.

[12] `http://code.google.com/p/android-ndk-profiler/` accessed January 6, 2014.

model the overall power consumption of the smartphone. Each power state is annotated with a (power, timeout duration) tuple, and the timing and workload of recent events of the component. The transition between power states can be triggered by a timeout activity, a new system call, or a change in hardware usage. Timeout activities can be observed from a timer-based power-saving mechanism in which the component stays in a power state for a while after finishing the operation and then switches into another power state with lower power consumption. The duration the component stays in the first power state is controlled by a predefined timer.

Second, although the power consumption within a power state is constant for components like GPS and camera, for components like WNIs it varies with the workload on the components. In the latter, usage-based linear regression power model is built for each power state. This is different to the usage-based power model presented in Section 9.3, as the level of use is estimated from the system call trace instead of the OS-level hardware performance counters. There are two reasons why the system call trace is chosen. First, as I/O components are always accessed through system calls, the system call trace can tell which I/O component is requested. Together with the parameters of the system calls, it can clearly indicate the usage of each I/O component. Second, a system call can be related back to the calling subroutine and the hosting thread and process, which makes it possible to account for energy consumption on a per-subroutine, per-thread, or per-process basis.

Pathak et al. developed the CTester application suite, based on the domain knowledge of system calls for each OS, to automate the construction of the FSM. I/O system calls are classified into two categories: 1) initialization-based system calls that start or stop a component, such as file open and close; 2) workload-based system calls which generate workload for the component, such as file read and write. For each component, the test application exercises the relevant system calls by interleaving initialization and workload system calls. The range of input parameters to the consecutive workload-based system calls is decided based on the throughput of the component. The idea is to uncover the threshold on the workload that triggers the transition between sub-states of the power state. After uncovering the FSM for each single component, a wrapper application is invoked to create scenarios of concurrent system calls on multiple components. The wrapper application invokes individual applications at a predetermined timing. While running the application suite, the power consumption is measured using an external power meter. The models using system calls as input are then built following the deterministic and statistical power modeling methodologies.

9.4.3 Profile presentation

Energy consumption estimated from the power models is mapped back to the routines following the call graph collected by the tracing tools (see Section 9.4.1). For tail energy, Eprof applies a last-trigger policy which always includes the tail energy in the last entity out of all the entities that would have triggered the tail. Eprof can provide a call-graph view, which mimics the output of gprof [26] by replacing each time value with a (time, energy) value tuple. The results of energy accounting are visualized using an extended version of Traceview in the Android SDK.

9.4.4 Accounting accuracy

As it is difficult to measure per-entity accounting accuracy, Pathak et al. proposed to define accounting error as the percentage difference of the sum of all entity energies except process 0 (which does not use any hardware component) with the ground truth energy measured. They evaluated the accuracy of Eprof for Android with eight applications: Google on Browser, Facebook, AngryBirds, New York Times app, CNN on Browser, Photo uploading, MapQuest, and Free Chess. Compared with split-time [9] and usage-based power modeling, the error in Eprof at the process level is less than 6% for all the applications, much lower than the errors of the other two. At thread and routine granularities, the accuracy of Eprof is as high as split-time and is higher than that of usage-based power modeling. More detailed results are presented in [25].

Generally speaking, although the more complex logging functionalities may generate more overhead, the system-call-based power modeling used by Eprof provides finer-grained and more accurate power estimation. Users can choose which power-modeling method to use, depending on the actual requirement of accuracy, granularity, and overhead.

9.5 Summary

In this chapter we introduced two power modeling methodologies: deterministic and statistical. In practice, the deterministic methodology can be used for building power models for individual hardware/software components, while the statistical methodology can be applied to usage-based power models for a component or the whole device. In the case where the power consumption within a power state varies with workload, a usage-based power model can be built for each power state. We provide three case studies. The first two case studies use deterministic and statistical methodologies, while the last case study shows how to use a combination of these two methodologies.

References

[1] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, “Fine-grained power modeling for smartphones using system call tracing,” in *Proc. 6th Conf. on Computer Systems*. New York, NY, USA: ACM, 2011, pp. 153–168. [Online]. Available: http://doi.acm.org/ 10.1145/ 1966445.1966460

[2] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, “Quanto: tracking energy in networked embedded systems,” in *Proc. 8th USENIX Conf. on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 323–338. [Online]. Available: http: //portal.acm.org/citation.cfm?id=1855741.1855764

[3] G. Qu, N. Kawabe, K. Usami, and M. Potkonjak, "Function-level power estimation methodology for microprocessors," in *Proc. 37th Annu. Design Automation Conf.* New York, NY, USA: ACM, 2000, pp. 810–813. [Online]. Available: http://doi.acm.org/10.1145/337292.337786

[4] J. Laurent, N. Julien, E. Senn, and E. Martin, "Functional level power analysis: an efficient approach for modeling the power consumption of complex processors," in *Proc. Conf. on Design, Automation and Test in Europe - Volume 1*. Washington, DC, USA: IEEE Computer Society, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=968878.968987

[5] L. M. Feeney and M. Nilsson, "Investigating the energy consumption of a wireless network interface in an ad hoc networking environment," in *Proc. 12th Conf. on Computer Communications*, vol. 3, 2001, pp. 1548–1557.

[6] B. Wang and S. Singh, "Analysis of TCP's computational energy cost for mobile computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 296–297, June 2003. [Online]. Available: http://doi.acm.org/10.1145/885651.781065

[7] N. Potlapally, S. Ravi, A. Raghunathan, and N. Jha, "A study of the energy consumption characteristics of cryptographic algorithms and security protocols," *IEEE Trans. on Mobile Computing*, vol. 5, no. 2, pp. 128–143, February 2006.

[8] T. K. Tan, A. Raghunathan, and N. K. Jha, "Energy macromodeling of embedded operating systems," *ACM Trans. Embed. Comput. Syst.*, vol. 4, pp. 231–254, February 2005. [Online]. Available: http://doi.acm.org/10.1145/1053271.1053281

[9] T. Li and L. K. John, "Run-time modeling and estimation of operating system power consumption," in *Proc. 2003 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM, 2003, pp. 160–171. [Online]. Available: http://doi. acm.org/10.1145/781027.781048

[10] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proc. 2nd IEEE Workshop on Mobile Computer Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: http://dl.acm. org/citation.cfm?id=520551.837522

[11] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2010, pp. 105–114.

[12] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2011, pp. 335–348. [Online]. Available: http://doi. acm. org/10.1145/1999995.2000027

[13] Y. Xiao, R. Bhaumik, Z. Yang, M. Siekkinen, P. Savolainen, and A. Yla-Jaaski, "A system-level model for runtime power estimation on mobile devices," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int. Conf. on Cyber, Physical and Social Computing (CPSCom)*, 2010, pp. 27–34.

[14] C. Lawson and R. Hanson, *Solving Least Squares Problems*. Prentice-Hall, 1974.

[15] D. C. Snowdon, S. M. Petters, and G. Heiser, "Accurate on-line prediction of processor and memory energy usage under voltage scaling," in *Proc. 7th ACM & IEEE Int. Conf. on Embedded Software*. New York, NY, USA: ACM, 2007, pp. 84–93. [Online]. Available: http://doi. acm.org/10.1145/1289927.1289945

[16] X. Lu, T. Fernaine, and Y. Wang, "Modelling power consumption of a h.263 video encoder," in *Proc. 2004 Int. Symp. on Circuits and Systems*, vol. 2, May 2004, pp. 77–80.

[17] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, “Profiling resource usage for mobile applications: A cross-layer approach,” in *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, June 2011, pp. 321–334. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000026

[18] 3rd Generation Partnership Project (3GPP), “Radio resource control (rrc) protocol specification,” *3GPP TS 25.331*, 2006.

[19] K.-c. Lan and J. Heidemann, “A measurement study of correlations of internet flow characteristics,” *Comput. Netw.*, vol. 50, no. 1, pp. 46–62, 2006.

[20] M. A. Eriksen, “Trickle Bandwidth Shaper,” 2007, accessed January 12, 2014. [Online]. Available: http://monkey.org/~marius/pages/?page=trickle

[21] “IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks-Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,” *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, June 2007.

[22] Y. Xiao, Y. Cui, P. Savolainen, M. Siekkinen, A. Wang, L. Yang, A. Yla-Jaaski, and S. Tarkoma, “Modeling energy consumption of data transmission over Wi-Fi,” *IEEE Trans. on Mobile Computing*, vol. 99, no. PrePrints, 2013.

[23] C. Isci and M. Martonosi, “Runtime power monitoring in high-end processors: methodology and empirical data,” in *Proc. 36th Annu. IEEE/ACM Int. Symp. on Microarchitecture MICRO-36.* 2003, pp. 93–104.

[24] R. Azimi, M. Stumm, and R. W. Wisniewski, “Online performance analysis by statistical sampling of microprocessor performance counters,” in *Proc. 19th Annu. Int. Conf. on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088163

[25] A. Pathak, Y. C. Hu, and M. Zhang, “Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof,” in *Proc. 7th ACM European Conf. on Computer Systems*. New York, NY, USA: ACM, 2012, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168841

[26] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A call graph execution profiler,” *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982. [Online]. Available: http://doi. acm.org/10.1145/872726.806987

10 Power profilers

The rapid advancement of the communication and computing capabilities of smartphones has led to batteries depleting faster. The Android and Apple's iOS application ecosystem both include many applications that help their users manage the battery life of their devices. Some of these are automated, and give little or no control to the user, turning off functionality that is not used and reducing the amount of time the phone spends awake. Another class of battery-management applications is informational, providing users with options, indicating how much energy each option will save, and keeping track of the energy use of the phone. These applications are typically called energy profilers or mobile battery-awareness applications. The former is targeted developers whereas the latter is for users of the smartphones.

The primary goal of these applications is to make the developer or user aware of what consumes energy. They give the user insight into the factors that consume battery power on their mobile device and give advice on how to deal with them.

In this chapter, we consider the state of the art in energy and power modeling on smartphones. Following the terminology presented in Chapter 6, an energy profiler is a system that characterizes the energy consumption of a smartphone. Typically, the profiler relies on power models that represent power draw. An energy profiler can be a separate device or a software component running on the smartphone and using the device's battery interface for energy and power information. The energy profiler may simply use a power model that has been generated in a laboratory environment or it can build and maintain its own power model in order to adapt to a specific device and usage patterns. The profiler can also be used to generate new power models.

Profilers can be categorized based on their operating environment. They can be used in a laboratory setting in so called offline mode, they can support online on-device operation, or involve an offline calibration and online measurement parts. Any profiler that relies on on-device measurements is limited by the battery API offered on the device. We examined the battery APIs of today's smartphones for the implementation of online profilers in Chapter 4. In this chapter, we give a survey of well-known energy profilers and compare their features.

10.1 Overview

In the general power-modeling process outlined in Chapter 6, a power-measurement technique is used to build a model of a smartphone or its component with certain use cases. The power model can be correlated with execution traces, process lists, and other information to have more detail on the causes of energy consumption. The power measurement can be fully offline taking advantage of high-precision measurement devices and a server to correlate the energy data with the smartphone derived data. There are also alternatives to this laboratory setting, in which the profiler runs on the smartphone.

Ideally, an energy profiler would be:

- an online on-device that does not require external measurement devices or calibration,
- adaptive and able to cope with heterogeneous devices,
- having a high accuracy at a high sampling rate.

These requirements are challenging to meet, because it is difficult to determine the power draw of the smartphone and its components due to limitations of the battery interface and inaccuracies with battery modeling. We discussed the limitations of smart battery interfaces in Chapter 3. The online profiler also introduces bias, because it is running on the same device that it is measuring.

An energy profiler is a software or hardware component that monitors and characterizes the energy consumption of a device. The energy consumption estimation is based on power models. A power model can describe a single hardware subsystem or a combination of them. The power model used by a profiler can be generated offline in a laboratory setting, it can be updated and improved by the profiler, or it can be created online running on the target device. If the power model requires power estimation, an external measurement interface or a software API is needed for the energy and power information. The profiler is limited by the accuracy of the power estimation interface. Profilers can be categorized based on their operating environment: offline, online, and hybrid solutions. An energy profiler can work at different levels of granularity depending on the power model used.

The following categories give insight into the various implementation and deployment strategies for profilers:

- Offline or online in a laboratory setting with an external monitoring device. Offline energy profiling typically supports fine-grained and accurate characterization of the energy consumption of the target device. The difference between offline and online in this case is that with the latter the measurements are used immediately whereas in the former case the analysis can be performed later.
- Online and on-device with offline calibration. In this case, the energy profiler runs on the smartphone and uses a power model that was typically created in a laboratory setting together with online information to characterize energy consumption.
- Fully online and on-device with self-calibration (self-constructive). In this case, the energy profiler does not need a power model that has been created beforehand, but instead creates it online using local information obtained through the smart battery interface and the OS.

- Energy diagnosis engines that aim to identify harmful energy consumption anomalies and then mitigate these. A diagnosis engine can be on-device or it can be based on a centralized server for performing the analysis. Typically the the analysis is performed offline or when the smartphone is being charged to avoid performance and battery problems.

The online information that is used together with the power model typically include SOC and voltage and current from the smart battery interface, as well as details of how hardware components are used. The on-device profiler typically uses the smart battery interface in combination with a battery model, for example the Rint or Thevenin models discussed in Chapter 3; however, not all profilers have a model for the battery.

The level of detail depends on the abstraction level of the energy profiler. The levels of abstraction include the device, component, and process level. Component and process level operations require more information about the system state. The former requires the ability to map energy consumption to hardware components and the latter builds on this and maps processes to hardware components.

The energy profilers can be categorized based on the underlying energy accounting technique. Frequently used accounting techniques are time based, usage based, and FSMs.

- The time-based technique divides the time into slots and the power draw during a slot is mapped to the active entities at that time.
- The usage-based technique keeps track of each system component usage and applies a power model to obtain per-component power estimates.
- FSMs, on the other hand, build a more detailed view of how the system is used by tracking, for example, system calls.

The usage-based models are typically created with linear regression techniques. The regression-based models can be created in the lab with high-accuracy measurements of microbenchmarks or they can be built as online on-devices based on measurements based on the smart battery interface. Figure 4.1 gives an overview of the information provided by the smart battery interface. Typically, the on-device profilers use the SOC/SOD and voltage and current information provided by the battery interface. The current reading is typically preferred; however, it is not available on most smart battery interfaces. SOC/SOD can be used to construct an on-device power model; however, this reading is typically inaccurate and it has a long model generation time due to the slow update rate. The voltage reading can be used to infer current; however, either a discharge curve is needed or the instant voltage dynamics need to be used.

10.2 Survey of energy profilers

Various energy profilers have been proposed to address the problem of accurate power modeling and estimation. Table 10.1 lists well-known profilers in chronological order.

Table 10.1. A list of energy profilers

Name/Authors	Year	Purpose
PowerScope	1999	Energy profiling of device and processes
Joule Watcher	2000	Fine-grained thread-level profiling
Nokia Energy Profiler	2006–2007	On-device standalone profiler
Shye et al.	2009	Energy profiling of device and components with a logger application
PowerTutor	2009	Hybrid profiler based on PowerBooter
PowerBooter	2009–2010	Short-term power model for components
BattOr	2011	Portable power monitor
Sesame	2011	Self-constructive on-device power model for device and components
PowerProf	2011	Self-constructive API-level power profiler
MobiBug	2011	Automatic diagnosis of application crashes
Carat	2012–2013	Application energy profiling and debugging
eProf	2012	Fine-grained power model for device, components and applications
DevScope	2012	Self-constructive power model for device and components
AppScope	2012	Fine-grained energy profiler for applications based on DevScope
eDoctor	2012	Automatic diagnosis of battery drain problems
V-Edge	2013	Self-constructive power model for device and components

In addition to energy profilers, various power consumption observations and studies have been made [1].

The PowerScope is an early example of an offline profiler that can correlate power draw with processes [2]. Joule Watcher is another example of a profiler that has an offline calibration phase. Joule Watcher can provide fine-grained thread-level profiling [3]. The Nokia Energy Profiler (NEP) is an early example of a fully online profiler that uses the Symbian Series 60 device battery API to determine the power draw [4]. The NEP application is instrumented by Nokia to minimize measurement errors. NEP has been designed as a tool for software developers who want to create energy-efficient mobile applications.

Shye et al. presented a profiler based on a logger application and offline instrumentation [5]. PowerBooter is a short-term model for components and PowerTutor is a profiler based on PowerBooter [6]. PowerBooter was the first profiler to use an automatic power model creation by using on-device voltage sensors and battery discharge curves. Sesame

[7], DevScope [8], and V-Edge [9] are online self-constructive profilers that aim to automatically build and update smartphone and component power models. AppScope [10] is a kernel-based energy profiler for Android that is based on DevScope. PowerProf [11] is an unsupervised API-level energy profiler that uses genetic algorithms.

The battOr profiler is an example of a hardware device for portable power measurement [12]. BattOr is attached to a smartphone and it uses the phone's battery or a USB for power. The measurement is accomplished by attaching the battOr profiler to the phone's power connections with the battery inside the phone.

MobiBug [13], Eprof [14], and Carat [15] aim to diagnose mobile applications for energy anomalies. Of these Eprof is an example of an offline profiler that generates a FSM model.

An energy profiler can work on multiple levels of abstraction that include device, component, process, and application levels. For example, PowerBooter provides a short-term model for components and PowerTutor uses the model in combination with usage statistics to generate application-specific energy consumption reports and address the questions of how applications use hardware components. In a similar manner, DevScope generates device- and component-level models and AppScope builds on these to estimate application energy consumption.

The online energy profilers typically employ linear models that are less accurate in modeling the energy consumption of the communications subsystem than FSM-based models. Linear models do not capture state changes in hardware components that are necessary for fine-grained energy estimation of tail states and asynchronous system behavior.

The existing energy profilers focus mainly on single CPU systems and they do not model multiple cores, co-processors, or GPUs. As multicore systems have become more popular they are now being introduced into energy profilers as well as GPUs. On-device energy profilers do not typically model the cellular network state in detail. The RILAnalyzer application is an on-device tool for monitoring the RRC states of the 3G modem subsystem on specific Android phones with the Intel/Infineon XGold chipset [16].

In addition to the above examples, the Android operating system has a built-in energy profiler, BatteryStats, that shows statistics about battery use on the device. This can be accessed from the battery option in the settings on most devices. BatteryStats is examined in the next section in more detail.

10.3 Android battery management framework: battery statistics

The Android OS provides built-in energy models and a runtime statistics service called BatteryStats for estimating component-level power usage. In this section, we examine this service in more detail and report best practices from the Android forum [17] for smartphone energy-consumption measurement.

The framework performs basic registering of usage statistics with the BatteryStats service, mainly by tracking the usage of components, such as the CPU, display, cellular wireless, Wi-Fi device, and GPS. All state changes in devices such as on/off, idle/full speed, and bright/dim are reported to the registering service. The usage data is collected with timestamps and stored so that it can persist in the case of reboots. The BatteryStats service does not monitor the physical power draw of the components, but rather it records the timing that is then used to estimate the approximate power consumption of the device. This estimation is based on the average power consumption of the components that is given by component-specific profiles [17].

Furthermore, the system makes the effort to attribute the power consumption back to the level of applications. If several applications share the use of one resource, and thus the total cost of its energy consumption, BatteryStats makes the additional effort to allocate the cost between the participating applications. A wake lock for a component can be set by several applications and every instance of the lock is enough to prevent the device from shifting to the suspended state. Then every application that has set a partial lock shares the cost in power consumption of keeping the device active. Timing of partial locks during their lifetime determines the exact partial costs.

BatteryStats stores the statistics for half an hour and thus prevents losing vital information in a crash situation (which could be caused by battery problems). BatteryStats gets the information from devices in two different ways, both implemented in the framework:

1. The push method is used when services push all relevant state changes inside their component space to the BatteryStats service.
2. The pull method creates a snapshot of the CPU and other components used by applications. Typically this is done periodically or at critical points such as at the starting or stopping of an activity.

BatteryStats can be used by an energy profiler to obtain usage statistics data. For example, it is used by PowerTutor. Usage statistics can also be obtained from the Linux procfs/sysfs or performance counters. AppScope analyzes system call traces and Android IPC to accurately estimate how applications use device hardware components [10].

10.3.1 Power profile values

To be able to calculate the divided power costs over time, BatteryStats needs to have a power profile. This is the table where the current consumption of components in a device are stored and where the manufacturer must provide power consumption values for different activity states. The power consumption in the power profile is measured in milliamps (mA) of drawn current (nominal voltage). Fractional values can be used to denote microamps. The given value of electric current means milliamperes drawn directly from the battery [17].

The display is a typical example of a component in the power profile table. For the display, the target is to approximate how much power it will take to keep the display on for given time. For this the framework gathers two pieces of data: 1. brightness levels, and 2. times spent at each level.

In the power profile, BatteryStats can see the quantity of current in milliamps needed to set the display at minimum brightness or maximum brightness and keep it there. From the incoming data, BatteryStats obtains the time spent in each feasible brightness level and it can then approximate the battery drain by interpolating the actual brightness levels from the minimum/maximum values and then using simple multiplication and addition.

The CPU is another example component in the power profile table. The CPU power profile includes data for the CPU power drain in milliamperes for various clock speeds. An application's CPU time as a foreground process and as a background process can be then multiplied by the power value (needed to have the CPU operate at the speeds required by the application's executable code) to find the battery drainage for the application.

In general the measure for the power consumption of a component can be calculated after estimating two separate values:

1. The power used by the component when it is in the targeted state (whether on, active, or scanning).
2. The power (current) used by the component when it is switched off.

The comparative value for power consumption is then obtained by subtracting the idle power usage from the usage in the target (active) state.

10.3.2 Measurement of the current

In a typical measurement of, for example, the power value for screen on, the device should be kept at the following [17]:

- stable state,
- CPU speed constant,
- airplane mode,
- a partial WakeLock set for blocking the system suspend.

This facilitates the measurement of stable current values. A measurement is taken of the screen current off value. Then the screen is turned to minimum brightness. The increase measured is the screen on power value.

The system's suspend state itself is sometimes a target for power usage measurements. Outside this it is a nuisance because when triggered it causes auxiliary variability in the power consumption when it moves the system into C-states or P-states. Then it must be avoided/prevented and this is accomplished by setting partial WakeLocks from a development host. But sometimes we want to measure the power needed to enter the suspend state in an Android system and then we must induce the suspension knowingly.

The complication with some components, such as cellular wireless or Wi-Fi, is that their current usage waveform is not flat. Then we have to resort to measuring the average current usage over some given time. This requires the monitoring system to be able to compute average consumption.

10.3.3 Measurement details

The Android power profile assumes that all CPUs share the set of available speeds and other powering characteristics. The list of the available CPU speeds for the device should be specified in the cpu.speeds entry of the power profile. Also in Android, the framework can be stopped with sdb shell stop. This reduces the scheduling and helps to keep the background power consumption static [17].

The number of online cores in the system often considerably affects the device's power consumption; to control the situation it might be necessary to make modifications to the cpufreq driver. A satisfactory solution is to employ the userspace cpufreq governor while using sysfs interfaces for speed settings. The specific cpufreq implementation of the device is likely to set different requirements for the exact command structure.

Networking measurements need a network with a planned and known traffic density. Then we can be sure that no unrealistic loads are weighing on the measurement values of power consumption and that we get an approximation of average Wi-Fi usage. Note that network traffic can be simulated in a controlled way using iperf.

There are two values in the system that can be useful for Wi-Fi measurements:

1. wifi.on gives a value for the power consumption of Wi-Fi, when it is enabled but not actively participating in traffic.
2. wifi.scan gives the value for power consumed during a Wi-Fi AP scan. An application can start a scan of this kind using the WifiManager class startScan() API.

In the following we compare and examine the power profiles in more detail. Our categorization of profilers is based on their mode of operation, online or offline, and energy diagnosis capability.

10.4 Offline energy profiling in a laboratory

In this section, we examine offline energy profilers. An offline energy profiler requires an external infrastructure, such as an external power monitor tool. Our key examples in this category are PowerScope, Joule Watcher, the system by Shye et al., and the Eprof profiler. We briefly examine each of the systems and then present a comparison of their features.

10.4.1 PowerScope

PowerScope is an example of an early energy profiler that mapped energy consumption to program structure [2]. The approach involves combining hardware measurement and statistical sampling of system activity through kernel software support. The technique is an example of offline energy profiling, in which the instrumented system is measured by an external tool, and then later the data is post-processed and analyzed.

The PowerScope energy analyzer generates activity-based profiles by integrating the product of the instantaneous current and voltage over time. The value is approximated by sampling the voltage, V_t, and current, I_t, at regular intervals of length, Δt. In the implementation, this is simplified by assuming a constant voltage for the samples. The energy over n samples using voltage measurement, V_m, is given by

$$E \approx V_m \sum_{t=0}^{n} I_t \Delta t. \tag{10.1}$$

This technique requires that the current is sampled at intervals of Δt and that the voltage can be determined at a suitable level to keep the end result accurate. The energy use of applications is estimated by measuring the time that the device spends in each state and then determining the energy spent during that period.

PowerScope uses both online and offline techniques to profile applications. The online part is realized by the profiler running on the target device connected to the external power monitor. The data given by the profiler and the power monitor are then combined at a server to obtain a process-specific energy profile. The profiler samples the program counter and process identifier of code execution. The output of the online profiling stage is a sequence of current samples and a correlated sequence of program counter and process identifier samples. These are then combined in the offline stage with symbol table information obtained from binaries. An example energy profile created with this process would give the CPU time, total energy, and average power for processes and procedures. The scalability of this approach is limited, because it requires that the offline stage is done for each hardware configuration and device.

The Odyssey OS mentioned in Section 8.6 uses PowerScope to understand process energy consumption [18]. This system modifies the system behavior to meet energy objectives by choosing the most suitable fidelity for data transmission and processing.

10.4.2 Joule Watcher

Joule Watcher is an energy profiler that offers fine-grained thread-level profiling of the device [3]. The profiler is based on counters in hardware drivers to register events that indicate subsystem-level energy consumption. This system can measure the energy consumption for a single fine-grained event, such as a disk access or a floating point operation. The Linux kernel context switch routines and data structures were modified to store performance monitoring event counter values. The energy consumption is determined with an external power monitor device and the regression-based power model is built based on microbenchmarks. The power model can then be used in an online manner once the power model has been created; however, this approach is limited, because the power model needs to be trained for each device type and configuration in the lab.

An energy-aware scheduling system was proposed in conjunction with the Joule Watcher profiler. The scheduler in this system evaluates the energy usage of each thread and then throttles the system to meet the desired energy target. The throttling is implemented by a dedicated throttling thread.

10.4.3 Linear regression with a logging application

Shye et al. developed a technique for the energy profiling of devices and components based on data gathered by a logger application [5]. The key idea is to gather smartphone usage information and performance statistics with a userspace Logger application on Android. The gathered data is then uploaded to a server by the logger. The data is used to analyze usage patterns and for building a power-estimation model. The power estimation is based on linear regression and the scheme is instrumented with offline power measurements with a multimeter. Linear regression is used to fit a power-consumption variable to a set of independent parameters, the statistics collected by the logger. The aim is to find the relationships between the statistics that the logger has collected and smartphone power consumption. The main limitation of the system is that it requires the offline device-specific calibration phase.

Assuming that we have n hardware components and m samples, then in each sample, i, we have n measurements $\beta_{i,j}$. Given the n components and m samples, we can construct the measurement matrix X.

More formally the model takes the form,

$$P = k \times e + Xc, \tag{10.2}$$

where P is the total system power, k is a constant offset of power not attributable to any available measurement, $e = (1 \dots 1)^T$, X is the measurement matrix, $c = (c_0, c_1, \dots, c_n)$ for n measurements is the regression coefficient vector.

Given a single measurement, the power, $p_{i,j}$, contributed by a hardware component with coefficient, c_j, is given by $p_{i,j} = \beta_{i,j} \times c_j$. Given that k and c have been determined, this equation can be used to determine the power due to each hardware component. Similarly, Eq. (10.2) gives the power consumed by the whole system with any sample of measurements.

This work approximated the values of k and c with offline system measurements, X, with the measured power consumption of P. Different workloads are used to generate logs for the creation of the model. The training logs relate to specific hardware components of the smartphone. The generated model is then validated with use-case-specific workloads that are minutes long and combine various smartphone functions. The reported resulting median relative error rate is less than 0.1%.

10.4.4 Fine-grained profiling with Eprof

The Eprof [14] system answers the question where is the energy spent inside applications. The system is a fine-grained energy profiler that uses OS-level instrumentation to determine where energy is spent and identify power states and power-state transitions. The system uses an offline process to construct an FSM-based model for a smartphone. The FSM can capture power-state transition details, such as the the tail effect observed with wireless communications. The FSM can also correlate energy draw with source code. The system was used to find a family of energy bugs in applications [14]. The Eprof profiler is investigated in more detail in Section 9.4.

Table 10.2. Example power model based on regression

Component	Parameter	Description	Range ($\beta_{i,j}$)	Coefficient (c_j)
CPU	High	Average CPU usage at 384 MHz	0–100	3.97 mW/%
	Medium	Average CPU usage at 246 MHz	0–100	2.79 mW/%
Screen	Screen on	Fraction of time screen is on	0–1	150.31 mW
	Brightness	Screen brightness	0–255	2.07 mW/step
Call	Ringing	Fraction of time phone is ringing	0–1	761.70 mW
	Off Hook	Fraction of time interval during a call	0–1	389.97 mW
EDGE	Has traffic	Fraction of intervals with EDGE traffic	0–1	522.67 mW
	Traffic amount	Number of bytes over EDGE	≥ 0	3.47 mW/byte
Wi-Fi	On	Fraction of time with Wi-Fi	0–1	1.77 mW
	Has traffic	Fraction of time with Wi-Fi traffic	0–1	658.93 mW
	Traffic amount	Number of bytes over Wi-Fi	≥ 0	0.518 mW/byte
SD Card	Traffic	Number of sectors transferred	≥ 0	0.0324 mW/sector
DSP	Music on	Fraction of time with music playback	0–1	275.65 mW
System	System on	Fraction of time not idle	0–1	169.08 mW

10.4.5 Summary

Table 10.3 presents a comparison of the three offline energy profilers presented above. PowerScope is the oldest of these and requires kernel access for the instrumentation as well as the power-measurement phase. A server is used to generate the power model based on the observed software traces and the power measurements. Joule Watcher also relies on a server and an external multimeter. The system by Shye et al. does not require kernel or low-level access, but instead uses a userspace logger application for monitoring the device and a multimeter for the power measurements. An external server is also needed in this case.

The Eprof system follows a similar strategy, it instruments the kernel to obtain fine-grained information regarding system calls. This information is then correlated with the external power measurement. All three can profile the energy consumption of system components and threads. The Eprof system also provides more fine-grained analysis of energy consumption.

In a comparison of the three frequently used energy accounting techniques–usage based (such as PowerTutor), time based (such as Powerscope), and system-call based

Table 10.3. Offline energy profilers

Name/Authors	Year	Purpose	Measures	Model	Accuracy	Kernel or low-level access
PowerScope	1999	Energy profiling of device and processes	Current and voltage, external	Current integration, time-based model	Depends on sampling interval and multimeter	Yes
Joule Watcher	2000	Fine-grained thread-level profiling	Thread energy consumption	Hardware registers (event counters)	Depends on the multimeter and the accuracy of the statistics	Yes
Shye et al.	2009	Energy profiling of device and components	Voltage, current, external	Current integration, usage, linear model	Median absolute relative error 6.6% Less than 0.1% mean error when comparing the total energy	No
eProf	2012	Power model for device, components, and applications	System calls and energy	Finite state machine (FSM)	Error under 6%, high rate of 20 Hz	Yes

(Eprof) the–usage-based energy accounting was observed to have errors in the range 3–50%, the time based between 15 and 80%, and Eprof below 6% [3]. The Eprof profiler is more accurate, because it operates on the level of system calls which is more fine-grained than thread or process level. Usage-based profiling has been shown to demonstrate larger errors when working at finer granularities, because they do not capture power state level details and asynchronous power behavior [19, 3].

10.5 Online on-device energy profilers with offline support

In this section, we examine online on-device energy profilers that rely on an offline calibration and measurement phase. Profilers in this category are different from the pure offline category, because the profiler is mainly expected to operate on the device in an online manner. In the following, we briefly examine PowerBooter, PowerTutor, and battOr. At the end of the section, we summarize the key similarities and differences of the systems.

10.5.1 PowerBooter

PowerBooter is an automated power model creation technique that uses on-device voltage sensors and battery discharge curves based on the Rint model to estimate power consumption [6].[1] The power consumption is then correlated with individual components using regression. The system does not require external measurement equipment; however, a smartphone-specific discharge curve is needed. The new idea of PowerBooter was to use battery-state-based power-model generation. This involves keeping smartphone components in specific power states so that their power consumption can be determined through the change in battery SOD using a voltage sensor. This change can be used to estimate the average power draw. When the component-specific average power draw is known, it is possible to derive the power model using regression.

Figure 10.1 illustrates the key phases of PowerBooter. In the first step, the battery discharge curve is constructed for the phone. The discharge curve varies from phone to phone due to differences in battery type, age, temperature, and operating parameters. The discharge curve can be obtained online and on-device by observing the constant discharge behavior from a fully charged state. In the second step, the power consumption is determined for each component state. The state of a component is varied while keeping the rest of the system in a static configuration. The battery voltage is recorded at the beginning and end of a discharge interval. The voltage is measured for 1 minute and the battery is discharged for 15 minutes between the component voltage measurements. In the third step, regression is used to create the power model. The battery voltage differences for each discharge interval is used to determine the average power draw of the

[1] We have included PowerBooter in this category with the assumption that the discharge curve is given. With automatic on-device discharge curve determination, this technique is in the online on-device category.

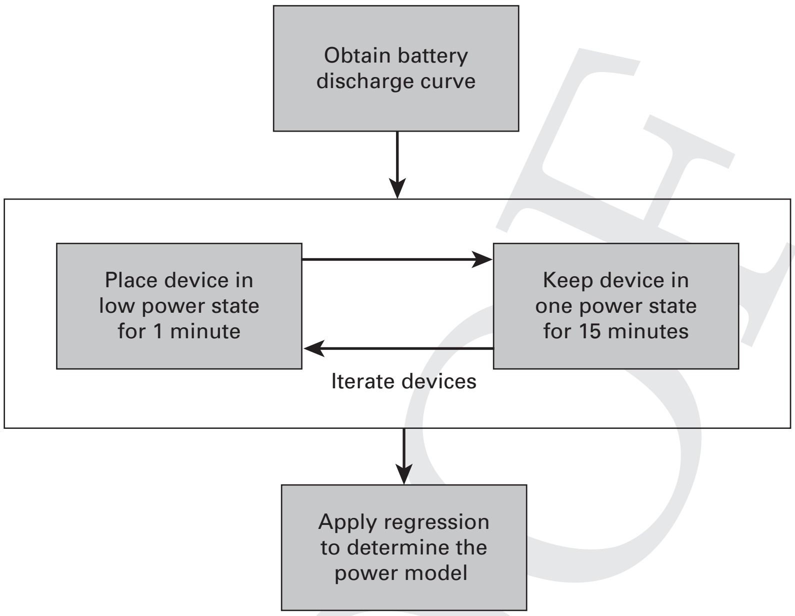

Obtain battery discharge curve
Place device in low power state for 1 minute
Keep device in one power state for 15 minutes
Iterate devices
Apply regression to determine the power model

Figure 10.1 Overview of the PowerBooter model

15-minute intervals. Regression is then used to create the power model based on the component average power draw estimates.

Table 10.4 presents an example PowerBooter power model for the HTC Dream smartphone [6]. The model includes categories for CPU, Wi-Fi, cellular network, audio, LCD, and GPS. Some of the categories have multiple variables for accurately modeling the subsystem, such as the CPU, Wi-Fi, and the cellular module. Equation 10.3 gives the power model describing the overall power consumption based on the power draw of the components.

$$\begin{aligned}
&(\beta_{uh} \times freq_h + \beta_{ul} \times freq_l) \times util + \beta_{CPU} \times CPU\_on \\
&\quad + \beta_{br} \times brightness + \beta_{Gon} \times GPS\_on + \beta_{Gsl} \times GPS\_sl \\
&\quad + \beta_{Wi-Fi\_l} \times Wi - Fi_l + \beta_{Wi-Fi\_h} \times Wi - Fi_h + \\
&\quad + \beta_{3G\_idle} \times 3G_{idle} + \beta_{3G\_FACH} \times 3G_{FACH} + \\
&\quad + \beta_{3G\_DCH} + 3G_{DCH}.
\end{aligned} \tag{10.3}$$

10.5.2 PowerTutor

The PowerTutor[2] application is based on PowerBooter power models. This Android application shows energy use in a similar way to Android's built-in profiler, but with

[2] `http://powertutor.org` accessed January 7, 2014.

Table 10.4. PowerBooter power model for HTC Dream smartphone

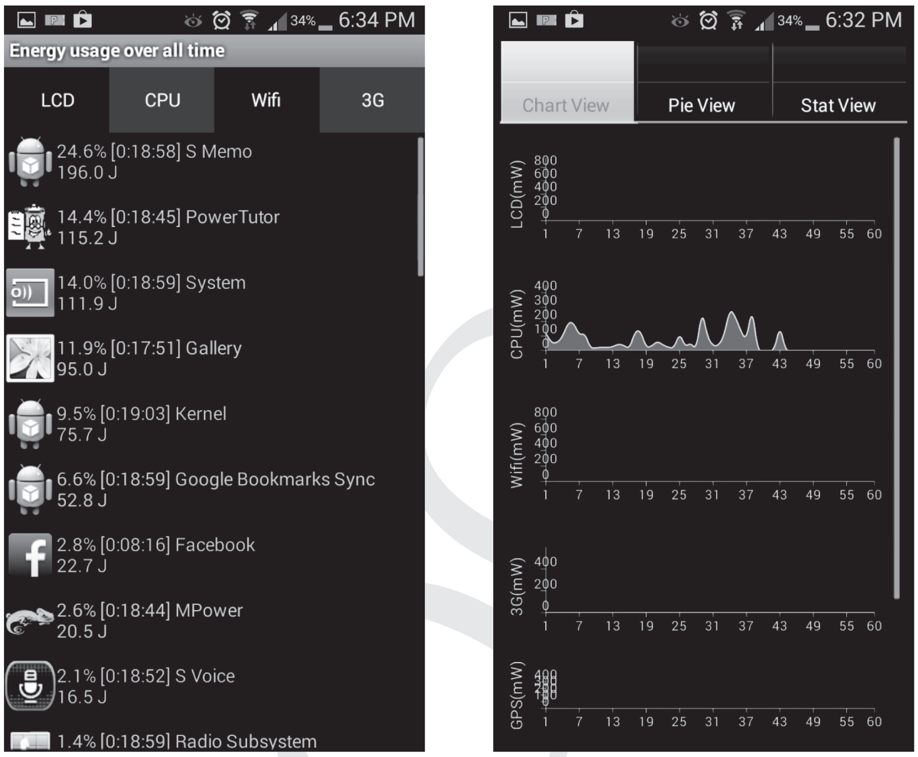
Category	System variable	Range	Power coefficient
CPU	util	1–100	$\beta_{uh}: 4.34$ $\beta_{ul}: 3.42$
	freq_l and freq_h	0,1	–
	CPU_on	0,1	$\beta_{CPU}: 121.46$
Wi-Fi	npackets, R_{data}	0–	–
	$R_{channel}$	1–54	$\beta_{cr}(R_{channel}) = 48 - 0.769 \times R_{channel}$
	Wi-Fi_l	0,1	$\beta_{Wi-Fi\_l}: 20$
	Wi-Fi_h	0,1	$\beta_{Wi-Fi\_h}: 710 + \beta_{cr}(R_{channel}) \times R_{data}$
Cellular	data_rate	0–	–
	downlink_queue	0–	–
	uplink_queue	0–	–
	$3G_{idle}$	0,1	$\beta_{3G\_idle}: 10$
	$3G_{FACH}$	0,1	$\beta_{3G\_FACH}: 401$
	$3G_{DCH}$	0,1	$\beta_{3G\_DCH}: 570$
Audio	Audio_on	0,1	$\beta_{audio}: 384.62$
LCD	brightness	0–255	$\beta_{br}: 2.40$
GPS	GPS_on	0,1	$\beta_{Gon}: 429.55$
	GPS_sl	0,1	$\beta_{Gsl}: 173.55$

breakdowns per resource, such as CPU, Wi-Fi, and the screen, and per category, such as how different applications have consumed energy or how different components of the phone have been draining the battery. The PowerTutor application and chart views are shown in Figure 10.2. Like the Nokia Energy Profiler, PowerTutor relies on power profiles that have been created for a handful of devices. On other devices, measurements will be less accurate.

PowerTutor uses the procfs and the Android BatteryStat service to obtain information about usage statistics. PowerTutor does not consider the effects of running multiple applications simultaneously but rather estimates the energy consumption for each application separately. This approach simplifies the estimation and mitigates the challenge of dividing the power consumption of hardware components to multiple applications.

10.5.3 BattOr

The BattOr profiler is an interesting example of a hardware-based profiler that is attached to the smartphone battery or a USB for power [12]. This profiler can give a reliable power estimate of the smartphone. The main advantage of BattOr is mobility – it can be carried with the smartphone. The measurements are stored on a SD memory card while the the device is mobile and they cannot be used at runtime. When the device is stationary it is possible to stream the power measurements. BattOr requires copper tape wiring to the phone that typically requires instrumentation in a laboratory.


Energy usage over all time

LCD CPU Wifi 3G

24.6% [0:18:58] S Memo
196.0 J

14.4% [0:18:45] PowerTutor
115.2 J

14.0% [0:18:59] System
111.9 J

11.9% [0:17:51] Gallery
95.0 J

9.5% [0:19:03] Kernel
75.7 J

6.6% [0:18:59] Google Bookmarks Sync
52.8 J

2.8% [0:08:16] Facebook
22.7 J

2.6% [0:18:44] MPower
20.5 J

2.1% [0:18:52] S Voice
16.5 J

1.4% [0:18:59] Radio Subsystem

Figure 10.2 PowerTutor attributes energy use to applications and subsystems

10.5.4 Summary

Table 10.5 presents a summary of the profilers examined in this section. The purpose of the profilers varies from fine-grained thread-level profiling to portable power monitoring with BattOr. PowerBooter relies on an offline phase for calibrating the battery model that determines the discharge curve. The discharge curve is then used online to determine the energy consumption given the voltage readings. It should be noted that PowerBooter also has a fully automatic on-device process for determining the discharge curve.

Kernel- and low-level access are typically required for component-level operations; however, it is also possible to implement the profilers in the userspace. For example, the PowerTutor application based on PowerBooter model runs in the userspace and accesses the Android OS features for the statistics.

10.6 Online on-device energy profilers

In this section, we survey online on-device energy profilers that do not need external power measurement or offline calibration. We briefly examine the Nokia Energy

Table 10.5. Online energy profilers with offline component

Name	Year	Purpose	Measures	Model	Discharge curve	Accuracy	Kernel or low-level access
PowerBooter	2009–2010	Short-term model for components	Load voltage	OCV Thevenin, usage linear model	Required, can be automated for fully online operation	Error 4.1% for 10 second intervals, 0.1 Hz	No
PowerTutor	2009–2010	Energy profiler application for Android	Application energy consumption based on hardware component usage	PowerBooter	PowerBooter	Component-level average error 0.8% and reported maximum error 2.5% 1 Hz rate	No
BattOr	2011	Portable power monitor	Voltage, current	External measurement	No	High	Yes (hw)

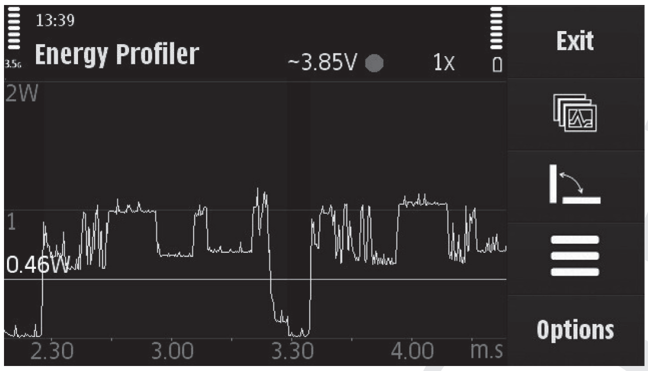


Figure 10.3 Nokia Energy profiler, running on an N97 smartphone, recording estimated power use in watts

Profiler, PowerProf, Sesame, DevScope, AppScope, and V-Edge. At the end of the section, we examine the similarities and differences of the profilers.

10.6.1 Nokia Energy Profiler

The Nokia Energy Profiler (NEP) is a stand-alone measurement application for the Symbian OS that can simultaneously monitor multiple parameters, such as uplink and downlink rates, WLAN and cellular signal strengths, and CPU usage [4]. The NEP tool is a software-based energy-consumption profiler that is run on the mobile device being profiled. Power-consumption measurements are implemented by observing the battery voltage and current drain values. Battery current values are truly average values, because the hardware integrates current consumption in the analog domain over the entire measurement period. An important characteristic for any kind of instrumentation is that it should introduce minimal disturbance to the actual measurements. This characteristic must apply particularly for an energy-profiling application. In other words, NEP's processing and sleep periods have to be heavily optimized for low-power operation. This is important because any kind of processing increases the total energy consumption.

Early consumer tools for energy awareness on smartphones include the NEP [20] shown in Figure 10.3. This tool is used manually, and once activated it records a trace of power usage. The user performed some activities with the device, and then returned to the NEP. The application showed the amount of energy that had been used, in watts, as a graph over time. The tool also recorded the Wi-Fi and mobile network activity, battery voltage and current, and screen use. NEP ran on Symbian devices and relied on device profiles built by the device manufacturer.

10.6.2 PowerProf

PowerProf [11] is an example of an unsupervised API-level energy profiler that uses genetic algorithms. The basic premise is similar to PowerBooter; however, now genetic algorithms are used to generate the model. PowerProf allows API-level tracking of the energy consumption. The genetic algorithms operate on the on-device measurements and estimate a power model. The power model has one conditional function for each phone feature. A conditional function models the feature's power consumption through four different power states defined by time parameters. PowerProf uses the NEP to obtain battery information. The main focus of the work is to predict power consumption.

10.6.3 Sesame

Sesame is another example of an online energy profiler that automatically generates the power model [7]. This proposal uses the smart battery interface and a number of techniques to increase the accuracy and rate of the battery interface. They observed that the smart battery interfaces are limited in terms of the update rate (4 Hz being the highest observed), the error of the instant battery interface reading is high, and that the instantaneous errors can be mitigated by averaging the readings thus reducing the rate. Sesame aims to realize adaptive models that statistically find the best predictors. It uses a variant of linear regression, the total-least-squares method, to minimize the impact of predictor errors. Sesame also uses model molding to improve the accuracy and rate by first building accurate but low-rate models, and then compressing these to obtain high-rate models. A technique called predictor transformation is used to improve the accuracy of a molded model by applying principal component analysis (PCA).

Model molding consists of two phases that improve the accuracy and rate. The first phase, called stretching, creates an accurate energy model at a much lower rate than needed. This phase is especially useful if the smart battery interface does not give reliable readings. This technique is inspired by the observation that the smart battery interface has a slow update rate and that a more accurate estimate can be achieved by averaging over several readings. For example, if the target rate is 100 Hz, Sesame first constructs an accurate model at 0.01 Hz. The accurate low-rate model is simply constructed by averaging the energy consumption readings over an interval. The second step compresses the low-rate model to derive a high-rate model. The compression is achieved by applying the linear regression coefficients in calculating the energy consumption for the desired time interval.

The key finding of Sesame is that the accuracy of the smart battery readings can be improved with the proposed techniques. For example, an accuracy of 86% at 1 Hz and 82% at 100 Hz are reported using the Nokia N900 smart battery interface with an accuracy of 55% at 0.1 Hz. The accuracy is lower at a higher rate. This reduction in accuracy is due to the overhead of data collection, the non-linearity of power behavior and the relation of power consumption and system statistics, and invisible components that are not included in the model.

10.6.4 DevScope

DevScope [8] is an example of a energy profiler that uses the smart battery interface to generate an on-device dynamic linear regression-based power model. The DevScope authors observed that the smart battery interface has a low update rate. They proposed a synchronization technique between the update rate and component-specific control. The profiler works by probing the OS to obtain information about the components and the configuration, such as the CPU details. The profiler also examines the smart battery interface and determines the update rate of the battery interface. Similar to PowerBooter and Sesame, the profiler then creates a component control scenario for power analysis for the specific smartphone. The control scenario is then run and DevScope first classifies the data to the terms of the power model and then analyzes the classified data to update the power coefficients of the regression model. For example, each CPU frequency is tested with zero and maximum use to derive the information needed for the CPU model.

To alleviate the slow update rate of the smart battery interface, DevScope synchronizes the smart battery update events with the component tests. This contrasts with Sesame's solution of averaging battery readings for higher accuracy readings at a slower rate. DevScope also tries to recognize power-state transitions; however, this requires knowledge of the power-state durations and the battery update interval. Automatic detection of power-state transitions is difficult, because the state transitions are governed by the workload and the operating conditions. During component testing, DevScope uses different workload sizes repeatedly to determine the threshold size that results in a power-state change. This technique is applied for cellular and Wi-Fi connections to determine the wireless network parameters and power state details.

10.6.5 AppScope

AppScope is an application energy-profiler framework for Android that uses the DevScope hardware power models and usage statistics for each hardware component [10]. The usage statistics are based on kernel-level monitoring of activities for hardware component requests. The AppScope framework has three key phases. The first phase is about detecting process requests that involve hardware access. The second phase is about analysis of usage statistics. The third phase involves a linear model-based application estimation by summing up the energy consumption of each hardware component used by an application.

The hardware component usage analyzer is responsible for collecting the component-specific usage data that is needed to apply the power model. The analyzer uses hardware-component to specific methods to collect the data, for example the Linux Governor interface is used to obtain the CPU usage and frequency and the network interface provides the Wi-Fi and cellular data. The Android IPC messaging is analyzed for LCD display and GPS statistics.

The AppScope is reported to have an error rate below 7.5% during a 100-second experiment of typical smartphone applications that do not use the GPU. The GPU-based application, the Angry Birds game, in the experiment resulted in a 14.7% error due to the fact that the GPU is not explicitly modeled by AppScope.

Experimental validation of the accuracy of AppScope reported that all applications, with the exception of Angry Birds, showed an error rate below 7.5%.

10.6.6 V-Edge

V-Edge [9] uses the battery voltage dynamics based on the Thevenin model to generate a power model. The system is similar to PowerBooter, Sesame, and DevScope in that it aims to generate the model without external measurement. V-Edge does not require current information and it estimates power consumption based on the instantaneous voltage that is calibrated once with SOD metering. This model avoids the slowness of fully SOD-based approaches. The actual model is constructed in a similar manner to PowerBooter by running carefully designed tests to determine the power consumption of system components.

V-Edge determines the battery interface update interval in a similar manner to DevScope. The smartphone is first placed into idle mode that is longer than the battery update interval. The CPU usage is then increased, after which the voltage values are sampled at 1 Hz. When the voltage drop due to the increased CPU use is detected at time $t1$, the CPU is placed into idle mode again. When the new voltage change is detected the sampling is stopped at time $t2$. The update interval is the difference, $t2 - t1$. When this interval is known, the detection of the instantaneous current, the V-Edge, can be bounded to two seconds. This then facilitates accurate current measurement based on V-Edges. The OCV-based V-Edge technique is discussed in more detail in Chapter 3.

10.6.7 Summary

Table 10.6 presents a summary of the above online profilers. The NEP relies on low-level access to the battery interface on Nokia devices. The tool has an internal model calibrated by Nokia engineers. Thus this application is not truly online as it has certain features that have been calibrated. Nevertheless, we have included it in this category, because the developer does not have to have an external power-monitoring device to use the tool. PowerProf is an API-level energy profiler that relies on the NEP API. Sesame, DevScope, AppScope, and V-Edge are examples of truly online profilers that probe the device and its battery interface to determine the operating conditions and to create test cases for creating the power model on the fly. AppScope estimates the energy consumption of applications and processes based on DevScope models in a similar manner to PowerTutor without requiring device type specific offline calibration.

10.7 Energy diagnosis engines

In this section, we survey energy diagnosis engines that aim to find and analyze energy-consumption anomalies.

10.7.1 MobiBug

The MobiBug [13] system advocates the use of failure reporting for mobile devices. The idea is to gather the failure reports and statistics to a server and then diagnose mobile applications. The analysis builds on three fundamental changes for failure reporting:

Table 10.6. Online energy profilers

Name/Authors	Year	Purpose	Measures	Model	Accuracy	Kernel or low-level access
Nokia Energy Profiler	2006–2007	On-device standalone profiler	Current, capacity, voltage (Smart Battery interface)	Internal	Sample rate between 0.2 and 4 Hz	Yes
PowerProf	2011	Application-level profile	API call energy consumption	Genetic algorithm	Uses Nokia Energy Profiler	Varies
Sesame	2011	Power model for device and components	Current, capacity, voltage (Smart Battery interface)	Linear regression and molding, usage	86% accuracy at 1 Hz, 82% accuracy at 100 Hz on a smartphone	Yes (D-Bus), can also be userspace
DevScope	2012	Power model for device and components	Current, focus on battery updates	Usage linear model	Varies	Yes
AppScope	2012	Power model for applications	Component usage, builds on DevScope	Builds on DevScope, linear model	Error rate below 7.5% for applications that do not use GPU	Yes
V-Edge	2013	Power model for device and components	Voltage, instant dynamics	OCV Thevenin, usage linear model	Average error within 4%	Yes

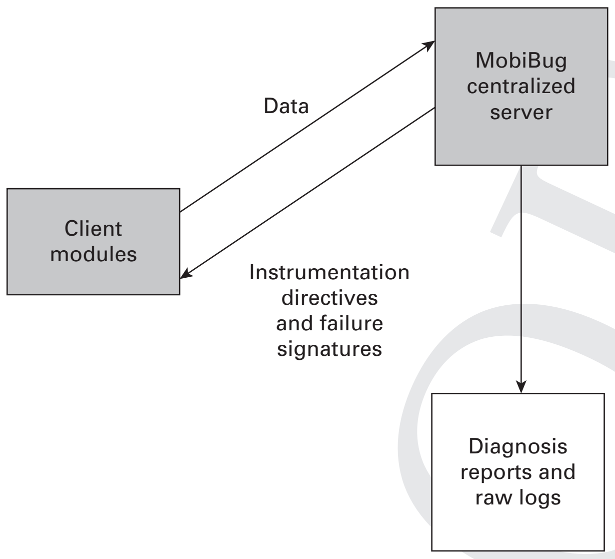

MobiBug centralized server
Data
Client modules
Instrumentation directives and failure signatures
Diagnosis reports and raw logs

Figure 10.4 Overview of the MobiBug system

- spatial spreading that spreads monitoring load across devices,
- statistical inference that is used to model the application behavior and its dependencies, and
- adaptive sampling that instruments the data collection across phones so that the system learns about the failures.

Figure 10.4 depicts the MobiBug system. The purpose of the system is to provide clues and automatic insight into the nature of software failures for developers. This is achieved with the spatially spread monitoring of features enabled on the devices using the system, and concentrating on software failures not seen before.

MobiBug's spatial spreading of monitoring enables data to be gathered from diverse environments with smaller total bandwidth costs. Each time a failure report is obtained, it is compared against known failures. The more common a failure is, the more information is gathered about it, and the more complete the view of its environment of occurrence will be. When a failure is thoroughly explored, the system will stop instrumentation related to it, and concentrate on more rare failures. This will help developers quickly gather a complete picture of the failure without incurring data costs for all users.

To the best of our knowledge, the MobiBug system was not deployed and is not accessible to the public at the time of writing.

10.7.2 eDoctor

The eDoctor system is a recent example of a tool that helps regular users solve problems with abnormal battery drain. eDoctor runs on smartphone xxx and examines application-execution phases to capture the runtime behavior of an application and then identifies abnormal execution phases that correlate with high energy draw. Based on this analysis, the system can then recommend repair solutions to the end users [21].

Execution phases are identified by eDoctor based on active resources used by the application, such as the GPS, network connectivity, or CPU use. The k-means clustering algorithm was used to find phases within the execution intervals of applications. When an application starts to exhibit previously unseen phases, or phases of abnormal length, this is investigated as a potential problem. Together with configuration changes and other information for an application, eDoctor can detect applications with abnormal energy drain and identify a possible root cause.

The eDoctor data collector gathers data between recharges of the phone battery. All analysis is done locally, the application does not send data out from the smartphone. Therefore the application cannot detect energy issues caused by the local platform, or those already present on the device when eDoctor is first used.

When eDoctor detects issues, it recommends installing a previous version of the application, if the behavior manifested after an upgrade. It suggests reverting the application's configuration, if better performance was obtained with previous values, and if the user wishes to use the problematic application, it recommends keeping it closed between uses.

The eDoctor system was developed as an Android application, but it is not publicly available at the time of writing.

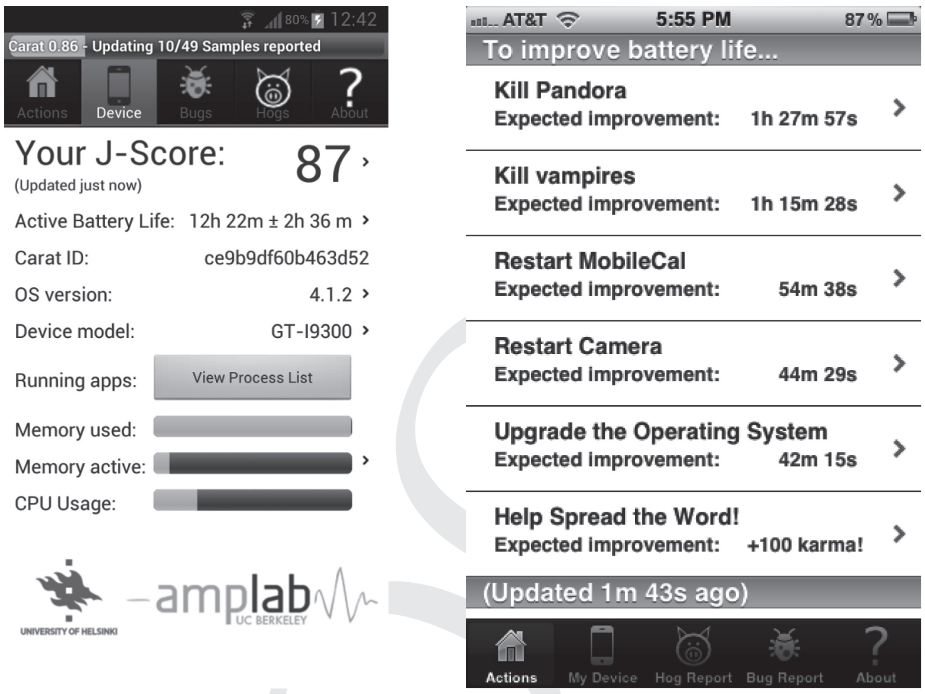
10.7.3 Carat

To the best of our knowledge, Carat [15] is the first collaborative approach to mobile battery awareness. The application can provide information about possible energy problems with a very low energy cost and minimal user attention. On a single device, it is not possible to diagnose abnormal energy use. High energy use could be normal, specific to a device, or a single user. Information from multiple devices is needed to identify incorrect energy behavior of an application. By tapping into its community of devices, Carat can identify applications that are globally high energy consumers or anomalous energy users on particular devices.

In the Carat approach, data from the community of smartphones was combined to find the average energy consumption of applications. Then, if the application exhibits higher energy consumption on a subset of the smartphones, the distribution of energy-consumption measurements on the subset and other smartphones is compared. If the 95% confidence error bars of the two distributions do not overlap, the application is marked as a bug in the subset of smartphones. Bugs are further diagnosed with conditions like Wi-Fi enabled or disabled, device moving or stationary, and which OS version is used.

Carat has been implemented as an application on both the iOS and Android platforms. It is available as a free download from Apple's App Store and Google's Play Store. The application is opensource.[3] The client application sends intermittent, coarse-grained

[3] `http://carat.cs.berkeley.edu` accessed January 6, 2014.



80% 12:42
Carat 0.86 - Updating 10/49 Samples reported
Actions | Device | Bugs | Hogs | About

Your J-Score: 87
(Updated just now)

Active Battery Life:	12h 22m ± 2h 36 m
Carat ID:	ce9b9df60b463d52
OS version:	4.1.2
Device model:	GT-I9300
Running apps:	View Process List
Memory used:	
Memory active:	
CPU Usage:	

UNIVERSITY OF HELSINKI
amplab UC BERKELEY

AT&T 5:55 PM 87%
To improve battery life...

Kill Pandora	Expected improvement: 1h 27m 57s
Kill vampires	Expected improvement: 1h 15m 28s
Restart MobileCal	Expected improvement: 54m 38s
Restart Camera	Expected improvement: 44m 29s
Upgrade the Operating System	Expected improvement: 42m 15s
Help Spread the Word!	Expected improvement: +100 karma!

(Updated 1m 43s ago)
Actions | My Device | Hog Report | Bug Report | About

Figure 10.5 The Carat application advises users on how to improve battery life on iOS and Android

measurements to a server. The server correlates the running applications, device model, and operating system with energy use. The system generates actions that the user could take to improve the battery life, shown in Figure 10.5. The actions take the form of suggestions to stop or restart applications, or to change the OS version. The amount of improvement, error, and confidence of the suggestions given by Carat is calculated and presented to the user along with the actions. Carat has been installed on more than 600,000 devices.

How Carat distinguishes energy bugs

For each application, a, Carat calculates the average battery drain, e_a, across the entire community, and the average battery drain of the *average application*, essentially the average battery drain of the whole userbase of Carat, E:

$$E = \frac{\sum_a^n e_a}{n}. \tag{10.4}$$

Carat then marks applications that have $e_a > E$ as *hogs*, if the difference between the distribution of the battery drain measurements for a and that for all applications is significant. Significance is discussed in detail in Section 10.7.3.

Hogs are typically applications that drain the battery faster because of their nature; examples of hogs include VoIP, internet radio, and streaming video applications. For the set of applications that are not hogs, Carat splits up the battery drain for each device, u, into the drain observed on that device, e_{au}, and on other devices, $e_{a\neg u}$. If $e_{au} > e_{a\neg u}$ and the difference of the distributions is significant, Carat marks the pair (a, u) a *bug*.

Energy bugs may be caused by a code error that only triggers under certain conditions (which our analysis tries to discover), configurations, or user behaviors. Distinguishing between hogs and bugs requires a collaborative method.

Battery drain rate distributions

Carat obtains a sample of the battery level and a list of running applications from its client devices. For consecutive pairs of such samples, Carat creates a battery drain rate probability *distribution*, r. The expected value, v, of this distribution is the expected battery drain rate during the time period delimited by the two samples. Taking the expected values of all the rates that satisfy a condition, c, for example "Facebook was running on device u" we get a set of means, $V_c = \sum_i^n v_i$.

These are extracted from a large number of random i.i.d. variables that represent the true drain at each moment in time on participating devices, which is approximately normally distributed as $\mathcal{N}(\mu, \frac{\sigma^2}{n})$, according to the central limit theorem (CLT). Although the parameters μ and σ^2 are not known, they can be estimated using the rates probability distributions $(r_1, \ldots, r_n)$. The well-known maximum likelihood estimators for these parameters–obtained by maximizing the log-likelihood function–are as follows [15]:

$$\hat{\mu} = \bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n} (r_i - \bar{r})^2.$$

By the Lehmann–Scheffé theorem, $\hat{\mu}$ is the uniformly minimum variance unbiased estimator for μ: $\hat{\mu} \sim \mathcal{N}(\mu, \frac{\sigma^2}{n})$.

This agrees with the CLT method. The estimator $\hat{\sigma}^2$, however, is biased, so Bessel's correction must be applied to obtain the uniformly minimum variance unbiased estimator for the sample variance:

$$s^2 = \frac{n}{n-1}\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n} (r_i - \bar{r})^2.$$

By our normality assumption, we can construct the t-statistic, $t = (\hat{\mu} - \mu)/(s/\sqrt{n})$, which has the student's t-distribution with $n - 1$ degrees of freedom. We can approximate the error bounds on this estimate of μ using a standard formula, where h is chosen according to the desired confidence level. (For 95% confidence error bounds, $h = 1.96$):

$$\mu \approx\in \left[\hat{\mu} - \frac{hs}{\sqrt{n}}, \hat{\mu} + \frac{hs}{\sqrt{n}}\right] = \hat{\mu} \pm \epsilon$$

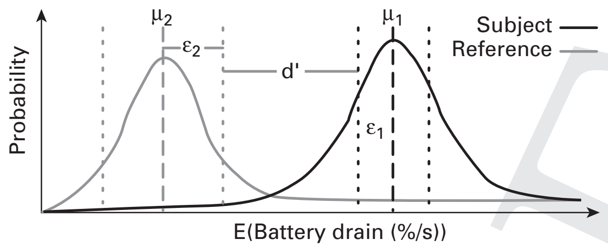


Figure 10.6 Carat compares distributions of the expected values of battery drain to identify anomalies ($d' > 0$) and quantify the error and confidence ranges for expected battery drain under different conditions

Significance in battery drain distributions

Let c_1 be the conditions of the subject distribution (e.g., application A is running) and c_2 be the conditions of the reference distribution (e.g., application A is not running). We aim to ascertain whether c_1 corresponds to *significantly greater* energy use than c_2. For this to be answered in the affirmative, we require the following:

$$\hat{\mu_1} - \frac{hs_1}{\sqrt{n_1}} - \hat{\mu_2} - \frac{hs_2}{\sqrt{n_2}} = \hat{\mu_1} - \hat{\mu_2} - (\epsilon_1 + \epsilon_2) > 0.$$

Otherwise, the data does not support the assertion with the desired confidence. Graphically, this corresponds to a positive value of d' in Figure 10.6.

Carat suggests actions that would improve battery life along with the expected value of that improvement for an average client (starting from full charge and fully draining the battery). The improvement if the client was to change from c_1 (experiencing the anomaly) to c_2 (not experiencing it) follows directly from the distance metric $d = \hat{\mu_1} - \hat{\mu_2}$. Within our confidence bounds, however, the value of d could be as much as

$$e = h\left(\frac{s_1}{\sqrt{n_1}} + \frac{s_2}{\sqrt{n_2}}\right).$$

The estimated improvement is therefore $d \pm e$. The distributions are in terms of battery drain, so to convert the improvement into battery life in hours, they need to be inverted:

$$b_1 = \frac{100}{\hat{\mu_1} \times 3600}. \tag{10.5}$$

The worst-case subject distribution battery life within the 95% confidence error bounds is then

$$b_{1w} = \frac{100}{(\hat{\mu_1} + \epsilon_1) \times 3600}, \tag{10.6}$$

and the best-case reference distribution battery life is

$$b_{2b} = \frac{100}{(\hat{\mu_2} - \epsilon_2) \times 3600}. \tag{10.7}$$

Then

$$i_{max} = b_{2b} - b_{1w} \qquad (10.8)$$

gives the maximum battery life improvement in hours of battery life. If the best-case subject distribution battery life is compared instead against the reference worst case we get the minimum improvement

$$i_{min} = b_{2w} - b_{1b}. \qquad (10.9)$$

These give the 95% confidence bounds for the improvement that Carat can present to the user. As Carat gathers more data, the error associated with these improvement numbers decreases.

10.7.4 Diagnosing energy bugs with Eprof

The Eprof energy profiler examined with offline profilers is used to identify energy bugs in mobile applications. Energy bugs are a new breed of system misbehaviors. An energy bug manifests in an application that exhibits excessive energy consumption due to a software bug or hardware problem. The Eprof system was developed for fine-grained application energy consumption analysis and for pinpointing energy hotspots in source code.

Figure 10.7 gives an overview of discovered energy bugs by their type [22]. The taxonomy divides energy bugs into unknown, software, external, and hardware categories. The software category consists of OS- and applications-related bugs. The application category includes three different bug types: no-sleep, loop, and immortality. The no-sleep bug relates to applications that do not properly release a wake lock thus preventing the device from sleeping. The loop bug relates to situations where a part of an application unnecessarily repeats some activity. The immortality bug occurs when a buggy application is stopped by the user and is then restarted with the same buggy behavior.

In addition to bugs, the findings include the observation that 65–75% of the energy consumption of certain free applications is due to third-party advertisement modules [14].

10.7.5 Summary

Several energy diagnosis engines have been proposed recently that are summarized in Table 10.7. These systems aim to find harmful energy consumption problems and ways to mitigate them. The MobiBug system is based on failure reporting and centralized analysis of the failure reports to diagnose mobile applications. The eDoctor system takes a different view and detects abnormal battery drain online on the devices. This system aims to find abnormal execution phases that have a high energy draw and then recommend solutions to end users. The Carat diagnosis engine, on the other hand, takes a collaborative approach and uses data gathered from the community to detect energy

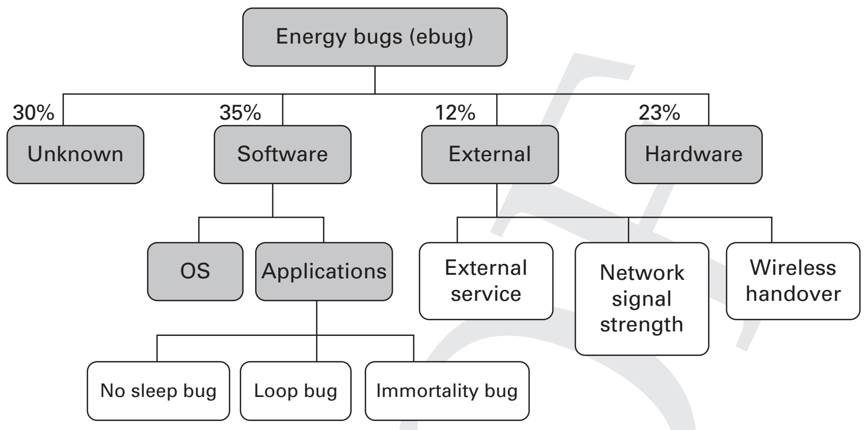


Figure 10.7 Overview of energy bugs

anomalies. Thus Carat can also identify energy problems with the hardware and mobile platform across a wide range of devices.

10.8 Summary

In this chapter, we surveyed energy profilers that are software or hardware components that monitor and characterize smartphone energy consumption. Our survey included sixteen energy profilers each belonging to one of four categories: offline or online in a laboratory setting, online and on-device with offline calibration, and fully online and on-device categories. The categories are defined as follows:

- The offline or online in a laboratory setting involves the use of an external power monitor device. Offline energy profiling typically supports fine-grained and accurate characterization of the energy consumption of the target device.
- The online and on-device with offline calibration setting places the energy profiler on the smartphone where the profiling relies on a power model that is typically created in a laboratory setting.
- The fully online and on-device with self-calibration (self-constructive) setting runs the profiler on the smartphone without any need for offline calibration or model building.
- The energy diagnosis engines aim to identify harmful energy consumption anomalies and then mitigate these with either on-device or server-based solutions.

An energy profiler can operate on different levels of abstraction that include the device, component, and process levels. Component- and process-level operation requires more information about the system state. The former requires the ability to map energy consumption to hardware components and the latter builds on this and maps processes to hardware components.

Table 10.7. Summary of energy diagnosis engines

Name/Authors	Year	Purpose	Measures	Model	Discharge curve	Accuracy	Online	Kernel or low-level access
MobiBug	2010	Automatic diagnosis of application crashes	Adaptive sampling,	Statistical model	No	N/A	No	N/A
Carat	2012	Collaborative application energy profiling and debuging	SOC with regular API	SOC-based	No	Error on the order of 0.00088%/sec	Yes, cloud backend	No
eDoctor	2013	Automatic diagnosis of battery drain problems	Application-level energy consumption, regular API	Phase-based analysis	No	N/A	Yes	Optional

Typically, power models generated in the laboratory are based on fixed use cases. These can be inaccurate with new devices and applications. The online self-constructive energy profilers aim to generate these on the fly and make them personalized so that the power models adapt to the use cases for higher accuracy.

The online information that is used together with a profiler power model, typically include SOC, voltage, and current from the smart battery interface as well as details on how hardware components are used. The underlying battery model is typically the OCV Rint or Thevenin as discussed in Chapter 3; however, not all profilers have a battery model that is used for the power estimation. For example, PowerBooter is based on the Rint and V-edge on the Thevenin models.

The online energy profilers typically employ linear models that are less accurate in modeling the energy consumption of the communications subsystem than FSM-based models. Linear models do not capture the state changes in hardware components that are necessary for fine-grained energy estimation of tail states and asynchronous system behavior.

Online on-device self-constructive profilers do not require external measurement or information to create, update, and use power models. PowerBooter with the automatic discharge curve determination, Sesame, DevScope, AppScope, and V-Edge are examples of profilers in this category that probe the device and its battery interface to determine the operating conditions and to create test cases for creating the power model on the fly.

Several energy diagnosis engines have been proposed recently. These systems aim to find harmful energy consumption problems and ways to mitigate them. The MobiBug system is based on failure reporting and centralized analysis of the failure reports to diagnose mobile applications. eDoctor runs on smartphones and identifies abnormal execution phases that correlate with high energy draw. Based on this analysis the system then recommends repair actions to users. Carat is a collaborative energy diagnosis engine that relies on crowdsourcing the data gathering. Carat identifies energy anomalies across devices and reports energy hogs and bugs to users as well as giving recommendations on how to mitigate energy problems. Carat is collaborative and distributed whereas eDoctor is a standalone application running locally on a device. In addition, the Eprof system was used to find a number of energy bugs and characterize them.

References

[1] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. 2010 USENIX Annu. Tech. Conf.* Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855861

[2] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proc. 2nd IEEE Workshop on Mobile Computer Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=520551.837522

[3] F. Bellosa, "The benefits of event: driven energy accounting in power-sensitive systems," in *Proc. 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*. New York, NY, USA: ACM, 2000, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/566726.566736

[4] G. Creus and M. Kuulusa, "Optimizing mobile software with built-in power profiling," in *Mobile Phone Programming*, F. H. Fitzek and F. Reichert, Eds. Netherlands: Springer, 2007, pp. 449–462. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-5969-8_25

[5] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures," in *Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669135

[6] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2010, pp. 105–114.

[7] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2011, pp. 335–348.

[8] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, "Devscope: a nonintrusive and online power analysis tool for smartphone hardware components," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2012, pp. 353–362. [Online]. Available: http://doi.acm.org/10.1145/2380445.2380502

[9] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: fast self-constructive power modeling of smartphones based on battery voltage dynamics," in *Proc. 10th USENIX Conf. on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2013, pp. 43–56.

[10] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "Appscope: Application energy metering framework for Android smartphones using kernel activity monitoring," in *Proc. 2012 USENIX Annu. Tech. Conf.* ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 36–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342857

[11] M. Kjrgaard and H. Blunck, "Unsupervised power profiling for mobile devices," in *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, A. Puiatti and T. Gu, Eds., vol. 104. Berlin, Heidelberg: Springer, 2012, pp. 138–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30973-1_12

[12] A. Schulman, T. Schmid, P. Dutta, and N. Spring, *Demo: Phone Power Monitoring with BattOr*, 2011, ACM Mobicom 2011. [Online]. Available: http://www.stanford.edu/~aschulm/battor.html

[13] S. Agarwal, R. Mahajan, A. Zheng, and V. Bahl, "Diagnosing mobile applications in the wild," in *Proc. 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. New York, NY, USA: ACM, 2010, pp. 22:1–22:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868469

[14] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proc. 7th ACM European Conf. on Computer Systems*. New York, NY, USA: ACM, 2012, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168841

[15] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma, "Carat: Collaborative energy diagnosis for mobile devices," in *Proc. 11th ACM Conf. on Embedded Networked Sensor Systems*. New York, NY, USA: ACM, 2013, pp. 10:1–10:14. [Online]. Available: http://doi.acm.org/10.1145/2517351.2517354

[16] N. Vallina-Rodriguez, A. Auçinas, M. Almeida, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft, "RILAnalyzer: A Comprehensive 3G Monitor on Your Phone," in *Proc. 2013 Conf. on Internet Measurement* ser. IMC '13. New York, NY, USA: ACM, 2013, pp. 257–264. [Online]. Available: http://doi.acm.org/10.1145/2504730.2504764

[17] *Power Profiles for Android*, Android Open Source Project, Nov 2013. [Online]. Available: https://source.android.com/devices/tech/power.html

[18] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 48–63, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/319344.319155

[19] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proc. 6th Conf. on Computer systems*. New York, NY, USA: ACM, 2011, pp. 153–168. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966460

[20] G. Creus and M. Kuulusa, "Optimizing mobile software with built-in power profiling," in *Mobile Phone Programming*, F. H. Fitzek and F. Reichert, Eds. Netherlands: Springer, 2007, pp. 449–462. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-5969-8_25

[21] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "eDoctor: automatically diagnosing abnormal battery drain issues on smartphones," in *Proc. 10th USENIX Conf. on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2013, pp. 57–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482634

[22] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *HotNets*, 2011.

Part III

Advanced energy optimization

11 Overview

We have so far learned the principles of how smartphones and their different components consume energy. We have also looked at the power-management mechanisms that are employed with the hardware components. Examples of such mechanisms include DVFS to scale the CPU power draw with the workload, and the use of sleep state and discontinuous reception to optimize the energy consumption due to wireless communication.

In this part, we take a look at a set of mechanisms to reduce energy consumption that are built on top of the existing power-management techniques. In this chapter, we first describe the nature and necessity of these mechanisms on a high level after which we briefly introduce the different classes of solutions.

11.1 Underlying power-management techniques alone are not enough

Although almost all the hardware contained in modern smartphones has certain kinds of in-built power-management mechanism, usually the whole device still consumes more energy than would strictly be necessary unless another layer of optimization mechanisms is employed. That slightly counter-intuitive observation stems from three main observations:

1. The hardware components are not perfectly power proportional to the offered load.
2. The energy utility of the different subsystems, especially of wireless communication, is context dependent.
3. The underlying hardware power-management mechanisms are rarely optimal for the applications being used.

We discussed power proportionality earlier in the book, particular in the case of wireless communication. Indeed, if we measure energy efficiency as energy consumed per bit of data transmitted or received, this metric is hardly ever constant for a given wireless communication technology. Instead, it usually changes as a function of the data transmission rates, for instance. There are also other contextual factors that influence the energy utility of data transmission, such as the time-dependent quality of the communication channel.

The third observation deserves a little bit more explanation. The reason why hardware power-management mechanisms are application agnostic is largely the desire to design mechanisms that work with all kinds of application. For wireless communication, there is an additional reason. The power management of radios is tightly integrated with the lowest layers of the protocol stack – the physical and medium access layers – because these layers implement the different wireless communication technologies. In addition, in a typical layered network architecture, such as the one we use today for internet communication, each layer of the protocol stack is only concerned with those functionalities that are its responsibility and is completely ignorant of the lower and upper layer protocols that it interfaces with. Cross-layer solutions are an exception, but they are in the minority. This design decision leads to a nice separation of concerns and makes application development easy. These two facts, power management being an integral part of the lowest layers of the protocol stack, and strict separation of concerns between the layers, together lead to the fact that the power-management mechanisms are usually completely ignorant of the applications.

This because observation matters applications may have several ways to reach their goals. These different ways may appear the same from the user's perspective and at the same time vastly different from the underlying hardware's perspective. Let us take a simple example: A smartphone application decides that it should synchronize its data with a cloud service (e.g. email or backup). This synchronization is not time critical and can be either performed immediately using the HSPA connectivity that is the only kind of wireless connectivity available at the moment, or it can be deferred until a Wi-Fi connection is available. The former approach uses a less energy-efficient way to transmit and receive data, which is why deferring the synchronization until the more energy efficient Wi-Fi can be used would save energy. The two wireless communication technologies are ignorant of such application semantics and cannot alone optimize the energy consumption in the above described way.

In Figure 11.1 we have redrawn the illustration of the overview of which factors contribute to the energy consumption in a smartphone that we introduced while discussing power modeling in Part 2. This time we have annotated the figure to highlight further ways to reduce the energy consumption of running a particular set of applications. In principle, there are two ways:

1. *Reduce the energy expenditure of doing a unit of work, that is improve the energy use*, such as running a piece of code or transmitting a bit of data, by either leveraging power disproportionality or the context-dependency of energy consumption.
2. *Reduce the amount of work* that needs to be done by changing the application logic and adapting it to user behavior.

The first two from our list of observations concerning energy efficiency – power disproportionality and the context-dependency of energy consumption – can be exploited to reduce the energy spent doing a unit of work. The third observation, application agnosticity, gives rise to possibilities to further save energy by changing the behavior of the application to reduce the amount of energy used or to change the nature of the workload imposed by a given application on the smartphone hardware. An additional way to optimize the application behavior is to learn user behavior and adapt to it, which we demonstrate for the case of video streaming in a later chapter.

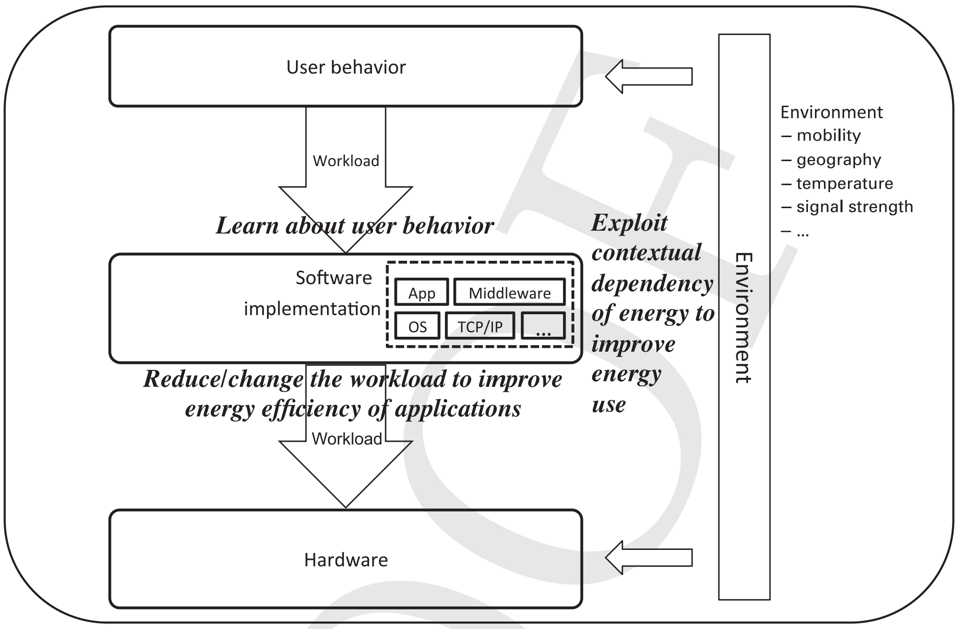


Figure 11.1 How to further reduce the energy expenditure of applications

The distinguishing feature of these mechanisms is taking to a certain degree a holistic viewpoint of energy-consumption optimization, which considers the device as a whole with its diverse capabilities and also accounts for the application specificities and user behavior. To this end, the mechanisms usually need to operate on software layers above the device drivers of an individual hardware component. Another reason for the solutions to operate on the application or middleware layer is that the application workload is sometimes impossible to change in the desired way from the lower software layers.

11.2 The importance of timing and context

We learned earlier that wireless communication is one of the biggest energy consumers in smartphones. The power disproportionality and context dependency of energy efficiency are especially important concepts in wireless communication. They make energy-aware scheduling a crucially important mechanism for achieving energy efficiency. We cover this topic in Chapter 12. There are several dimensions to consider in energy-aware scheduling. We briefly introduce them here and defer the detailed treatment of the topic to the dedicated chapter.

The power disproportionality in the wireless communication technologies used in modern smartphones is typically such that the energy utility measured, for instance, in terms of bits per joule, increases with data rate. Therefore, scheduling application traffic in such a way that the aggregate instantaneous transmission rate is maximized is favorable for the sake of energy efficiency. This can sometimes be achieved through traffic shaping, in which lower rate traffic is turned into higher rate bursts by means of buffering. Note that the average rate throughout a lifetime of a specific TCP connection, for example, can remain the same, which sometimes makes it possible for some applications to use this mechanism to extend the battery life without any noticeable change in the quality of experience (QoE) perceived by the user.

Another way to optimize energy efficiency by taking the power disproportionality into account is through energy-aware scheduling of multiple flows, possibly generated by different applications. Scheduling background traffic in this way is especially important because of its detrimental effect on the smartphone battery life. Background traffic can be generated by applications that would not otherwise use the wireless connectivity at all, such as mobile advertising embedded into applications, or as a side product of running applications that also need to transfer data as part of their core purpose, such as web browsing or video streaming. It usually consists of very small occasional transfers that consume a proportionally large amount of energy because of the tail energy phenomenon, especially when using cellular network technologies. The goal for scheduling background flows is to either bundle them in larger batches or to synchronize them with foreground transfers. In this way, it is possible to amortize a large part of the tail energy.

Scheduling the traffic in an energy-optimized way for a single smartphone is not always enough because of contention with other phones using the same wireless access network. When the shared channel is busy because another device is using it, other devices must wait for their turn during which energy is consumed because of the powered-on radio. There are several solutions that can alleviate the problem. Some are very straightforward to deploy but do not necessary solve the problem in an optimal way, while other solutions provide closer to optimal performance but would require changes in the core protocols used, for instance, in Wi-Fi networks, which makes them cumbersome to deploy.

The contextual dependency of the energy consumption of wireless communication has also been studied quite a lot, but it is still an area of active research. There are basically two contextual dimensions that influence the energy consumption: location and time. A mobile smartphone user will experience that the quality of connectivity changes with location. The reason is simply that the coverage of wireless networks varies by geographic location due to many reasons. A static user may also experience changes in the quality of wireless connectivity, which are mainly caused by the actions of others using the same access network. For example, a crowded cell of a cellular network cannot offer as high throughput for a single user as a lightly loaded one simply because the shared resources are used by a different number of users. Theses changes in the quality of connectivity directly impact the energy utility of data transfer. For this reason, the link to traffic scheduling is obvious: The objective is to schedule data communication to occur during time periods when the energy utility is as high as possible. The main challenge lies in knowing or even predicting when such periods occur.

11.3 Taking full advantage of the smartphone capabilities

As we discussed in the second part of the book, there is a lot of different hardware packed into modern smartphones. Specifically, the communication capabilities are extremely versatile because of the many technologies integrated into the phones (LTE, HSPA, GSM, Wi-Fi, Bluetooth, BLE, and NFC).

An important thing to notice is that these technologies typically have different kinds of energy consumption characteristic, which means that data transfer using a specific WNI is likely to cause a different amount of energy to be consumed than the same data transfer using another WNI. Therefore, energy can be saved if the WNI is chosen in a smart way.

The selection of WNI must take into account that the technologies strike different trade-offs in their characteristics. For example, Wi-Fi's high data rates and energy utility do not come free, since some sacrifices have been made in terms of communication range, which means that Wi-Fi coverage is typically not as ubiquitous as cellular network connectivity. Therefore, some knowledge about the availability of different alternative wireless networks must be gathered, and this is often done through prediction. There is a strong link here to traffic scheduling, because sometimes data transfers can be delayed until a more energy-efficient WNI can be used.

One of the most energy-consuming activities related to wireless communication that smartphones routinely do is trying to discover available networks. Specifically, scanning for available Wi-Fi networks is done by the phone continuously when it is not connected to an access point. If Bluetooth was used in a similar opportunistic way as Wi-Fi, it would need to continuously scan for available other devices which would be able to provide connectivity. The problem is that scanning draws a significant amount of power. Therefore, doing that all the time significantly reduces the battery life of a smartphone. Fortunately, the scanning behavior can be optimized so that it is only done when new networks could be in range, such as when the phone has moved. Movement, on the other hand, can be predicted in different ways. One way is to make use of alternative WNIs that are already powered on, such as tracking Bluetooth contact patterns or changes in the set of overheard neighboring cells of a cellular network.

In Chapter 13, we study these different solutions that take advantage of the multiple WNIs embedded in smartphones.

11.4 Getting a little help from friends

Sometimes everyone needs a bit of help from others. Smartphones are no different in this sense. There are situations where the phone could ask another device to do some work on its behalf, which would save energy by reducing the application-generated workload to the underlying hardware.

Let us take sudoku solver as a simple example: An application needs to compute solutions to a specific sudoku and present it to the user. It has two choices to accomplish this task. First is to use its own CPU to calculate the answer and the second is to ship the characteristics of the sudoku, that is, the numbers and where they are located, to a cloud that provides such a computing service. In the latter approach, the smartphone's own CPU is hardly used at all, making it potentially a lot more energy efficient, whereas in the former one it needs to do all the work. Such a computing paradigm is often called *computation offloading* and we have devoted Chapter 14 this topic.

A careful reader would have noticed a potential problem in the above example. While such computation offloading reduces the computational work, it requires some wireless communication to ship the required code and data to the cloud and the results back to the phone, which is unnecessary in the local computing approach. Indeed, the biggest challenge for computation offloading is to be able to accurately evaluate this tradeoff between computational and communication energy to make decisions on whether a given task should be offloaded. Notice that because of the context dependency of the energy consumption of wireless communication, this tradeoff must be evaluated and offloading decisions made dynamically at runtime.

Communication offloading is a related concept. It refers to reducing the communication workload or, more specifically, the energy expenditure of the smartphone caused by wireless communication by remotely executing some tasks that require wireless communication. In fact, there are two different factors that contribute to the energy efficiency: reducing the amount of traffic from/to the smartphone and optimizing the traffic patterns (we learn more about energy-efficient traffic patterns in Chapter 12). It is important not to confuse this concept with traffic offloading which refers to vertical handovers, typically so that some traffic is offloaded from a cellular access network to a Wi-Fi access network (Section 13.6).

There are several different systems proposed for offloading program execution from smartphones. In Chapter 14, in addition to delving into the details of the basic principles, we look at the different factors that should be taken into account when designing such a system.

11.5 Summary

In this chapter, we presented an overview of the energy-optimization principles and techniques that can be applied as an additional optimization layer on top of the individual hardware-level power-management and optimization mechanisms. The key observations are the following:

- Additional opportunities to save smartphone energy arise from three facts: 1. the different hardware components are not perfectly power proportional, 2. the energy utility of the different subsystems is context dependent, and 3. the underlying hardware power-management mechanisms rarely understand anything about the applications being used.
- There are two fundamental ways to achieve the additional energy savings: 1. reduce the energy expenditure of doing a unit of work, that is improve the energy utility,

which is the target of solutions that exploit the power disproportionality and context dependency of energy consumption, and 2. reduce the amount of work, which is the goal of solutions that change the application logic.

- Traffic scheduling is an important class of mechanisms to improve the energy utility of data transmission.
- Taking advantage of the many wireless technologies deployed in the smartphones is another effective way to improve the energy utility of data transmission.
- Mobile cloud offloading can be used to reduce the amount of work that the smartphone needs to perform. However, the offloading systems must carefully evaluate the tradeoff between reduced computational work vs. increased communication work.

12 Traffic scheduling

Scheduling means timing actions in a proper way. Mobile systems undergo such decisions continuously. For example, when to transmit the next packet or when to switch to executing another process. Traditionally, computing systems have strived for maximum performance or throughput of a system when designing different kinds of scheduling mechanisms. Considering energy consumption provides a different perspective.

12.1 How scheduling saves energy

The first motivation for considering scheduling for the sake of energy efficiency is that smartphones and their subsystems are not perfectly power proportional, sometimes very far from it. In a perfectly power-proportional system the power draw would linearly scale with the workload throughout the entire range of possible workloads.

To understand the benefits of energy-aware scheduling, let us consider an example of wireless communication with a smartphone. Wireless communication typically provides a better energy utility with a higher data rate. In other words, a higher data rate leads to fewer joules consumed per bit transmitted or received. Because of this, it would be more energy efficient to schedule multiple lower rate data flows in a maximally overlapping manner than to schedule them individually.

Generally speaking, it is beneficial to exercise the above kind of "race-to-sleep" scheduling policy on subsystems whose power consumption scales sublinearly with the workload. WNIs exhibit such behavior because of their inherent tail energy, as shown by Pathak et al. [1]. The main challenge is to find the optimal scheduling algorithm that takes the possible constraints of the particular subsystem into account. Scheduling algorithms in general have been studied for several decades.

There is also another motivation for using scheduling to improve the energy efficiency of smartphones, especially when it comes to wireless communication. Energy consumption is often context dependent and the context varies with time. The context that affects the energy consumption includes wireless network conditions: signal-to-noise ratio (SNR), network congestion (which in turn are influenced by many factors), available wireless networks, and even temperature. The main challenge here is to predict those conditions that affect the scheduling decisions. In some cases it is possible, while in others the conditions are simply too unpredictable.

We focus in this chapter specifically on traffic scheduling. We look at different ways of scheduling traffic and how that can help to reduce the energy consumption of smartphones. We summarize the different methods in Table 12.1 and discuss each method in detail in the following sections.

12.2 Shaping traffic to “race to sleep”

The first technique that we study is traffic shaping. In a nutshell, it means buffering packets that a server has sent at a constant rate for some time and then forwarding these packets in a burst. To understand when and why that saves energy, we first discuss the available bandwidth of an end-to-end network path and, specifically, the amount of available bandwidth in different segments of such a path. Then we describe the actual technique and make some model-based calculations of its effectiveness in different situations. Finally, we conclude with a short literature survey to give examples of proposed systems that rely on traffic shaping to save energy.

12.2.1 Mismatch between throughput and available bandwidth

As we discussed in Chapter 7, all the radios integrated in smartphones are power managed in one way or another. The basic mechanism is to switch a radio to low-power sleep mode by powering off some of the circuitry when no data is being transmitted or received. Furthermore, we know that the energy utility, namely bits transmitted per joule spent, increases with throughput, regardless of which kind of wireless communication technology is used. Hence, energy can be saved if data communications can be scheduled in such a way that the throughput of individual transfers and the sleep time of the radio are increased compared to “normal” operation.

When is such scheduling possible? Intuitively, some spare bandwidth must exist at the radio access network. Otherwise, there is no way to increase the throughput over the last hop link (or first hop in case of upstream data transmission). Typically, a bulk data transfer using TCP would try to use all the available bandwidth leaving no opportunities for such scheduling. However, there are two situations where a discrepancy between the throughput and available bandwidth arises. We illustrate those situations in Figure 12.1. The numbers in the drawing correspond to the available bandwidth for the specific user in the client side (left) and bandwidth use at the server side. They are only indicative and can vary considerably case by case.[1]

In the first case, the application intentionally uses less bandwidth than is available on the end-to-end path. For example, multimedia streaming applications fall into this category when the server delivers content to the client at a rate which is relative to

[1] At the time of writing this book, the latest smartphones in the market were category 3 LTE devices which could provide downlink data rates up to 100 Mbit/s using 2 x 2 MIMO. HSPA+ could offer up to 42 Mbit/s data rates. As for Wi-Fi, the latest phones were already 802.11ac compliant but could only achieve data rates up to 430 Mbit/s using a single spatial stream with an 80 MHz channel.

Table 12.1. Ways to save energy through scheduling

Method	How it saves energy	Example solutions	Pros	Cons
Shape traffic to bursts	Reduces total amount of tail energy	*Catnap:* Application agnostic middlebox sitting at wireless AP rescheduling and batching transfers.	Works in principle with all kinds of application.	Must be deployed separately at each wireless AP.
		Estreamer: Multimedia stream delivery system that transforms CBR stream into bursts of energy optimal size.	Can be deployed as proxy as well as server component. Finds energy optimal traffic profile for a given client.	Targeted only for multimedia streaming, does not handle other traffic.
Synchronize background traffic	Reduces total amount of tail energy	*AdCache:* Uses prefetching and caching to deliver ads to mobile applications.	In addition to energy, reduces traffic overhead by mitigating redundant transfers.	Undermines real-time auctions of mobile “ad space”. Requires applications to explicitly use the framework.
		NSRM: Uses gating to intercept app requests for later bundled transmission.	No need for explicit support from applications, get benefits right away with existing applications.	Some applications may behave unexpectedly as they are unaware of the mechanism.

Context-aware scheduling	Improves energy use (fewer joules per bit)	*Bartendr:* Predicts upcoming signal strength using previously learned “signal tracks” in orderto schedule transfers during times of strong signal.	Learns context almost for free (energy-wise).	Signal strength alone does not determine the resulting energy utility. System also needs training.
		CasCap: Uses crowd-sourced context monitoring and provides scheduling as a cloud-based service.	Crowd-sourcing can yield a large and continuously updated contextual information base.	Crowd-sourcing requires volunteering participants. Sharing the contextual information requires communication with cloud.
		LoadSense: Estimates load in the cellular network to schedule background transfers when load is small.	There is evidence that network load correlates with energy use better than signal strength (Bartendr).	Efficient usage requires implementation within modem to avoid constantly high CPU power.
Cooperative scheduling	Improves energy use (reduces transmission time)	*NapMan*: Uses different PSM beaconing schedule for each client to avoid simultaneous wake-ups and requests for queued packets.	Requires no change to 802.11 standard protocols.	Requires virtual AP support. Does not mitigate inter-AP contention.
		Sleep Well: Solves inter-AP contention so that each AP that hears each other adapts its PSM schedule to not coincide.	Requires no change to 802.11 standard protocols.	If a Wi-Fi deployment uses non-overlapping channels for adjacent APs, this solution is unnecessary.
		Scheduled PSM: Use TDM scheduling.	Nearly theoretically optimal solution.	Requires changes to clients and APs.

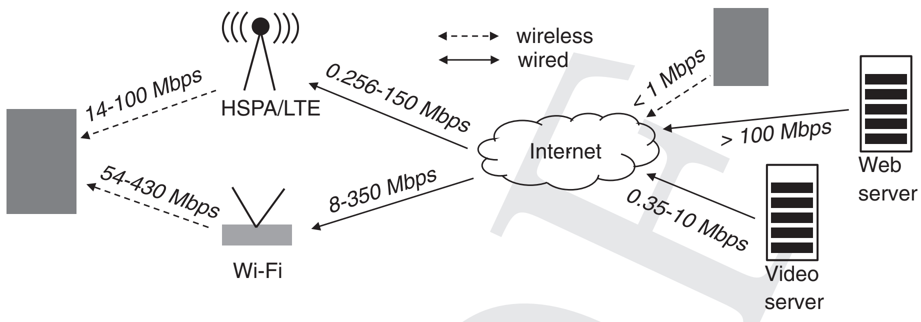


Figure 12.1 Available and used bandwidth depends on networks and applications

the encoding rate, the rate at which the client player consumes the content. This situation is illustrated in the figure with a video server that serves the client at 1.5 to 2 times the encoding rate. Such behavior is common among the popular on-demand video streaming services, such as YouTube. The absolute rate of course depends on the video encoding rate. Also some other applications throttle their transmission rate, for example P2P content distribution applications do it to avoid saturating their uplink bandwidth because that could severely hurt the performance of other applications. In contrast, a web server would try to send the web page contents to the client as fast as possible to minimize the transaction latency and maximize the QoE.

The second situation where the discrepancy arises is when the amount of available bandwidth on an end-to-end path from a server to a wireless client is less than the available bandwidth of the wireless access link. A typical scenario is a home or office environment where Wi-Fi is the last hop. As the figure suggests, Wi-Fi nowadays offers data rates of several hundred megabits (even up to gigabit rates with the latest 802.11ac), whereas the data rates of home internet subscriptions, for instance, offered by ISPs over ADSL or cable are usually much smaller. We note that this mismatch also exists with cellular access networks where the peak downlink rates usually greatly exceed the typical subscription rates of the clients (indicated by the rates between internet and radio access in the figure). So, at least in principle, such traffic-scheduling opportunities also exist with cellular access, although in practice, such solutions are much simpler to deploy with Wi-Fi access.

12.2.2 How traffic shaping works

In the above-mentioned two cases, using the "race-to-sleep" strategy can save energy. The trick is to schedule the data transmissions in such a way that as much of the wireless access link bandwidth as possible is used at a time, which increases the total amount of sleep time and reduces the overall energy consumption. In practice, it means delaying the transfer of some of the packets to batch them into high-speed bursts. This kind of scheduling is sometimes also called traffic shaping.

In the case where the application intentionally throttles the transmission rate, traffic shaping at any vantage point along the path can save energy, for example at the server,

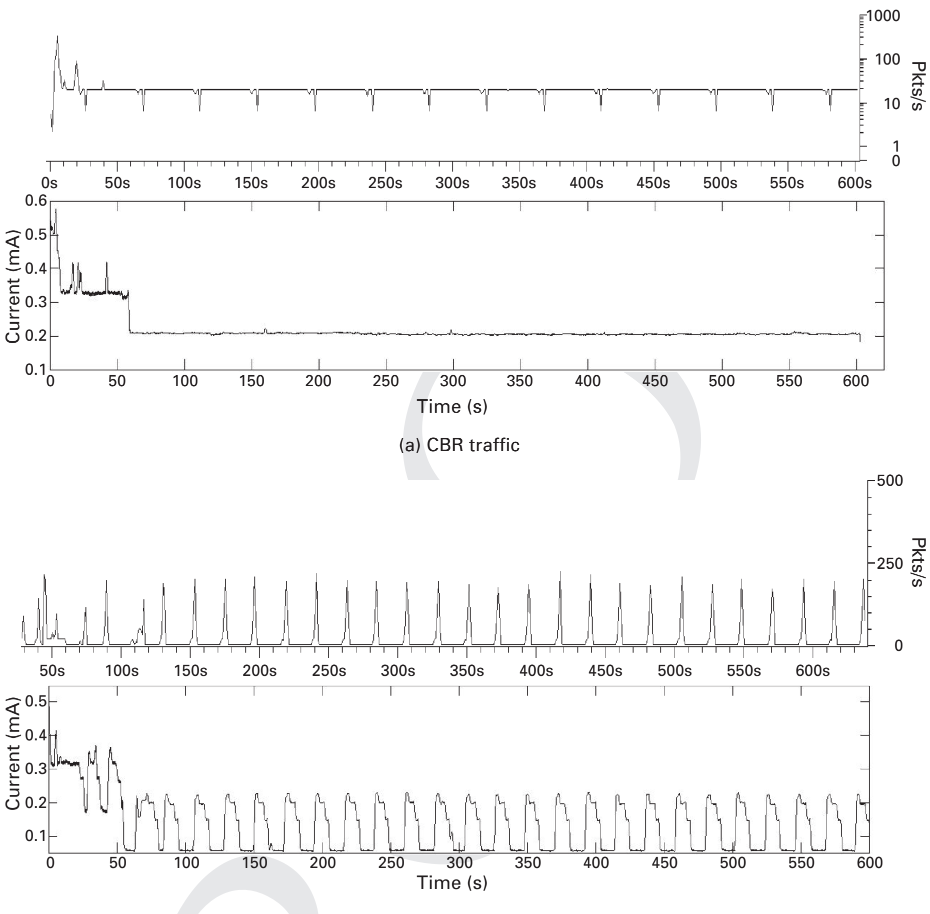


Figure 12.2 Comparison of power consumption of audio streaming with different traffic shapes

proxy, or wireless access point. On the other hand, if the application uses all the bandwidth available on the network path and traffic shaping is used to exploit the bandwidth discrepancy between the different segments of the network path, the traffic shaping must occur after the bottleneck link to have any impact. For example, the Wi-Fi access point in Figure 12.1 would be an ideal location.

Because traffic shaping requires delaying some of the packets and delay directly impacts the QoE of certain applications, it must be applied carefully. Some applications do not tolerate extra delay well. Inelastic applications, such as VoIP or video calls, belong to this category. Also, delaying packets of transfers within web transactions must be subtle. It is all right to delay some packets as long as the whole transaction is not delayed, which would be visible to the user as a degraded QoE.

Figure 12.2 illustrates how this behavior leads to lower energy consumption when playing an audio stream with a smartphone using HSPA connectivity. The figure shows traffic and power traces measured in two different cases. In both, the same audio stream was played with the same smartphone, but in one case the constant bit-rate (CBR) audio stream is shaped into bursts by a proxy before delivering it to the phone (Figure 12.2(b)), whereas in the other case the CBR stream is directly received from the server (Figure 12.2(a)). It is easy to see how the bursty traffic patterns allow the radio to catch some sleep in between receiving a batch of packets. In addition, comparing the duration of power spikes to the duration of traffic spikes, the relatively long tail energy spent after the reception of each burst can be clearly identified.

12.2.3 How much energy can be saved?

The exact amount of energy saved through traffic shaping depends on a number of factors. The discrepancy between the bandwidth used and the bandwidth available between the traffic shaping point and the client is one of the most determinant factors. Indeed, the more speed-up is achievable for the transmission, the more the sleep time can be prolonged. In addition, the characteristics and configuration of the wireless communication technology used have an influence. Specifically, the inactivity timers that determine the amount of tail energy are important, as we just saw in the example power traces. When shaping traffic into bursts, the interval between the bursts must be longer than the inactivity timers before any energy can be saved.

To better understand the magnitude of energy savings, we next compute some results with example scenarios using simple power models for wireless communication. We assume constant power draw when the WNI is in the idle and sleep modes. In addition, HSPA is assumed to have constant power draw in the active CELL_DCH and CELL_FACH states as well as for LTE receiving and being idle in the RRC_CONNECTED mode. These assumptions are not too far from reality, since the most significant differences in power draw happen when switching from one mode to another. To characterize the power draw while receiving data using Wi-Fi, we use a simple model where the energy utility (bits transferred per joules spent) grows linearly as a function of the data rate. The model is derived from the results presented in [2]. The parameters are listed in Table 12.2 and we have derived the power values from measurements with smartphones. All active-state power values have been computed by subtracting the power drawn by the WNI when it is in inactive mode (i.e., CELL_PCH, RRC_IDLE, and Wi-Fi sleep) from the original measured total power draw. *Max delay* denotes the longest amount of time that a single packet can be delayed before being bundled into a transmission. A given value of *max delay* and the throughput of unshaped traffic together determine the amount of data to transmit in a single burst. The variables *shaped rate* and *unshaped rate* mean the throughput achievable end-to-end (all the way from server to client) and only from the traffic-shaping point to the client, respectively. Hence, the *rate factor* is always smaller than or equal to one and it is possible to save energy only if it is smaller than one.

We compute the energy consumed by receiving the content as unshaped traffic using Eq. (12.1) and the energy consumed by receiving the same content as shaped traffic using Eq. (12.2). The former case is calculated simply by multiplying the RX power

Table 12.2. Parameters used in power modeling the impact of traffic shaping

Constants	Wi-Fi	HSPA	LTE	Variables	
rx power (W)		$P_{rx} = 0.8$	$P_{rx} = 1.5$	Data rates (Mbit/s)	unshaped rate r_{us} shaped rate r_{sh} rate factor $r_f = r_{us}/r_{sh}$
Tail power(W)	$P_1 = 0.4$	$P_1 = 0.8$ $P_2 = 0.6$	$P_1 = 1.2$	Max delay (s)	T
Tail timers (s)	$T_1 = 0.2$	$T_1 = 8$ $T_2 = 3$	$T_1 = 10$	Amont of data (MB)	s

with the time it takes to receive the content at the delivery rate of the server. In the latter case, we calculate the energy using the bulk transfer rate from the vantage point of traffic shaping to the client. In addition, we take into account the tail energy which is spent in between receiving the bundles of content. Note that because of the periodicity of the traffic shaping, we only need to compute the energy spent over a single period of traffic shaping (max delay) to obtain comparable results.

$$E_{unshaped} = \frac{s}{r_{us}} P_{rx}(r_{us}) \tag{12.1}$$

$$E_{shaped} = \frac{s}{r_{sh}} P_{rx}(r_{sh}) + E_{tail}(s) \tag{12.2}$$

To compute the tail energy, we need to consider different cases depending on the inactivity timers used by the WNI and their durations compared to the amount of idle time in between receiving a bundle. Indeed, different amounts of idle time may allow a different number of timers to expire which means that the radio can spend time in different power states depending on the access network technology type and timer values. We write out the tail energy computation in Eq. (12.3). We include two inactivity timers, T_1 and T_2, which correspond to the case of 3G. If the WNI only uses one inactivity timer, as is the case for Wi-Fi and LTE, we simply set $T_2 = 0$ and discard the middle equation which is no longer valid.

$$\begin{aligned} 0 < \frac{s}{r_{us}} - \frac{s}{r_{sh}} < T_1 &: E_{tail}(s) = P_1 \left(\frac{s}{r_{us}} - \frac{s}{r_{sh}} \right) \\ T_1 < \frac{s}{r_{us}} - \frac{s}{r_{sh}} < T_1 + T_2 &: E_{tail}(s) = P_1 T_1 - P_2 T_1 + P_2 \left(\frac{s}{r_{us}} - \frac{s}{r_{sh}} \right) \\ T_1 + T_2 < \frac{s}{r_{us}} - \frac{s}{r_{sh}} &: E_{tail}(s) = P_1 T_1 + P_2 T_2 \end{aligned} \tag{12.3}$$

We plot the resulting average power draw (simply divided energy consumed by max delay) when shaping traffic in Figure 12.3. The results are relative to the case when no shaping is done, that is the rate factor is equal to one. In calculating the results, we

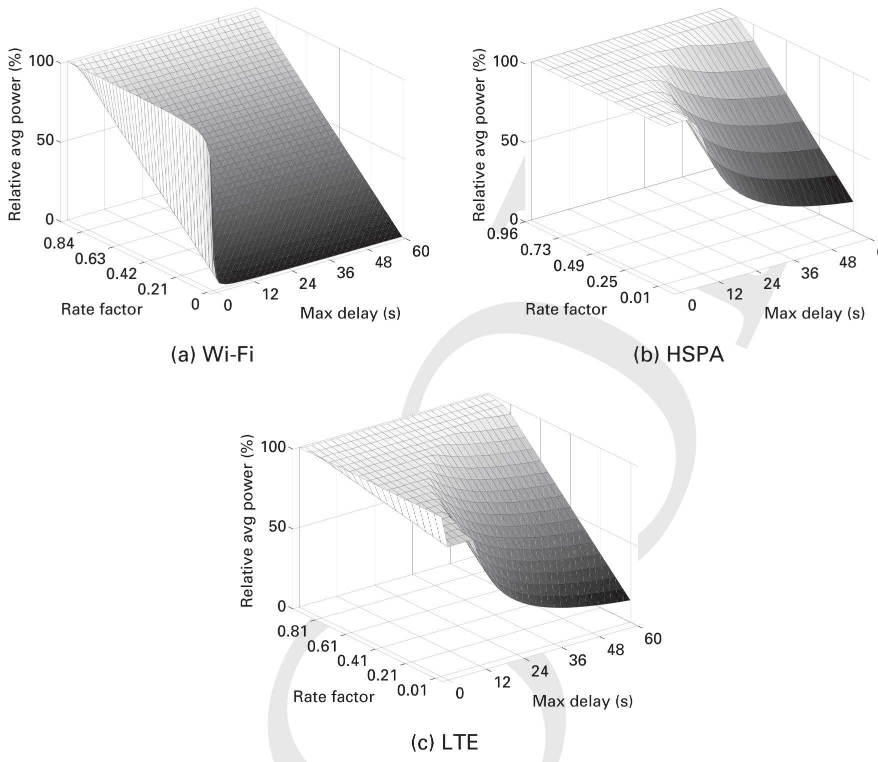


Figure 12.3 Quantifying the energy-saving potential of traffic shaping

used for the shaped rates the maximum data rates supported by modern smartphones today with the different technologies that we mentioned earlier (430 Mbit/s for Wi-Fi, 42 Mbit/s for HSPA+, 100 Mbit/s for LTE). The first thing to note is that the Wi-Fi results look quite a bit different from the HSPA and LTE results which have a similar shape. The reason is the major difference in the tail energy: the Wi-Fi tail is amortized already by adding just a little bit of delay whereas such a delay delivers no visible energy savings for the cellular network technologies. However, if traffic can be shaped with a relatively long delay, such as a few tens of seconds, the energy-saving potential is also considerable when using the cellular access network.

Supporting longer values of max delay for traffic requires first of all that the traffic is such that the application generating the traffic does not become unusable but also that the traffic shaper has enough buffering capacity. The required amount of buffer space depends on the data rate of the unshaped traffic and max delay. To understand how much is required, we plot example results for Wi-Fi and LTE in Figure 12.4. The shapes of the surface plots are similar but the range of values is different. Wi-Fi, being able to offer four times higher data rates, has equivalently larger buffer space requirements for the same rate factor. However, the required buffer space for the same absolute values of unshaped data rate and max delay are of course equal.

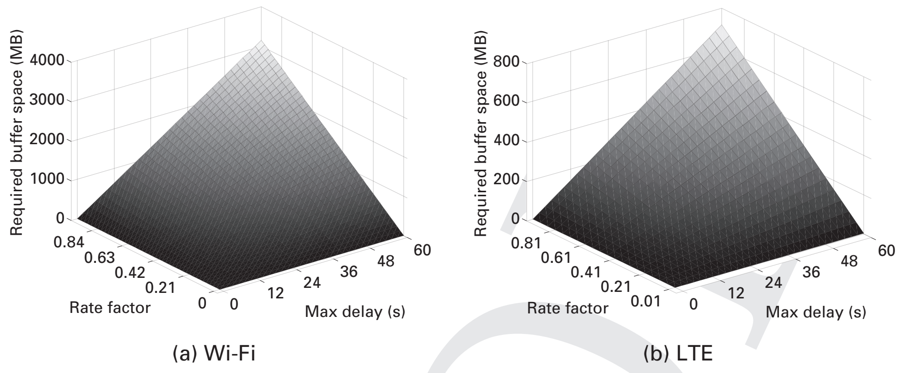


Figure 12.4 Traffic shaping requires differing amounts of buffer space to hold delayed packets depending on the difference in shaped and unshaped data rates and the amount of delay

How to translate these results into practice? Taking video streaming as an example, a relatively high-quality video with 2 Mbit/s average encoding rate gives rate factors of about 0.005 and 0.02 with Wi-Fi and LTE, respectively, which puts them right into the lower end of the rate factor scale shown in the figures, delivering potentially large energy savings with relatively little buffer space required. Let us consider bulk file transfer as another example. If we assume that the user's home internet connection policed to the subscription rate forms the bottleneck on the network path and that the user has a fairly high-end subscription of 100 Mbit/s, the rate factor becomes 0.23 for Wi-Fi. Such a setup still has potential to cut down the average power consumption by more than half.

The energy consumption numbers presented here should be, as in any context, taken with a pinch of salt; they always depend on the exact hardware used. Furthermore, wireless access is often shared among multiple users in which case the peak downlink rates may not be achieved. Nevertheless, the results illustrate that such a simple mechanism can be very powerful in saving energy.

12.2.4 Example solutions

There are numerous kinds of traffic-shaping solution proposed in the literature of which we briefly introduce two here. EStreamer [3] is a multimedia stream delivery system that can be deployed as a proxy or integrated into a streaming server. It shapes the CBR traffic into bursts before delivering it to the mobile client. We revisit EStreamer in a later section that focuses specifically on video streaming.

In [4], the authors propose using home routers as proxies for mobile clients in a BitTorrent content-sharing network. These proxies can deliver content energy-efficiently to the mobile devices by making use of the above-mentioned discrepancy in bandwidth usage and availability. Their measurement results suggest that, compared to downloading the content directly to the phone with a native BitTorrent client, the proxy setup can make energy saving of 40–50%.

Catnap [5] aims to serve a similar purpose but in a more generic sense. It is essentially an application-independent middlebox that sits at the wireless access point and schedules delivery of data blocks, a.k.a. Application Data Units (ADUs), to the mobile devices. The scheduler delays transmission of the first packets just enough to deliver a particular ADU over the wireless access link without adding to the total transaction delay. If the client indicates its willingness to accept additional delay, it can also run in batch mode where it further delays individual packet transmissions to send an even larger bunch of packets at a time.

12.3 Scheduling background traffic

Big data is a challenge for many computing systems today, but for mobile communication systems, small data is also a challenge. To understand why this is the case, we next look at smartphone background transfers.

12.3.1 Problematic small transfers

Background traffic consists of typically small data transfers which happen either completely in the background, that is when the smartphone screen is off and the user is not actively doing anything with it, or as a side product of using a particular application, such as gaming and video streaming. In both cases the background traffic does not contribute directly to the main purpose of the application. Common examples of background traffic include advertisement or analytics traffic, and synchronization of email, calendar, and other such applications. There is typically no synchronization between different kinds of background traffic, and applications independently establish data flows for such traffic.

From the energy-consumption perspective, these small data transfers are harmful if they emerge in an isolated unsynchronized manner. In such cases, the radio is powered on only because of that traffic. The long tail timers together with the high-power draw in the active modes of cellular network interfaces make the situation especially bad. For example, a large portion of the mobile advertisement traffic by default emerges in an isolated manner and causes significant smartphone energy expenditure.

Such isolated background traffic causes an additional kind of harm for cellular network operators. The small sporadic transmissions generate a large amount of state transitions by the RRC protocol. For example, the RRC protocol used in the HSPA network moves from the CELL_PCH state to the CELL_DCH state and back to the CELL_PCH state again via the CELL_FACH state just for the sake of transmitting or receiving a few kilobytes of data. The problem lies in the fact that each transition generates a number of signaling messages exchanged between the phone and the network. When all the phones connected to the network go through such state transitions frequently, the operator faces a signaling storm that has potential to degrade the service [6]. Hence, there are also incentives from the operator side to minimize the number and frequency of isolated background transactions.

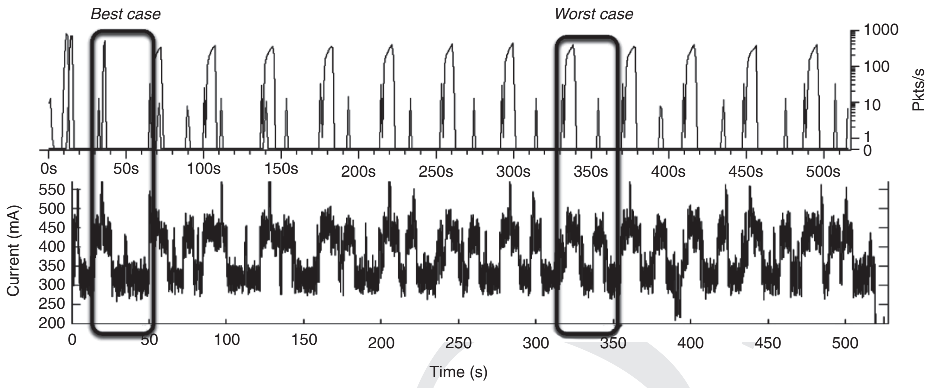


Figure 12.5 Example traffic and power traces illustrating the detrimental impact of background traffic on energy consumption

12.3.2 How much energy is wasted?

Figure 12.5 shows an example of detrimental background transfers that are interspersed with the actual content download periods in an unfortunate manner. Larger peaks in traffic are the actual content and small peaks correspond to background traffic (note the logarithmic scale). The smartphone used an HSPA network. In this particular case, the content received is video stream and the background transfers were identified as periodic updates of viewing progress sent to the video service provider's statistics server. Each individual chunk of content downloaded is in fact of the same size. We calculated the energy consumption for the best and worst case chunk downloads for similar traces. These are also highlighted for this example in the figure and the net effect of the background traffic was up to a 30% increase in energy consumption.

Let us now look at the problem more systematically. The underlying issue is that the energy efficiency of small transfers is very poor, especially when using a cellular access network, and therefore isolated small transfers cause a disproportionally large drain on the battery. To understand how poor, we next model the energy utility as a function of transfer size. To quantify this effect, we adapt the simple models we used in the previous section when analyzing the effectiveness of traffic shaping for the case of small transfers. The difference is that this time we account for the whole tail energy for each transfer. In other words, we always consider the bottom case in Eq. (12.3). Another thing that we do differently this time is to add TCP behavior into the model. TCP, with its slow start mechanism, substantially slows down small transfers, which is why it is necessary to take this into account [7]. We do not include packet loss in the model because that would render the model unnecessarily much more complex.

In the absence of packet loss, TCP remains in slow start mode during the entire small transfer. We further assume that the congestion window size has been reset to its initial value before the transmission starts. We approximate the TCP's congestion window growth using geometric progression, where subsequent rounds of transmission of a congestion window's worth of packets follow one another. In between these

Table 12.3. Additional parameters used in energy modeling of small transfers

Parameter	Value range	Description
t_{rtt}	50/100/80 ms	Round-trip time (RTT) for Wi-Fi/HSPA/LTE
w_1	3	Initial congestion window (cwnd) size
γ	1.5	cwnd increase rate per RTT
C	100/21/100 Mbps	TCP bulk transfer capacity for Wi-Fi/HSPA/LTE
S	0–1 MB	Amount of received data
MSS	1460 B	Maximum segment size

transmissions, the TCP sender must wait for incoming acknowledgments. We compute the resulting transmission delay as the number of these rounds times the round-trip time (RTT). We use the power constants in Table 12.2 and the new parameters described in Table 12.3. TCP bulk download rates have been chosen according to typical peak rates achievable with today's smartphones and subscriptions. RTT values represent relatively short-distance communication, such as within a continent, and an expected amount of added delay by the HSPA (+50 ms) and LTE (+30 ms) access networks have been added to their RTTs compared to Wi-Fi access.

We consider here specifically the case of receiving data but the same models can be applied for data transmission by replacing the receive power with the transmit power and updating the transfer rates. If the amount of data received is large enough to completely saturate the path, that is the congestion window grows larger than the bandwidth delay product of the path, the TCP sender can transmit packets continuously and it does not need to wait for incoming acknowledgments. Therefore, we divide the modeling into two separate cases depending on whether the path become saturated or not. This condition is expressed in Eq. (12.4) and is derived from the geometric progression by solving the number of rounds required to transmit all the packets (left side of inequality) and to grow the window beyond the bandwidth delay product of the path (right side of inequality). As a side note, it is straightforward to solve the equation for the amount of data (*S*) and check when this condition applies. With the values for RTT and TCP bulk transfer capacity that we use, the path becomes saturated only if the amount of data is greater than 1.9 MB, 780 kB, and 2 MB for Wi-Fi, HSPA, and LTE, respectively.

$$\frac{S(\gamma-1)}{MSSw_1}+1 \le \frac{Ct_{rtt}\gamma}{MSSw_1} \tag{12.4}$$

We then obtain Eq. (12.5) for the time it takes to receive the data depending on whether the path become saturated during the transmission.

$$\text{If Eq. (12.4) true: } T_{rx} = log_{\gamma}\left(\frac{S(\gamma-1)}{MSSw_1}+1\right)t_{rtt} \tag{12.5}$$

$$\text{If Eq. (12.4) false: } T_{rx} = \left[log_{\gamma}\left(\frac{Ct_{rtt}}{MSSw_1}\right)+1\right]t_{rtt} + \frac{S-\frac{MSSw_1-Ct_{rtt}\gamma}{1-\gamma}}{C}$$

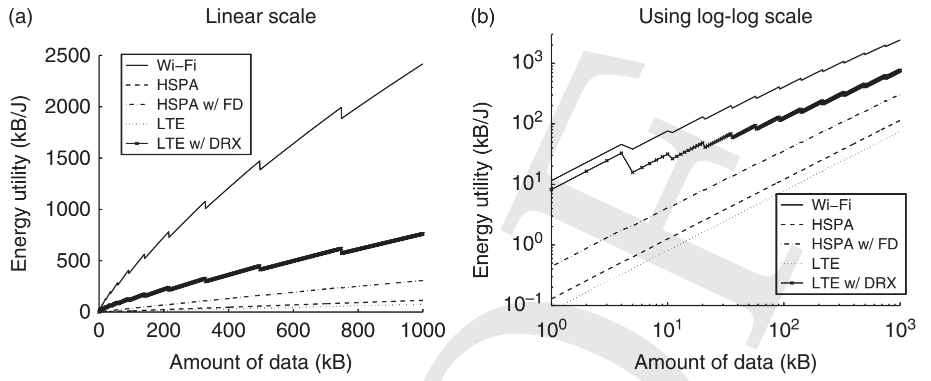


Figure 12.6 Energy utility of data transfer as a function of transfer size shows that small transfers are poison to battery life

Using this time, we compute the energy spent on receiving the data according to Eq. (12.6). We can safely ignore the impact of transmitting TCP acknowledgments because of their small size and the fact that they interleave with the transmission of the data packets. With the RTT values we used, the inactivity timers of the WNI power management do not trigger, so we can also ignore any effects due to WNI transitioning into sleep mode in between receiving flights of packets.

$$E_{rx} = \frac{S}{C}P_{tx} + \left(T_{rx} - \frac{S}{C}\right)P_{idle} + E_{tail} \tag{12.6}$$

Figure 12.6 plots the resulting energy utility as a function of the amount of data received. We used a tail timer of three seconds with HSPA FD and an activation timer of 100 ms for LTE DRX. The sawtooth shape of the curves is caused by the behavior of TCP, which uses the congestion window to control how many packets can be sent at a time. There is a drop in energy utility each time the amount of data to transfer grows beyond the size of the congestion window during the last round, because then the leftover packet must be sent in the next round, which delays the completion of the transfer by roughly one RTT. It is clear that the energy utility degrades quickly when the amount of data become small. The plot on the left shows that there is a big difference between the different technologies: Wi-Fi energy utility has a very rapidly increasing trend and LTE with DRX support also has a clearly higher slope than the plain LTE and HSPA curves. The reason is again the differences in the magnitude of the tail energy. When using Wi-Fi and LTE with DRX enabled, the tail energy is amortized with a smaller amount of data transfer, while the long tail of HSPA and LTE without DRX dominates the energy consumption even with 1 MB downloads.

The right-hand-side plot shows the same results on a log-log scale. We can see first of all that there is a difference of two orders of magnitude in the energy utility with the shortest tail technology compared to the longest tail technology for very small (single packet) transfers. Furthermore, the decrease in energy utility is from two to three orders of magnitude when comparing downloads of 1 MB to 1 KB.

In the light of the above analysis, it is not so surprising that the small isolated background transfers have caught the attention of the research community and the industry. As a result, different kinds of solution to mitigate their effect have been developed and we look at these next.

12.3.3 Mitigating the energy overhead of background transfers

How to deal with the isolated background transfers? There are in principle three ways to improve the energy efficiency of these transfers:

1. *Schedule the background transfers so that they overlap with other foreground transfers.* This solution improves the energy utility of the background transfers because they are merged together with foreground transfers that would in any case take place and consume energy. One clever example of this approach is to schedule data transfers with no tight timing constraints to happen during phone calls [8].
2. *Bundle several small background transactions into larger batches*. This option is still available even when the first solution cannot be applied due to lack of foreground traffic. Energy utility improves as a function of data size as we saw earlier.
3. *Reduce the amount of background traffic.* This solution is sometimes available but it depends on the applications used. It must be applied per application, whereas the above two can be applied in application-agnostic ways.

We next describe a couple of example solutions that implement one or several of these approaches.

Synchronizing background transfers with network socket request manager

Qualcomm has developed a solution called NSRM: Network Socket Request Manager, which at the time of writing was claimed to be commercially available soon [9].

NSRM is an example solution that batches small transfers together. It uses a gating concept, where background application requests are intercepted by the NSRM and delayed until the gate opens. The gate is opened either periodically or when the user activates the screen. Figure 12.7 illustrates how the gating works. There are two parameters: T_{open} defines for how long the gate remains open once it has been opened and T_{period} controls how frequently the gate is opened.

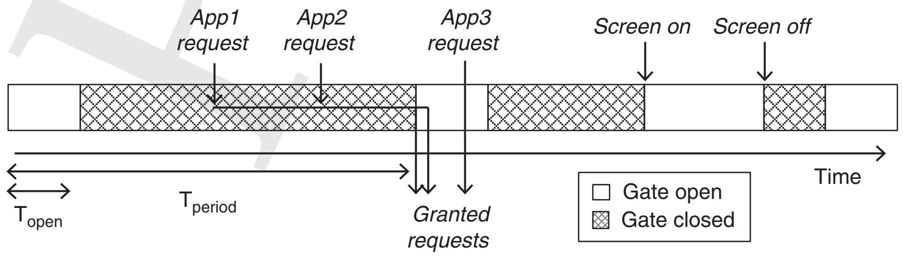


Figure 12.7 NSRM gating mechanism

An attractive property of this solution is that it works with existing applications without requiring modifications to them. However, since the application developers are potentially unaware of such a mechanism, it is unclear whether some applications that have background tasks would behave in an unexpected manner in the presence of NSRM. For example, the application developer may have integrated application-level techniques to improve robustness, such as retransmitting messages in case they are not delivered in time.

Box 12.1 HTML5 WebSockets and energy

HTML5 brings a number of convenient mechanisms that web developers can use. One of these is WebSockets that provide a means for a server to push content to the client. The underlying mechanism of a WebSocket is simply a TCP connection that is continuously kept alive using heartbeat messages and it allows for two-way, full-duplex communication. It is an improvement over techniques that use polling where the client always needs to make a request before the server can send new content to it.

WebSockets generate one form of background traffic that can be equally detrimental to the battery life of a smartphone as any other background traffic. This happens when a WebSocket is kept alive and idle for an extended period of time. Although WebSockets reduce the header overhead significantly compared to basic HTTP, it is often more important from the energy-consumption perspective to consider the timing of the transfers and not the total amount of data. Therefore, the frequency of the heartbeat messages used to keep the socket alive matters.

One of the commonly used libraries to implement WebSocket communication is Socket.io. It uses a 25 s heartbeat interval by default. Obviously, by doubling this interval, the energy consumption caused by the exchange of heartbeat messages can be cut in half. A measurement study presented in [10] confirms that substantial energy savings can be achieved when using an access technology that exhibits a large tail energy, such as HSPA, whereas with Wi-Fi access the energy drain due to these messages is almost negligible because of its small tail energy. However, with certain devices and browsers, the WebSocket is automatically closed due to unknown reasons, most likely another timer within a browser or the mobile OS, after less than a minute if the heartbeat is configured to a value longer than that. This unfortunate behavior underlines the need for further work in refining the support for Websockets with long heartbeat intervals.

Dealing with mobile advertisements

Mobile advertising has become an important business as the capabilities of smartphones has improved and their usage increased. The usual way advertisements are delivered to smartphones is through ad networks [11]. They distribute ads created by ad publishers on behalf of ad agencies that brands have commissioned to make a mobile advertising campaign. Ad networks provide a software development kit (SDK), so that advertisements can be embedeed into native applications in an easy way. The ads are typically either banners shown persistently at the top or bottom of the page or full-screen ads shown in between transitioning from one activity to another in an application.

Some measurement studies have shown that up to 75% of the energy consumed by an applicaiton can be caused by advertising [1]. A very large part of this energy is consumed by the traffic generated by fetching the ads to the applications from the ad network in real time, which often causes isolated HTTP transactions that have a very poor energy utility.

Fortunately, it is possible to combat this energy inefficiency by pre-fetching ads and caching and serving them locally, which is what AdCache does [12]. So, in a sense, it implements the approach where many background transactions are bundled together but does it in a proactive manner instead of delaying transactions. The evaluation results of AdCache suggest that such a solution can help to reduce the energy consumption of certain applications by upto a half.

The downside of pre-fetching and caching ads is that the ad networks can no longer deliver ads in real time, strictly speaking. This aspect becomes an issue with advertising systems that sell "ad space" through real-time auctions, which is no longer possible if ads need to be pre-fetched. A possible solution to this dilemma is to also do the auctioning proactively. In other words, the system can change so that it tries to predict the ad slots ahead of time for auctioning. Obviously, such predicted information is unreliable by nature and sometimes too many or too few slots may be sold. Nevertheless, it seems that such prediction can achieve pretty good accuracy enabling the ad network to maintain a tolerable level of overselling ads [13].

12.4 Context-aware scheduling

12.4.1 How context impacts the energy consumption

We have already stated at the beginning of this chapter that context can significantly influence the energy consumption of smartphones. But what does context mean here? Context really boils down to two characteristics of wireless connectivity: type and quality. Figure 12.8 illustrates the context that impacts the energy consumption of wireless data transmission with smartphones.

The type of wireless network technology being used determines its energy-consumption characteristics. Today's smartphones allow seamless switching of the network connection between cellular and Wi-Fi networks when the opportunity arises. These two wireless networks have significantly different characteristics in terms of energy consumption as we learned in Chapter 7. Hence, in addition to having more bandwidth available, such opportunistic switching also can save energy. We defer detailed discussion of this topic until Chapter 13.

Impact of signal strength

The quality of wireless network access varies over time. These temporal variations can also have a major impact on the energy consumption of data communication. In this

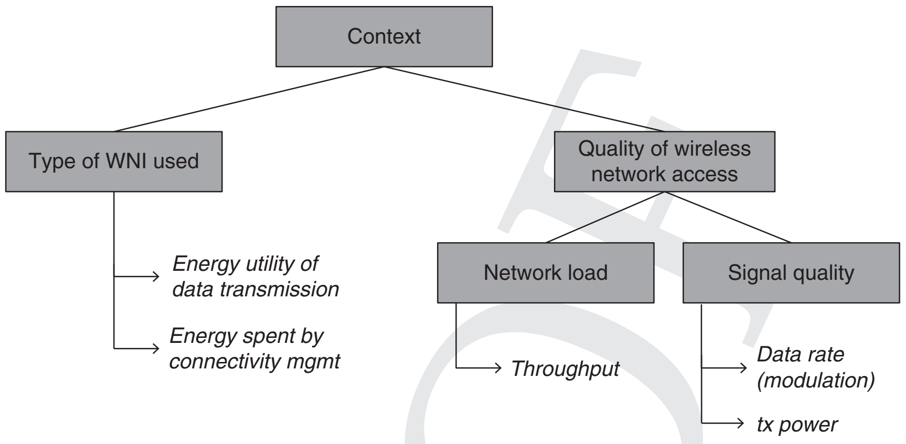

Context
Type of WNI used
Quality of wireless network access
Energy utility of data transmission
Energy spent by connectivity mgmt
Network load
Signal quality
Throughput
Data rate (modulation)
tx power

Figure 12.8 Context, such as throughput and data rate, directly impacts the energy utility

section, we focus on these contextual changes and their impact on the energy consumption. We first need to distinguish between two different causes of wireless network access quality degradations: signal quality and network load.

For signal quality, when the smartphone is in a poor network coverage area and receives a weak signal from the base station, the first consequence is that it switches to use a higher level of power amplification for transmission in an attempt to ensure that any transmitted bits are correctly received by the base station. The higher transmit power affects the power drawn by the WNI while receiving data as well, because in that case the radio is also continuously transmitting signaling messages, such as received signal strength reports, to the base station. More data needs to be potentially retransmitted as well.

The second consequence of poor signal quality is that it typically also leads to switching to a lower over-the-air data rate which means lower energy utility. This is because many of the wireless access networks, such as cellular and Wi-Fi networks, can use several different modulation schemes and those schemes provide different tradeoffs between data rate and robustness (i.e., bit error rate for a specific SNR regime). The phone and the network can switch between the different schemes on the fly to control that tradeoff dynamically. Hence, if the signal is weak, a scheme that provides a lower data rate but more robust data transmission is selected.

We show some example results from measuring the impact of signal strength on the power consumption and throughput in Figure 12.9. The measurements were conducted in laboratory settings in a room that was completely isolated from external sources of interference using a complete 3G/LTE network dedicated to the testing. We had a control knob with which we could add a desired amount of attenuation to the signal.

The results comparing signal strength with power (Figure 12.9(a)) reveal a couple of interesting things. First of all, the measurements confirm that the power drawn by both the HSPA and the LTE smartphones increases when the wireless signal quality becomes worse. The power consumption of HSPA scales less with the signal strength than LTE,

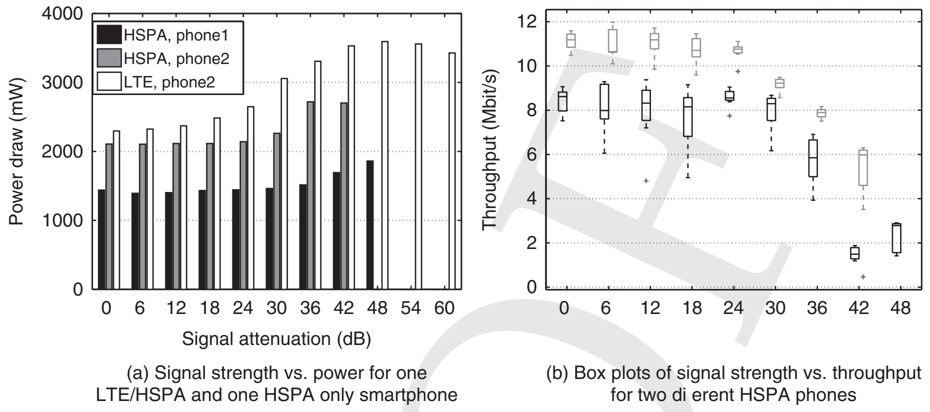
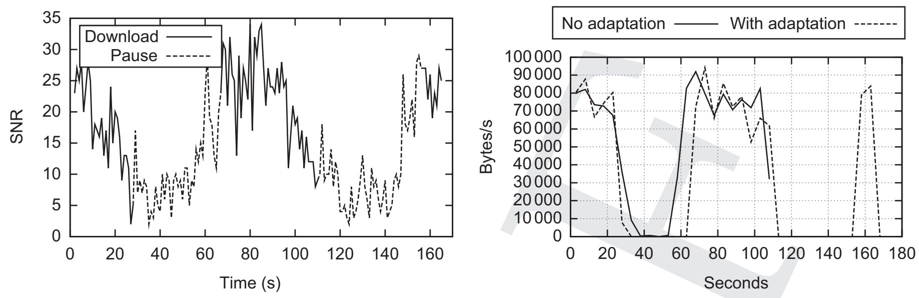


Figure 12.9 Impact of signal strength on power draw and throughput

where the power increases significantly more when the signal quality degrades. Interestingly, the LTE phone seems to hit a power ceiling at approximately 45 dB attenuation after which the power draw no longer increases. Indeed, 3.5 W is already a critically high power. It also seems that the LTE phone can operate in a smaller SNR regime than the HSPA phones. There may be many underlying reasons. The power controls of the amplifiers may differ. Also the fact that the tested HSPA uses WCDMA while LTE uses OFDMA as the multiple access protocol likely has some impact.

Figure 12.9(b) similarly confirms that the throughput of the TCP transfers is seriously degraded when the signal is attenuated. The figure only shows HSPA results but LTE behavior is comparable. Ten consecutive TCP transfers were measured in each case and the box plots reveal that there is a notable amount of variation in the results. The TCP throughput is severely influenced by IP-level packet loss (radio layer first tries to retransmit data a certain number of times) which do not occur similarly for each transfer.

Similar phenomenon can be observed from Wi-Fi networks. We collected the traces of signal and noise levels at 1 Hz while walking through a U-style corridor at a fixed speed. The SNR was calculated, as shown in Figure 12.10(a), and tagged with the position. We downloaded a 5.3 MB file from www.openss7.org using wget. We compared the performance of the downloads with and without adaptations. With the adaptation based on location information obtained from the signal strength traces is shown in Figure 12.10(a), the mobile device pauses the downloading when it enters the areas where SNR is predicted to be lower than 15 and turns off the WNI during the pause. As shown in Figure 12.10(b), the downloading was paused for 85 seconds in total. The WNI spent 83.637 seconds in receive mode, which cost 82 joules. Without adaptation, the WNI spent 85.07 seconds in receive mode and another 20.29 seconds in idle mode trying to repair the connection, which cost in total 97 joules. Hence, the downloading without adaptation consumed around 18% more energy. This result can be further improved with more accurate location estimation, because, as shown in Figure 12.10(a),



(a) Wi-Fi Signal-to-Noise ratio trace collected from a U-style corridor at 1Hz

(b) Comparison of throughput with/without adaptation.

Figure 12.10 Impact of signal strength and noise level on throughput in Wi-Fi network

the error in location prediction causes a delay in resuming the download at the end of the pauses.

Impact of network load

Network load refers to the number of clients connected to a specific wireless access network and the amount of traffic that they generate. Obviously, as the total amount of resources is limited and shared among all the users, the more competition there is, the less each one gets. Therefore, the level of network load has a direct effect on the throughput achievable by one client's transmissions. Throughput, as we have learned, directly impacts the energy utility so that the lower it is, the lower the utility is as well.

In [14], some measurement results on throughput variation are presented. The measurements on two different cellular networks, one HSPA and one LTE network, show that the variation is indeed quite large in a real network during peak hours, although such results should not be generalized. In the same work, the authors hypothetically compared downloading 5 MB of data within each 15 minutes using two different strategies: downloading right away, in other words without any attempt to avoid high-load periods, or by magically knowing the perfect moment, with respect to throughput achievable, to download within each 15-minute period. The results suggest that, in theory, it would have been possible to save almost 50% of the energy used if the network load and, consequently, the throughput could have been perfectly predicted.

12.4.2 Leveraging contextual information to save energy

Given that the energy consumption of wireless data communication is so much affected by these contextual factors, scheduling transfers at times when energy utility is at its peak can save a considerable amount of energy compared to completely context-unaware scheduling. The problem obviously lies in knowing, or rather being able to predict, when the phone is going to have good connectivity.

Predicting good connectivity with real-time measurements

There are a couple of ways to predict good connectivity. One approach is to do real-time measurements with the smartphone and try to infer the relevant contextual information from those measurements.

For example, continuous measurements of the signal strength might make it possible to predict the future values of it based on the history. The intuition behind this kind of "blind" prediction without the use of any other contextual information is that there is some structure in the signal strength variations over time which can be identified and used by using time series analysis algorithms. Some of these algorithms require offline training to fit the model to the kind of data observed. We go through a case study that uses this approach in Section 12.4.3.

A more recently introduced solution called LoadSense [14] builds on the insight that the network load correlates well with energy efficiency. A high load causes low throughput which prolongs transfers and leads to poor energy utility. The idea is to estimate the load in the cellular network by comparing the total power of the channel with the power of the pilot signal transmitted by the base station. Their ratio gives an indication of the current load in the cell because the total power includes power from the communication of other clients, whereas only the base station contributes to the pilot signal power. This power ratio is then measured in real time by each smartphone independently to predict the current load. Using this estimate, background transfers can be scheduled to happen when the load is low. A CSMA-type of protocol can be used together with the load-sensing component to avoid all sensing clients starting background transfers simultaneously.

Learning contextual information

Another way to predict good connectivity is to use the information that the quality of the wireless connectivity is often tied to the geographic location of the smartphone. Many phones together in a crowd-sourcing manner, for instance, can build an information base that maps geographic location to the quality of the wireless network access. With the help of such an information base, it is possible to predict good connectivity if only the user's geographic location can be guessed correctly ahead of time.

The key to practical location prediction is the observation that people tend to repeat certain paths in their daily life. For example, going to work and back generates the same trajectory every working day. The smartphone can learn this information and by tracking the connectivity along such trajectories makes it possible to predict when the phone will have good connectivity. Bartendr [15] is an example of a system that uses this kind of mobility prediction to schedule transfers energy efficiently.

CasCap [16] takes a crowd-sourcing approach to learn contextual knowledge. Each phone shares the information it has measured and possibly predicted along with location information to a cloud. The gathered information and knowledge can then be used by all the smartphones that participate in this crowd-sourcing to, among other things, schedule traffic when good connectivity is predicted. The challenge with such an approach is to get enough participants, because the context monitoring and sharing consume some energy and bandwidth, and you need to install the application on each phone to do the data collection and sharing. On the other hand, to reduce this overhead, monitoring does not need to be continuous and the context information can also be shared and downloaded only at times of good connectivity. Furthermore, each phone gets to use the knowledge learned by all other phones in addition to the contextual knowledge learned through its own monitoring. So, the more participants there are in the system, the more beneficial it is for each individual one.

12.4.3 Case study: Prediction of Wi-Fi SNR

Next, we go through a case study where we apply different prediction methods to optimize the energy consumption of Wi-Fi communication. We consider the following scenario: *Alice's mobile phone is connected to a Wi-Fi access point and she is in the process of downloading a file. The network quality associated with the device fluctuates between poor and strong. The application, with the help of prediction, foresees the status of network, and hence chooses an adaptive action to transmit data during periods of good connectivity to save energy. The adaptive action can be pausing, stopping, or continuing the file download.*

In the above scenario, the power-management software adapts the network transmission to the wireless link quality to save energy. The wireless link quality can be measured using metrics like the SNR.

We take this scenario as an example and use the prediction techniques presented in Section 2.6.3 to predict the value of the SNR. Specifically, we build an ARIMA model which belongs to a class of models that are trained offline based on collected data sets. In other words, the procedure for offline training is to fit an ARIMA(p, d, q) model to the collected data sets. The output of the model fitting includes the orders of autoregressive, differencing, and moving average, p, d, q, and the estimated parameter values of the autoregressive operator, the moving average operator, and the intercept, φ, θ, μ.

The whole procedure starting from data collection can be summarized into the following six steps:

1. *Data collection:* We can log the SNR value at a fixed frequency on the phone, so that a time series can be divided into time slots with a fixed length. The length of the time slot is chosen so that the measured SNR value does not change more than once during one time slot. For example, we connected the SNR values of a Wi-Fi access point while walking around a defined path inside our office building. The path involves different types of physical obstacle, like wall structures and glass doors, and interferences caused by other wireless networks. We collected in total 12 data traces at different times during three days, where the collection of each set lasted 17 minutes. According to our SNR measurements sampled at 10 Hz, in the scenarios where the phones move with the users at walking speed, the measured SNR does not change more than once in one second. Hence, it would be accurate enough to sample SNR at 1 Hz in this scenario. The collected data sets can be divided into two parts, one for offline training, and the other for model validation.

2. *Data transformation:* Data smoothing is often used for working out future trends, such as the trend in stock prices. However, for short-term prediction, like SNR

prediction, it is less clear whether data smoothing still helps to improve the prediction accuracy. In practice, a good approach is to apply data smoothing over the training data set, and compare the prediction accuracy between the models using a smoothed data set and the original training data set. Here we give an example of applying a simple moving average algorithm for a time series data set smoothing. According to the moving average algorithm, from the collected N data points, we can perform averaging for every w data set, where w is often called the window size. The general expression for the moving average is given below.

$$M(t) = \frac{X_t + X_{t-1} + \cdots + X_{t-w+1}}{w} \tag{12.7}$$

In general, as the window size increases the trend of the smoothed data set becomes clearer, but with a risk of a shift in the function. To choose an optimal window size, we suggest performing smoothing with a varied window size and selecting the best one with the minimum MSE of the residuals. The residuals are the differences between the original values and the smoothed values.

3. *Model fitting:* Identifying the orders of autoregressive, differencing, and moving average by observing the ACF (autocorrelation function) and PACF (partial autocorrelation function) of the residuals. The ACF plot is a bar chart of the coefficients of correlation between a time series and lags of itself. The PACF plot is a plot of the partial coefficient between the series and lags of itself.

In practice, we first plot the ACF and PACF of the residuals of the prediction based on ARIMA(0, 0, 0). We can see from Figure 12.11(a) that the series has positive autocorrelation up to a high number of lags, which means it probably needs a higher order of differencing. As shown in Figure 12.11(b), the PACF plot has a significant spike only at lag 1 and the lag-1 autocorrelation is positive, meaning that all the higher-order autocorrelations are effectively explained by the lag-1 autocorrelation. In this case, the lag at which the PACF cuts off can be considered the indicated number of AR term. Similarly, if the ACF of the differenced series displays a sharp cutoff and/or the lag-1 autocorrelation is negative, then consider adding an MA term to the model. The lag at which the ACF cuts off is the indicated number of MA terms. More practical instructions of fitting ARIMA models can be found from an online course[2] provided by Duke University. Note that it often happens that different combinations of (p, d, q) give similar results at this stage. For the data sets we collected in Step 1, we get three candidate models, ARIMA(0,1,1) and ARIMA(1,0,1) for the original training data set, and ARIMA(2,1,0) for the smoothed training data sets.

4. *Parameter estimation:* Estimating the parameters of the expected models that obtain the minimum MSE is the next step. Here are the three models we obtain after this step:

ARIMA(0,1,1):

$$\hat{Y}(t) = Y(t-1) - 0.1142 \times e(t-1), e(t-1) = Y(t-1) - \hat{Y}(t-1) \tag{12.8}$$

[2] Website: http://people.duke.edu/˜rnau/411home.htm accessed March 7, 2014.

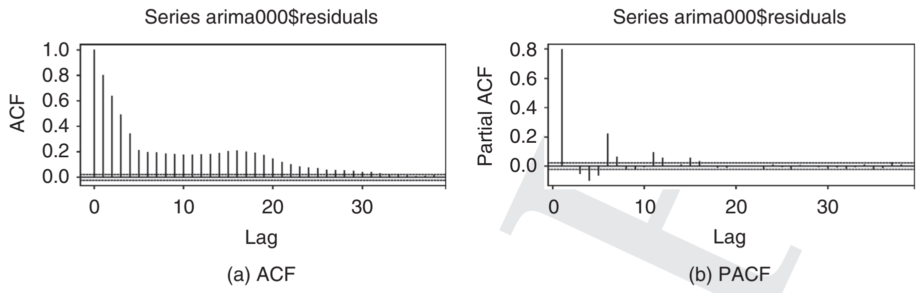


Figure 12.11 ACF and PACF plots of the residuals of the ARIMA (0,0,0) prediction

ARIMA(1,0,1):

$$\hat{Y}(t) = 22.165 + 0.7797 \times (Y(t-1) - 22.165) - 0.0052 \times e(t-1) \tag{12.9}$$

ARIMA(2,0,1):

$$\begin{aligned}\hat{Y}(t) = Y(t-1) + 0.7748 \times (Y(t-1) - Y(t-2)) - 0.015 \times (Y(t-1) \\ - 2 \times Y(t-2) + Y(t-3))\end{aligned} \tag{12.10}$$

5. *Model testing:* It is important to apply the models obtained from the previous step to the testing data sets and to compare the accuracy of the models. For example, we can calculate the MSE, NMSE, and SER for each model. NMSE is a function of the MSE normalized by the variance of the actual data. The smaller the MSE and NMSE are, the higher the accuracy is. Conversely, the bigger the SER is, the higher the accuracy is.

$$NMSE = \frac{MSE}{E[E(Y(t) - \hat{Y}(t))^2]} \tag{12.11}$$

$$SER = 10log_{10}\left(\frac{E[Y(t)^2]}{E[(Y(t) - \hat{Y}(t))^2]}\right) \tag{12.12}$$

6. *Model selection:* Finally, we select the model that provides the smallest MSE and NMSE, as well as the highest SER. Table 12.4 shows the accuracy of the three candidate models. ARIMA(2,1,0) with training based on the smoothed data set showed the lowest accuracy compared to the models obtained from the original data sets. ARIMA(1,0,1) fitted the testing data sets better than the others. Therefore, we chose ARIMA(1,0,1) for online SNR prediction.

The predicted SNR can be used as the conditions for triggering the adaptation to Wi-Fi data transmission. An example adaptive policy is controlling whether to pause or continue data transmission based on the comparison of the predicted SNR and predefined threshold values. The effectiveness of the adaptation depends on the tradeoff between the energy overhead caused by the SNR prediction and the energy savings made by increased network throughput. The former one is considered to be stable and

Table 12.4. Comparison of prediction accuracy among different models

Model	MSE	NMSE	SER
ARIMA(0,1,1)	97.586	0.425	7.457
ARIMA(1,0,1)	89.263	0.389	7.844
ARIMA(2,1,0)	174.452	0.760	4.934

Table 12.5. Classification of network conditions based on SNR statistics

Type	Description
1	SNR mean is smallerthan 15; SNR threshold is set to 15.
2	SNR mean is between 15 and 20; SNR threshold is set to 15 and 20.
3	SNR mean is bigger than 20, and standard deviation of SNR is smallerthan 5. SNR threshold is set to 20.
4	SNR mean is bigger than 20. and standard deviation of SNR is not smaller than 5. SNR threshold is set to 15 and 20.

independent of network conditions, whereas the latter one varies with network scenarios. To work out the impact from different network scenarios on the effectiveness of our adaptation, we divide the experimental network scenarios into four types based on the mean and standard deviation of SNR values. A description of each type can be found in Table 12.5. The type classification is mainly due to the fact that in real-time network measurements it is hard to get a stable SNR value.

According to our evaluation results, when SNR values are generally low, as in Type 1, the increase in network throughput is significant, and hence the adaptation is profitable. In the scenarios where the SNR values fluctuate heavily even though the mean of the SNR is high, such as Type 4, the adaptation could also save energy to a certain extent. If the network conditions are relatively stable, for example, in Types 2 and 3, when the standard deviation is less than 5, there is not much advantage for the threshold-based adaptation. Energy consumption could not be saved, but is wasted due to the energy overhead caused by network adaptations. In addition to the SNR range, the selection of the threshold has an impact on the effectiveness of our adaptation. For example, in the network conditions of Type 2, when the threshold is set to 20, the pause duration is close to 0. However, when the threshold is changed to 15, the adaptation becomes more energy efficient. Hence, in summary, threshold-based adaptation is energy efficient when network conditions fluctuate in a big range or remain in a relatively bad state. In other words, when SNR values are generally low, such as lower than 15, or when the standard deviation of the SNR values is large, network adaptations help save energy by up to 40%.

12.5 Scheduling multiple devices

So far, we have discussed how data communication should be scheduled for a single device. It turns out that when multiple smartphones communicate using the same network at the same time, new challenges arise. Specifically, interference from other communicating devices using the same wireless channel is another source of extra energy consumption that can be reduced through scheduling.

The problem of co-channel interference applies specifically to random access wireless networks, such as Wi-Fi that uses CSMA/CA, where no central coordination between the communicating devices exists. In such networks, interference due to two or more devices incidentally transmitting at the same time causes packet loss because the transmitted signals add up to a signal that cannot be decoded by the receivers. Packet loss requires retransmissions which consume energy.

Furthermore, random access requires channel sensing and the probability that a channel needs to be sensed several times before finding a free channel increases with the number of simultaneously communicating devices. Energy is consumed each time the channel is sensed. Energy-aware scheduling mechanisms for random access wireless networks have been extensively studied and Wi-Fi is, understandably, by far the most-studied technology in this context. We next look at the specific problems related to energy consumption that arise in Wi-Fi communication and solutions to mitigate them.

12.5.1 Reducing Wi-Fi contention when using PSM

In addition to the sources of energy waste described above that are inherent to random access networks, Wi-Fi exhibits particular issues related to scheduling due to the power-saving mode (PSM) which is widely used nowadays. Luckily these issues can be addressed to a certain extent with clever techniques as we will see.

The first problem stems from the way that PSM is designed to work. When a client using PSM is asleep, packets destined to it are typically put to the end of the transmit queue of the AP. When the client wakes up from sleep mode to active mode to receive the beacon frame and learns that packets are waiting to be delivered, it sends a request to the AP in order to receive the pending frames.

The problem is that the woken up client may need to wait for some time before being served, if packets destined to other clients are ahead in the queue. Figure 12.12 illustrates this scenario. Upon the second beacon, Client 1 receive its frames quickly because Client 2 had no frames buffered to it. However, after the third beacon, Client 1 needs to wait with the radio powered on and listening to the channel while Client 2 is being serviced. Energy is being consumed all that time. In this example, Client 2 is also using PSM but such contention may also happen for other clients that are in continuously active mode (CAM), that is they do not use PSM. Prioritizing the traffic of PSM clients would help reduce such contention when a PSM client competes with CAM clients but it creates unfairness. In addition, that solution does not work when multiple PSM-enabled clients compete over the channel.

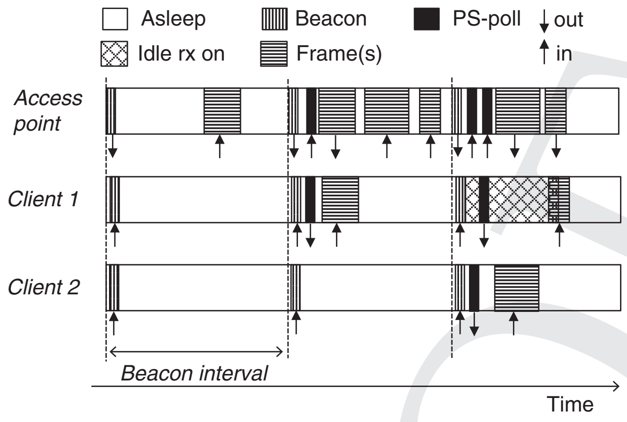


Figure 12.12 Contention adds to energy consumption when using PSM with Wi-Fi

Some modifications to the 802.11 standard have been proposed to schedule the PSM clients in a time-division multiplexing (TDM) manner [17]. In this approach, the traffic can be scheduled in a near theoretically optimal manner making sure that the clients do not need to spend time listening to the channel. The downside of this scheduled-PSM approach is that it requires changes to both APs and clients, which means that standard off-the-shelf equipment cannot be used as such.

A technique called AP virtualization is supported by modern Wi-Fi APs and it can be used to alleviate the problem with multiple PSM clients without modifying the standard 802.11 behavior. In brief, the solution consists of specifying different PSM beacon schedules for different clients in such a way that the devices do not wake up a the same time and, as a consequence, do not request the buffered packets from the AP at the same time. NAPMan is a solution that operates in this way [18]. Some software changes are required to the AP in this approach as well, but off-the-shelf clients can be used.

12.5.2 Reducing inter-AP contention in Wi-Fi

The second problem is related to dense Wi-Fi deployments. The IEEE 802.11 standard specifies a set of different channels that can be used to avoid co-channel interference between APs in close proximity to each other. Indeed, these APs do not need to hear each other's traffic so they can communicate on a completely different frequency. However, there are only a few completely non-overlapping channels, namely the channels 1, 6, and 11 in the 2.4 GHz band. There are more in the 5 GHz band, which is used by 802.11n/ac in addition to the 2.4 GHz band. For this reason, these same non-overlapping channels are very frequently used in Wi-Fi deployments.

As a result, it is not uncommon for a Wi-Fi-enabled smartphone to overhear many access points at the same time, for instance, in an office or home environment (e.g. in an apartment building). A client associated with one of these APs faces interferences from overhearing the communication of another AP or a client connected to another

AP even if the communication of the clients of a single AP would be energy-efficiently scheduled.

Of course, having identified the problem, it is possible to devise a solution for it. A logical approach would be to extend the solution that reduces single AP contention to multiple APs, which is in essence what SleepWell [19] does. It tries to identify the periodicity (caused by PSM beacon schedules) of the APs it overhears and reschedules its own PSM cycle so that they would minimally coincide with the others. Similar to the single AP solution, this approach also requires some software changes in the APs but again smartphones need not be modified.

12.6 Summary

In this chapter, we studied traffic scheduling as a way to improve the energy efficiency of the wireless communication of a smartphone. We identified several scenarios where scheduling can be used:

- *Traffic shaping*, where high-rate bursts are formed from lower-rate traffic through a kind of buffer and release mechanism, is an effective way to improve the energy utility (bits per joules) of wireless data communication. It is based on the fact that the higher the data transmission rate, the higher typically also the energy utility. Tail energy is spent only once per burst. It is applicable in situations where the data transfer would not otherwise use all the available bandwidth at the wireless access link. In addition, the application whose traffic is shaped must be such that delaying some of the packets is not critical to the QoS. The amount of energy that can be saved depends on the wireless access technology used, the amount of spare bandwidth available, and the longest allowed duration for delaying packets, which directly influences the interval between the bursts and their sizes.
- *Scheduling background transfers* is important with smartphones today because of many chatty background applications that cause small amounts of data to be transmitted sporadically, which is very energy inefficient. Examples of such applications include those that require periodic synchronization and those that push some content to the phones, such as social network applications. Also advertisements embedded in mobile applications generate such traffic. If the amount of background traffic cannot be reduced, the traffic can be either scheduled to overlap with foreground transfers or can be bundled together to form larger bursts of traffic. Both approaches improve the energy utility of the individual transfers.
- *Context-aware scheduling* can sometimes save a very substantial amount of energy. This is because the energy utility of wireless data communication is highly dependent on the load of the access network and on the quality of the connectivity, which both vary with time and location. Network load affects the throughput achievable, while the quality of the wireless connectivity, that is the signal strength, influences two energy-related attributes: the over-the-air data rate through the modulation scheme selected and the transmit power used. The biggest challenge in context-aware scheduling is

knowing or being able to predict the periods of time when connectivity will be good and energy efficiency of the data transfer will be high. Several schemes have been designed to that end, some of which track the mobility of the user and try to learn periods of good connectivity from the history, while others rely on direct probing. Yet another approach is crowd-sourcing and sharing of the context monitoring.

- *Reducing Wi-Fi contention through scheduling* is necessary due to the random access nature of the channel access in the 802.11 standard. Contention can cause unnecessary energy waste when a smartphone Wi-Fi network interface wakes up from sleep mode to receive packets but has to wait because other devices are occupying the channel. Two varieties of this problem can be found in typical Wi-Fi deployments: intra-AP contention, where several devices compete for the resources of a single Wi-Fi AP; and inter-AP contention, where the overheard neighboring APs cause completely unintentional contention in the AP in question. Scheduling solutions to mitigate both problems have been designed. The most attractive solutions can be used directly just by adding support to APs without any change to the 802.11 standard protocols or to the smartphones.

References

[1] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proc. 7th ACM European Conf. on Computer Systems*. New York, NY, USA: ACM, 2012, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168841

[2] Y. Xiao, Y. Cui, P. Savolainen, M. Siekkinen, A. Wang, L. Yang, A. Yla-Jaaski, and S. Tarkoma, "Modeling energy consumption of data transmission over Wi-Fi," *IEEE Trans. on Mobile Computing*, vol. 99, no. PrePrints, 2013.

[3] M. Hoque, M. Siekkinen, and J. K. Nurminen, "TCP receive buffer aware wireless multimedia streaming – an energy efficient approach," in *Proc. 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, 2013, pp. 13–18.

[4] I. Kelenyi, A. Ludanyi, and J. Nurminen, "Using home routers as proxies for energy-efficient bittorrent downloads to mobile phones," *IEEE Commun. Mag.* vol. 49, no. 6, pp. 142–147, 2011.

[5] F. R. Dogar, P. Steenkiste, and K. Papagiannaki, "Catnap: exploiting high bandwidth wireless interfaces to save energy for mobile devices," in *Proc. 8th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2010, pp. 107–122. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814446

[6] "Docomo demands google's help with signalling storm," [Online]. Available http://www.rethink-wireless.com/2012/01/30/docomo-demands-googles-signalling-storm.htm, Jan. 2012.

[7] N. Cardwell, S. Savage, and T. Anderson, "Modeling TCP latency," in *INFOCOM 2000. Proc. 19th Annual Joint Conf. of the IEEE Computer and Communications Societies*, vol. 3, 2000, pp. 1742–1751.

[8] J. Nurminen, "Parallel connections and their effect on the battery consumption of a mobile phone," in *2010 7th IEEE Consumer Communications and Networking Conf. (CCNC)*, pp. 1–5.

[9] Qualcomm, “Managing Background Data Traffic in Mobile Devices,” January 2012. [Online]. Available: http://www.qualcomm.com/media/documents/managing-background-data-traffic-mobile-devices

[10] E. Estep, “Mobile HTML5: Efficiency and Performance of WebSockets and Server-Sent Events,” Master’s thesis, Dept. of Computer Science and Engineering, Aalto University, School of ScienceHelsinki, Finland, Jun. 2013. [Online]. Available: http://nordsecmob.aalto.fi/en/publications/theses2013/thesis_estep

[11] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, “Don’t kill my ads!: balancing privacy in an ad-supported mobile application market,” in *Proc. 12th Workshop on Mobile Computing Systems and Applications*. ACM, 2012, pp. 2:1–2:6.

[12] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft, “Breaking for commercials: characterizing mobile advertising,” in *Proc. 2012 ACM Conf. on Internet Measurement*. New York, NY, USA: ACM, 2012, pp. 343–356. [Online]. Available: http://doi.acm.org/10.1145/2398776.2398812

[13] P. Mohan, S. Nath, and O. Riva, “Prefetching mobile ads: can advertising systems afford it?” in *Proc. 8th ACM European Conf. on Computer Systems*. New York, NY, USA: ACM, 2013, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465378

[14] A. Chakraborty, V. Navda, V. N. Padmanabhan, and R. Ramjee, “Coordinating cellular background transfers using loadsense,” in *Proc. 19th Annu. Int. Con. on Mobile Computing and Networking*. New York, NY, USA: ACM, 2013, pp. 63–74.

[15] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, and V. N. Padmanabhan, “Bartendr: a practical approach to energy-aware cellular data scheduling,” in *Proc. 16th Annu. Int. Conf. on Mobile Computing and Networking*. New York, NY, USA: ACM, 2010, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/1859995.1860006

[16] Y. Xiao, P. Hui, P. Savolainen, and A. Ylä-Jääski, “Cascap: cloud-assisted context-aware power management for mobile devices,” in *Proc. 2nd Int. Workshop on Mobile Cloud Computing and Services*. New York, NY, USA: ACM, 2011, pp. 13–18. [Online]. Available: http://doi.acm.org/10.1145/1999732.1999736

[17] Y. He and R. Yuan, “A novel scheduled power saving mechanism for 802.11 wireless lans,” *IEEE Trans. on Mobile Computing*, vol. 8, no. 10, pp. 1368–1383, 2009.

[18] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu, “Napman: network-assisted power management for wifi devices,” in *Proc. 8th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2010, pp. 91–106. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814445

[19] J. Manweiler and R. Roy Choudhury, “Avoiding the rush hours: Wifi energy management via traffic isolation,” in *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2011, pp. 253–266. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000020

13 Exploiting multiple wireless network interfaces

In this chapter, we continue to look at the different ways to reduce the energy consumed by wireless communication. Our focus is on how to take advantage of the fact that modern smartphones include many different wireless technologies integrated under the hood and can even SWITCH seamlessly between some of those.

13.1 How using multiple WNIs saves energy

Smartphones today contain many different radio technologies including Wi-Fi, Bluetooth, BLE, and cellular radios. As we learned in Section 7.3, using different WNIs can cause quite different amounts of energy to be consumed. More importantly, the energy utility of the different technologies vary substantially. For this reason, opportunities to save energy arise by using the WNIs wisely.

Recall from Section 7.3 that energy is consumed in non-connected and connected modes. The former essentially means discovering an AP or another device in order to establish communication, while the actual data transfer happens in the latter mode. The amount of energy spent in such a discovery process can account for a large part of the total amount of energy consumed. Fortunately, the differences in the energy consumption between the different kinds of radio can be used in that process. In addition, keeping multiple radios continuously powered on in the smartphone is usually unnecessary. In most cases, it is enough to have one radio active so that the phone remains reachable at all times to be able to receive phone calls or incoming messages pushed by mobile services.

Energy consumption can be optimized in both non-connected and connected modes by the smart use of different radios. We look at both cases and highlight the discussion with example solutions proposed in the literature. We group the different techniques into three categories which are summarized in Figure 13.1. We discuss each of them in the following sections. Toward the end of the chapter, we go through a more detailed case study on traffic offloading and its energy benefits for smartphones.

13.2 Tracking movement to optimize the discovery energy

The first category includes techniques to assist Wi-Fi ap discovery. There are two sub-categories, the first of which is tracking the movement of the smartphone. To

Table 13.1. Techniques to save energy by using multiple WNIs of the smartphone

Technique used	How it saves energy	Example solutions	Pros	Cons
Detecting user movement.	Reduce energy wasted by unnecessary scanning.	*Footprint:* Tracks movement using cellular network neighboring cell info.	Movement detection comes almost energy free. Does not require historical data.	Base station deployment density and patterns influence the performance of the system.
		WiFisense: Infers movement using built-in sensors.	Does not depend on other wireless comm. techs and networks and their characteristics.	Using motion sensors consumes some energy.
Assisted Wi-Fi AP discovery.				Training needed to create and maintain the fingerprint database.
Using hints from other radios.	Reduce energy wasted by unnecessary scanning.	*Blue-Fi:* Learns Bluetooth contact-patterns and cell-tower information.	Bluetooth deployed in all smartphones nowadays.	Mobile Bluetooth devices cause problems. Discovery in Bluetooth also consumes energy.
		Zi-Fi: Detects presence of Wi-Fi AP beacons using ZigBee radio.	Does not require changes to network infrastructure.	Smartphones currently on market do not (yet) have a ZigBee radio.

(*cont.*)

Table 13.1 (*cont.*)

Technique used	How it saves energy	Example solutions	Pros	Cons
Wake-up phone/other radios using single always-on (low-power) radio.	Reduce energy consumed by idle powered on radios.	*Wake on wireless:* Keep only low power radio on that will wake up the rest of device upon receiving magic packet.	Potentially very low power solution. The concept is applicable with different kinds of radio.	Notifying the smartphone through a custom short-range radio requires intermediate proxies.
		Ce112Notify: Notify phone through cellular network to receive VoIP call through Wi-Fi.	Does not require special hardware. Applicable also with lower-end phones.	Requires custom notification solution. May become obsolete with the coming of voice over LTE.
		Context for wireless: Context-based selection between cellular and Wi-Fi for data transmission.	Does not depend on external wireless comm. techs.	Requires historical data and training.
Energy-aware selection of WNI for data transmission.	Improve energy utility of data transmission.	*CoolSpots:* Policy-based selection between Bluetooth and Wi-Fi for data transmission.	Radio switching policies applicable also to other multi-radio setups.	Requires Bluetooth enabled APs that support their protocol.
		Wizi Cloud: Merges ZigBee and Wi-Fi together and uses either one depending on which is deemed more suitable (energy and bandwidth).	Potentially very good energy efficiency because of the low power ZigBee.	Smartphones currently on market do not (yet) have a ZigBee radio. Requires also ZigBee enabled APs.

understand why it can help save discovery energy, consider a phone that scans for existing Wi-Fi APs periodically with a given fixed interval. Now think about your movement patterns during a typical week day. Most people remain static most of the time, which is why it makes no sense for the smartphone to continuously search for new APs which do not move either. So, tracking mobility to perform an AP scan only when the phone has moved to a different location can save a substantial amount of energy.

There are different ways of tracking smartphone movement. A crucial point is that this continuous tracking must consume in total less energy than periodic Wi-Fi scanning. For this reason, using the default location services available in smartphones today is too power hungry to save any energy. They usually rely on GPS, among other techniques, which is very power hungry and does not work indoors. Furthermore, those location services available in smartphones that do not rely on GPS but extract location information from cell tower IDs, for example, usually make queries to online servers in order to map the extracted information into absolute location, that is coordinates.

A less energy consuming way for movement tracking is to continuously inspect the IDs and received signal strengths of the overheard neighboring cell towers of the cellular network that the smartphone is connected to. When that set of cell IDs and their relative signal strengths change significantly, the phone can infer that it has moved a certain distance. Note that the difference to using normal location services is that in this scheme the phone does not learn its absolute position but only relative displacement, which is enough for the purposes of movement-based triggering of discovery. Hence, the energy consumed in resolving the absolute location is saved. In addition, the information about neighboring cells enables the phone also to detect already visited and scanned locations to a certain extent in order to defer rescanning in such situations. This technique is used by Footprint [1], for instance, and the nice property of it is that the neighboring cell information comes almost without any extra energy consumption since the phone is in any case always connected through the cellular network interface. The downside of this approach is the fact that the movement estimation is dependent on the cellular network deployment: If the cell deployment and configuration are dense, the set of overheard neighboring cells changes after a shorter distance traveled compared to a less-dense network. Therefore, configuring the system to optimally trigger a new scan is challenging.

Another way to track the movement is by using the internal sensors of the smartphone. As we learned earlier, smartphones contain a number of motion sensors that can be used to track the movement of the phone. In addition, using these sensors typically draws relatively little power but it obviously depends on the rate at which they are providing measurement samples. Furthermore, accurate movement tracking requires sensor fusion, which means getting input from several motion sensors, such as the accelerometer and gyroscope, but coarser grained and less accurate information can be obtained with fewer or just one sensor (e.g. accelerometer). So, there is an inherent tradeoff between accuracy and energy consumption with this approach. WiFisense [2] is an example of such a technique.

Movement tracking is primarily applicable to reducing the energy consumption of Wi-Fi scanning. In principle, it could also be used with other technologies, like Bluetooth or BLE. An interesting future use case is where a smartphone opportunistically discovers BLE sensors deployed in different environments to provide contextual information. Such a scenario requires the smartphone to continuously scan for these BLE sensors and, therefore, movement tracking could help save energy. However, the margin for saving energy is much tighter in this case because BLE scanning consumes less energy than Wi-Fi does, which means that the movement tracking must be done using very little power.

13.3 Using hints from other WNIs

The second set of techniques is closely related to the first one and the objective is the same: to reduce the energy consumption of the Wi-Fi discovery process. However, instead of tracking movement, the idea is to extract different hints from other WNIs that either use less power than Wi-Fi or that are already powered on and, therefore, introduce no extra energy overhead.

A solution called Zi-Fi [3] detects the presence of beacons from a Wi-Fi AP using a ZigBee radio. The insight is that both radios operate on the same shared radio frequency band–the ISM (Industrial, Scientific, and Medical)–which allows the ZigBee radio to overhear the Wi-Fi communication, although it is unable to decode its contents. So, through clever processing of the received signal, the presence of the periodic Wi-Fi beacons can be detected. Hence, a smartphone that has both kinds of WNI can keep only the ZigBee powered on and switch on the Wi-Fi radio only when an AP presence is detected. Current smartphones do not typically include a ZigBee radio, which obviously undermines the use of this scheme. It is unlikely that a similar method could be applied with Bluetooth because of its frequency-hopping channel access.

Another approach is for the smartphone to learn the AP locations and the location signature from Bluetooth contact patterns, as is done by Blue-Fi [4]. The idea is that different kinds of Bluetooth devices and peripherals exist in the environments visited by a person during a typical day, and that continuously scanning these devices consumes overall less energy than Wi-Fi scanning would. So, the first time that a phone visits a location where it discovers a Wi-Fi AP and some Bluetooth devices, it memorizes the Bluetooth contact patterns. Next time it would only use these patterns to infer that a Wi-Fi AP should be present and Wi-Fi scanning should be performed. In addition, cellular network base station IDs are used to provide some hints to augment the usability of the scheme for situations where few or no Bluetooth devices are present. The drawback of this approach is that it requires training during which the Bluetooth fingerprint database is created. It also needs to be maintained afterwards. Another challenge is mobile Bluetooth devices that may give misleading information about Wi-Fi AP locations. Finally, the Bluetooth discovery process is also far from being energy free.

13.4 Sleep and wake-ups to reduce idle power

The third way that multiple WNIs can be used to improve energy efficiency is to reduce the idle power. The idea is to power off idle radios or the whole device if they are not currently in use but in such a way that one radio is kept powered on so that the phone can still receive notification messages that will trigger another radio or the whole device to wake up when, for instance, an incoming call is received.

Wake on wireless [5] is already a decade-old solution that introduced this concept. It keeps only a low-power radio on while the rest of the device sleeps. When there is an incoming call, this radio reacts to a wakeup message by waking up the rest of the device that otherwise sleeps. This solution is not used by smartphones today and it is mostly interesting from conceptual point of view. The concept itself could be applied with other sets of radios as well, so that only the lowest power radio is kept powered on to wake up the other radios on demand.

One specific way to apply this concept is to simply power off the Wi-Fi interface and receive a notification through the cellular network when it should be switched on. Cell2Notify [6] does this and the target application is VoIP over Wi-Fi. Whenever there is an incoming VoIP call, the phone receives a notification via the cellular network interface to wake up the Wi-Fi interface. There is a high chance that this scheme will become obsolete in the near future when smartphones and LTE networks begin to use voice over LTE (VoLTE) together with optimized DRX profiles.

The downside of this wake-up radio approach in general is that it requires intermediate proxies that handle the notification of the smartphone when there is an incoming data packets or call.

13.5 Energy-aware wireless network interface selection

The fourth set of techniques optimize the energy efficiency of data transmission. The energy savings come from dynamically selecting the WNI that is estimated to consume the least energy for a given transaction.

There are a couple of factors that a well-working energy-aware WNI selection policy must capture. As we have discussed before, the different WNIs do not generally have the same kind of power proportionality. Therefore, among the WNIs available in a smartphone, there may not be single one that has the highest energy utility while transmitting or receiving a specific amount of data as a bulk transfer. Instead, the best choice depends on the amount of currently available bandwidth. Similarly, applications that throttle the transmission rate may consume more or less energy over different WNIs, depending on the enforced data rate. For example, a relatively low-bit rate audio stream may be the most energy efficient to receive using Bluetooth, whereas a high-quality video stream would be best to receive using Wi-Fi. In addition, other factors such as signal strength should be accounted for.

For the proposed solutions, CoolSpots [7] makes decisions dynamically between using Bluetooth or Wi-Fi; Context-for-wireless [8] switches between cellular (3G) and

Wi-Fi networks; and Wizi Cloud [9] considers Wi-Fi and ZigBee WNIs. The challenge for CoolSpots and Wizi Cloud deployment is that they require Wi-Fi access points to integrate a Bluetooth or a ZigBee radio. Context-for-wireless, on the other hand, relies on historical data which means that some amount of training for the system is necessary.

13.6 Use case: Energy awareness in mobile traffic offloading

In this section, we continue the discussion on using different WNIs in smartphones with a detailed treatment of a specific use case. We look at techniques and examples for saving energy through mobile traffic offloading.

The primary purpose of traffic-offloading solutions is usually to alleviate traffic load from one network by automatically rerouting it via another network. In mobile communication, the most common scenario is to offload traffic from a cellular network to a Wi-Fi network. However, since smartphone battery life is crucial to the user experience, energy awareness of the traffic-offloading solutions plays an important role in the successful adoption of offloading solutions, as acknowledged by recent research [10, 11, 12, 13].

In this section, we first explain the basic concepts of mobile traffic offloading after which we illustrate the core issues in mobile traffic offloading from the energy perspective and discuss feasible approaches for improving the energy awareness in mobile traffic offloading.

13.6.1 Mobile traffic offloading

Cellular networks are suffering from the tremendous growth of mobile data traffic in recent years [14, 15]. The pressure has driven operators to search for solutions that can alleviate network congestion and fully use the existing network resources. Recent studies suggest that mobile traffic offloading is a feasible approach to alleviate this problem by using complementary communication technologies, such as Wi-Fi and femtocells, to deliver traffic originally targeted for cellular networks [16, 17, 18, 19].

The key enabler for mobile traffic offloading is the rapid evolution of wireless communication technologies. As we have already discussed in earlier parts of the book, smartphones now include a rich set of communication technologies, such as Wi-Fi, HSPA, LTE, Bluetooth, and BLE. On the network side, mobile operators are upgrading to LTE and LTE-advanced (4G), and Wi-Fi and femtocells are gaining popularity in metropolitan areas to offer diverse and convenient wireless access.

Traffic offloading requires that there is a suitable match between availability and demand for network capacity at the right time and the right place. Therefore, the deployment and performance of different wireless networks, user mobility, and traffic characteristics generated by application usage are key components in determining the applicability of a given traffic-offloading scheme. The typical mobile traffic offloading scenario consists of six main steps:

1. Offloading Initiation: The offloading procedure can be initiated by the network side (network-driven offloading), or the smartphone (user-driven offloading). Network-driven offloading is often triggered by dedicated signaling protocols, such as router advertisement, enabling operators to dynamically manage and balance their traffic load. User-driven offloading is triggered by applications that need to access the Internet for content, which is based on the demand of the user.

   Network-driven offloading introduces overhead in terms of extra signaling and potential energy cost, but it can offer timely and optimized offloading guidance based on the comprehensive knowledge from the network side, that is the network structure and condition. On the other hand, user-driven offloading avoids the extra signaling cost but lacks the network context and is less efficient for users moving at high speed, for example, driving or cycling.
2. Context Collection: The context information is essential for mobile traffic offloading, especially as input for the offloading decision. Users can obtain context information either from network operators or from the surrounding access environment. The key information includes the user location, network access condition (e.g., SNR of Wi-Fi access points, wireless fingerprint information [20]), potential offloading targets, and connection detail (e.g., ESSID or MAC addresses)).

   The collected context information will be fed to either remote/cloud controlling servers or local management components. For the remote option that uses the cloud support, a dedicated signaling channel is required, such as cellular data connection. Therefore the proposals relying on remote support are limited by the channel condition, especially when such a channel is congested or the infrastructure suffers from instability due to technical issues. This also affects the scalability due to the dependence on a centralized entity. Compared to the first option, the local solution does not depend on external entities. However, by relying solely on local resources, context information can be incomplete or less accurate compared to the remote option.
3. Offloading Decision: The decision process involves computation according to the pre-defined algorithm or operation logic, and delivering control messages to mobile users to carry out offloading. By taking the context information as an input, an offloading decision can be made either at the network side or using local resources on the mobile device.

   By offloading the computation to the network side, we can improve efficiency in terms of energy and latency for using the powerful hardware. However, this approach depends heavily on the infrastructure support. On the other hand, a local decision can be more flexible and robust to network conditions, but at the cost of local resources such as energy. The local operation also suffers from the limitation that there is no external knowledge available to improve the accuracy of offloading decision.
4. Network Association: Based on the offloading decision, mobile devices need to perform network association to enable traffic offloading. The association process includes access/peer discovery via a pre-defined configuration protocol, such as DHCP and DNS, to establish connectivity with the target offloading networks.

   When users are moving at high speed, the connectivity period for offloading is often short. This demands an efficient association at both hardware and software levels supported by optimized protocols to avoid an excessive cost of association, which will decrease the time for data transmission.

5. Data Transmission: As the key part of mobile traffic offloading, data transmission determines how much data can be offloaded from the congested cellular networks to improve the overall service quality. Depending on the types of data traffic (e.g., real-time streaming, delay-tolerant traffic, web surfing), the offloading design can use the traffic characteristic for optimization.

   In the short period of offloading, which is typical for mobile users, the bandwidth and condition of networks can greatly affect the offloading efficiency, that is, the amount of data offloaded against the data volume that would flow to the cellular network otherwise [17]. At the same time, offloading design should also take into account the hardware limitation on mobile devices, such as wireless antenna.
6. Offloading Termination: A successful offloading session must end with terminating the temporary offloading connection and smoothly switching to other available networks to continue the data communications. It is important to keep the data flow uninterrupted and hence deliver a satisfying user experience. The prior research work on handover mechanisms have illustrated how to seamlessly migrate from one access network to another [21, 22, 23, 24].

   To enable efficient and smooth termination, guidance can be obtained from either the network side or local heuristic prediction to spot potential connectivity [25, 26, 27]. The termination process is also one of the key factors affecting the adoption of mobile traffic offloading.

13.6.2 Energy consumption in mobile traffic offloading

The impact of mobile traffic offloading can be evaluated from two aspects, the network (operator) perspective and the user perspective, and the ultimate goal is that both network operators and users benefit from it. The network operator is concerned about the offloading efficiency, that is, to offload as much traffic as possible to alleviate the pressure of cellular networks, whereas a key factor from the user perspective is the smartphone battery life. While much of the earlier work in traffic offloading focused on the network perspective, more recent studies have also considered the user perspective [10, 11].

We break down the problem domain and analyze the energy cost in each step of offloading. The major concerns are highlighted as follows:

1. In the initiation phase, if offloading is triggered by the network, signaling consumes energy. If signaling messages are delivered too frequently with large volume of data, such unintentionally recurrent interactions can promote the cellular network interface state and lead to excessive draining of the battery [28]. The problem is comparable to the case of background traffic. On the other hand, the user-driven initiation can not benefit from the proactive guidance but it does not consume extra energy in this phase.

Table 13.2. Measured energy consumption of 20 MB data transfer

	Nexus S		N900	
	Throughput	*Energy*	*Throughput*	*Energy*
Cellular	1.99 Mbps	65.40 J	1.89 Mbps	109.4 J
Wi-Fi	0.302 Mbps	191.7 J	0.422 Mbps	116.0 J

Table 13.3. Average measured power (watt) for cellular P_{Cell} and Wi-Fi P_{WiFi}, and energy consumption (joule) related to mobile traffic offloading: E_{W_on} and E_{W_off} for turning Wi-Fi on and off, E_{asso} for network association, E_{scan} for Wi-Fi scanning, E_{GPS} for GPS positioning, measurements repeated ten times

Device	P_{Cell}	P_{WiFi}	E_{W_on}	E_{asso}	E_{W_off}	E_{scan}	E_{GPS}
Nexus S	0.891±0.02	0.658±0.16	0.27±0.02	0.25±0.05	0.29±0.02	0.27±0.02	10±1.3
N900	1.10±0.02	0.645±0.02	0.18±0.03	0.28±0.13	0.13±0.02	0.53±0.08	4.0±1.3

2. The context-collection phase has to be done carefully too. As we discussed earlier, frequent scanning of available networks consumes a significant amount of energy. If GPS is used, a cold start may take around 20 seconds and consume ~6.30 joules [10]. If the context information is delivered to a remote server, it also costs energy.
3. In the decision-making phase, if the computation is done locally, the computational complexity of the algorithm determines the energy consumption. If the computation is offloaded to cloud, the data communication due to the exchange of context and decision messages consumes energy. We discuss computation offloading in more detail in Chapter 14.
4. In the association phase, the connection establishment that involves various hardware and protocol operations can consume a non-negligible amount of energy. For instance, DHCP alone can consume 4.8 joules [10]. Therefore, when the smartphone user moves at a high speed, the frequent associations accelerate the smartphone battery drain.
5. We learned earlier that wireless communication is not power proportional to the data rate. Therefore, in the data-transmission phase of traffic offloading, the amount of available bandwidth greatly affects the energy consumption. By taking Wi-Fi-based offloading using the Nexus S as an example, as shown in Table 13.2, when Wi-Fi throughput is lower than the cellular access, offloading cellular traffic to Wi-Fi can double the energy consumption.
6. The termination phase should be handled in a timely manner. Otherwise, periods of poor connectivity and low bandwidth may cause much more energy to be consumed unnecessarily for data transmission.

To get a better understanding of the energy consumption in traffic offloading, we show in Table 13.3 the typical energy expenditures in Wi-Fi-based offloading.

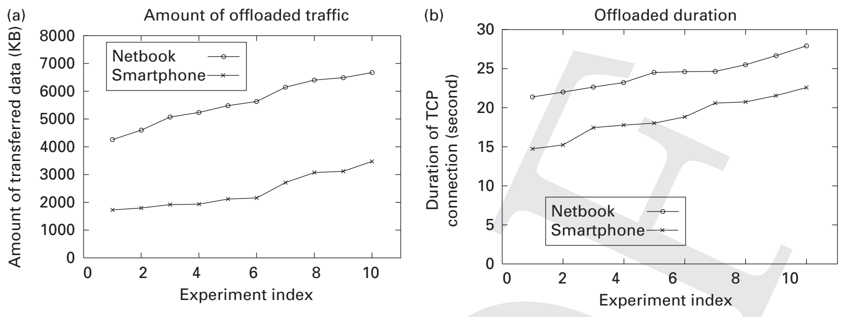


Figure 13.1 Offloading performance at driving speed

In the practice of traffic offloading, the offloading solutions that focus on maximizing the offloading efficiency may not always be able to reach their goal. One main reason comes from the fact that many existing mechanisms are based on experiments on laptops connected to a vehicle power supply. As shown in Figure 13.1(a), there is a clear gap between netbooks and smartphone-alike devices in terms of network performance.

For wireless communications, it is well-known that the antenna plays an important role. For example, a measurement study in the US [29] reported that a 12 dBi antenna provides better connectivity than 5 and 7 dBi antennas. Eriksson et al. [30] also found that mounting an external antenna on the roof of a car can significantly increase the signal strength of received Wi-Fi frames to gain better performance. However, due to the limited size of smartphones, it is very challenging to use external antennas. Figure 13.1(b) demonstrates that smartphones clearly have a shorter window of connectivity available for traffic offloading compared to netbooks, which suggests that traffic offloading is more challenging with smartphones than with netbooks, especially when moving at high speed.

13.6.3 Enabling energy awareness in mobile devices

We next look at different approaches to enabling energy awareness in mobile traffic offloading, which would benefit users by extending the smartphone battery life while still providing a desirable outcome from the network operator perspective. Because the traffic-offloading process involves multiple entities, including cellular network operators, alternative wireless access network providers, and end users, it is necessary to establish collaboration among these entities. We summarize the viable approaches to enable energy awareness in mobile traffic offloading in Table 13.4.

For offloading initiation, it is crucial to avoid frequent network signaling with a large payload that can lead to extra energy consumption due to the change of cellular radio status [28]. To benefit from the network support and to support dynamic traffic management, an adaptive scheme that combines the user-driven and network-driven initiation is recommended. Such a scheme strikes a balance between the efficiency and energy consumption [10, 13].

In the context-collection phase, constant scanning must be avoided if possible because of its high energy consumption on mobile devices. Due to the relatively high

Table 13.4. Approaches to enable energy-aware traffic offloading

Phase	Recommendations
Offloading Initiation	1) Avoid frequent signalling from network side that triggers cellular status change. 2) Combine the user-driven and network-driven in a adaptive manner to improve efficiency.
Context collection	1) Avoid unnecessary scanning and frequent GPS operation. 2) Use energy-efficient positioning mechanism such as Wi-Fi fingerprinting. 3) Adapt the context management for both local and remote processing to strike a balance.
Offloading decision making	1) Use energy-aware algorithm to guide the decision. 2) Adopt dynamic mechanism (e.g., machine learning) to update the logic according to the network condition. 3) Use the cloud support to offload the energy cost from intensive computation.
Network association	1) Avoid time-consuming association operation or protocol. 2) Use guidance from the network side if possible to assist authentication and access connectivity.
Data transmission	1) Adopt optimization schemes for different types of traffic (e.g., delay-tolerant, streaming). 2) Avoid transmission over unstable or low throughput wireless links by predicting the user mobility and network condition.
Offloading termination	1) Use the hints if possible from either the networks or local controller for switching between connections. 2) Avoid frequent termination that can shorten the data transmission time.

energy consumption and the long latency to obtain a fix from the coldstart, the GPS use in traffic offloading needs to be assisted by an energy-efficient design, such as using techniques proposed in [31, 32]. To support energy-efficient positioning, Wi-Fi-based positioning can also be used as an alternative. For context processing and management, sharing the load between local devices and remote servers is recommended, as this fully uses the knowledge of the access network and infrastructure, as well as the computing resources of both sides.

Energy consumption must be a key factor in making traffic-offloading decisions. In such decision-making, dedicated energy-aware algorithms can be used together with other factors, such as offloading capacity and network performance. For example, when the throughput over the cellular network increases, the energy savings achievable by offloading traffic to other networks obviously decrease because of the increased energy utility of the cellular network access. One such energy-aware algorithm is presented in [10]. Due to the fast change of network conditions, the energy-aware offloading decision needs to be adaptive. To save energy consumed in computational work, offloading intensive computation load, such as mobility predication, to the network side is recommended.

To minimize the time spent in the association phase, using light-weight configuration protocols and avoiding the ones that require complex message exchange is recommended. To assist authentication, network support can be used to deliver configuration information [25, 22].

For the data-transmission phase, the first recommendation is to adopt an optimization technique tailored for the characteristics of different types of traffic and thus maximizing the transmission throughput during the offloading period. For instance, data batching and energy-efficient scheduling of delay-tolerant traffic can effectively amortize the penalty caused by tail energy, as we discussed earlier. Plenty of optimization schemes have been proposed in the literature [16, 33, 18, 19, 10, 12, 34, 35]. The second recommendation is to avoid using unstable or low throughput wireless links by estimating the network condition and user mobility. Several techniques related to this have been proposed in the literature [25, 26, 33, 36].

Since the offloading termination affects the data transmission and service quality, we can achieve energy saving by using proactive approaches to plan the termination and apply efficient handover schemes to ensure service continuity. Examples of such approaches are presented in [16, 25, 21, 22, 23, 24]. To avoid the frequent unnecessary terminations that degrade the transmission performance, using the knowledge from the network side (e.g., network setup and position of APs) combined with mobility prediction techniques is recommended.

13.6.4 Conclusion and outlook

Mobile traffic offloading provides a promising way to alleviate the pressure on existing cellular networks overloaded by data traffic and it is gaining support due to the rapid growth of mobile data traffic. By making the offloading process energy aware, it is possible to improve the user experience by extending the smartphone battery life.

Some open problems in traffic offloading remain and we believe they merit further investigation. First, how to use available resources on mobile devices (e.g., sensors) and particularly how to benefit from the support from the network side? Second, how to enable effective collaboration between peer mobile users and mobile networks, including cellular and, for instance, Wi-Fi providers? Third, how to extend the traffic-offloading concept to also encompass OnLoading, opportunistic onloading of traffic to, not from, cellular network, which has recently been shown to be beneficial in certain scenarios [37, 38]?

13.7 Summary

In this chapter, we discussed the possibilities for using the multiple WNIs of the smartphone in a clever way increase the energy efficiency. We discovered that the following sets of techniques can be applied to that end:

- *Assisted Wi-Fi AP discovery* can help save a significant amount of energy by reducing unnecessary scanning by the smartphone. We identified two kinds of technique which both have the same objective: to trigger Wi-Fi scanning only when it is possible or even likely that a new AP is in range. The trick is to use other wireless communication technologies that consume less energy than Wi-Fi or that will be powered on anyway (i.e., cellular network). The first approach is to track the movement of the smartphone user, by monitoring the set of overheard cellular network base stations, for instance, and to trigger a scan only if the user has moved to a new and significantly different location. The second approach is to use other wireless communication technologies integrated within the smartphone. For example, Bluetooth contact patterns can be used to learn the locations of APs and to use that information afterwards to trigger the scanning.
- *Using wake-up radios* is a concept where a low-power radio is kept powered on and other high-power consuming radios are completely switched off during periods of no communication. Notifications are then received by the low-power radio when higher-power radios should be turned on to receive a call or incoming data traffic. The downside of these solutions is that a specific notification service, often deployed in proxies, is necessary. In addition, the idle power drawn by the WNIs is becoming increasingly optimized, which may render these kinds of solution unnecessary.
- *Energy-aware selection of WNI for data transmission* has the potential to save energy because of the differing characteristics, especially in terms of energy utility of data transfer, of the wireless communication technologies deployed in smartphones. The idea is simply to select the most energy-efficient technology available for transmitting or receiving a given amount of data. One specific challenge in this selection process is the contextual dependency of the energy efficiency of wireless data communication, which we discussed in the previous chapter. This dependency implies that the selection policies cannot always be static but must measure or predict the context at runtime and take that into account.

References

[1] H. Wu, K. Tan, J. Liu, and Y. Zhang, "Footprint: cellular assisted wi-fi ap discovery on mobile phones for energy saving," in *Proc. 4th ACM Int. Workshop on Experimental Evaluation and Characterization*. New York, NY, USA: ACM, 2009, pp. 67–76.

[2] K.-H. Kim, A. Min, D. Gupta, P. Mohapatra, and J. Singh, "Improving energy efficiency of wi-fi sensing on smartphones," in *Proc. IEEE INFOCOM, 2011*, pp. 2930–2938.

[3] R. Zhou, Y. Xiong, G. Xing, L. Sun, and J. Ma, "Zifi: wireless lan discovery via zigbee interference signatures," in *Proc. 16th Annu. Int. Conf. on Mobile Computing and Networking*. New York, NY, USA: ACM, 2010, pp. 49–60.

[4] G. Ananthanarayanan and I. Stoica, "Blue-fi: enhancing wi-fi performance using bluetooth signals," in *Proc. 7th Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2009, pp. 249–262.

[5] E. Shih, P. Bahl, and M. J. Sinclair, “Wake on wireless: an event driven energy saving strategy for battery operated devices,” in *Proc. 8th Annu. Int. Conf. on Mobile Computing and Networking*. New York, NY, USA: ACM, 2002, pp. 160–171.

[6] Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta, “Wireless wakeups revisited: energy management for voip over wi-fi smartphones,” in *Proc. 5th Int. Conf. on Mobile Systems, Applications and Services*. New York, NY, USA: ACM, 2007, pp. 179–191.

[7] T. Pering, Y. Agarwal, R. Gupta, and R. Want, “Coolspots: reducing the power consumption of wireless mobile devices with multiple radio interfaces,” in *MobiSys ’06: Proc. 4th Int. Conf. on Mobile Systems, Applications and Services*. New York, NY, USA: ACM, 2006, pp. 220–232.

[8] A. Rahmati and L. Zhong, “Context-for-wireless: context-sensitive energy-efficient wireless data transfer,” in *Proc. 5th Int. Conf. on Mobile Systems, Applications and Services*. New York, NY, USA: ACM, 2007, pp. 165–178.

[9] T. Jin, G. Noubir, and B. Sheng, “Wizi-cloud: Application-transparent dual zigbee-wifi radios for low power internet access,” in *Proceedings IEEE INFOCOM, 2011*, pp. 1593–1601.

[10] A. Y. Ding, P. Hui, M. Kojo, and S. Tarkoma, “Enabling energy-aware mobile data offloading for smartphones through vertical collaboration,” in *Proc. 2012 ACM Conf. on CoNEXT Student Workshop*, ser. CoNEXT Student ’12. New York, NY, USA: ACM, 2012, pp. 27–28. [Online]. Available: http://doi.acm.org/10.1145/2413247.2413264

[11] S. Liu and A. Striegel, “Casting doubts on the viability of wifi offloading,” in *Proc. 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*, ser. CellNet ’12. New York, NY, USA: ACM, 2012, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/2342468.2342475

[12] N. Ristanovic, J.-Y. Le Boudec, A. Chaintreau, and V. Erramilli, “Energy efficient offloading of 3g networks,” in *Proc. 2011 IEEE 8th Int. Conf. on Mobile Ad-Hoc and Sensor Systems*, ser. MASS ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 202–211. [Online]. Available: http://dx.doi.org/10.1109/MASS.2011.27

[13] B. Han, P. Hui, and A. Srinivasan, “Mobile data offloading in metropolitan area networks,” *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 14, no. 4, pp. 28–30, Nov. 2010. [Online]. Available: http://doi.acm.org/10.1145/1942268.1942279

[14] A. Aijaz, H. Aghvami, and M. Amani, “A survey on mobile data offloading: technical and business perspectives,” *IEEE Wireless Communications*, vol. 20, no. 2, pp. 104–112, 2013.

[15] J. Korhonen, T. Savolainen, A. Ding, and M. Kojo, “Toward network controlled ip traffic offloading,” *IEEE Commun. Mag.*, vol. 51, no. 3, pp. 96–102, 2013.

[16] A. Balasubramanian, R. Mahajan, and A. Venkataramani, “Augmenting mobile 3g using wifi,” in *Proc. 8th Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys ’10. New York, NY, USA: ACM, 2010, pp. 209–222. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814456

[17] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong, “Mobile data offloading: How much can wifi deliver?” in *Proc. 6th Int. Conf.*, ser. Co-NEXT ’10. New York, NY, USA: ACM, 2010, pp. 26:1–26:12. [Online]. Available: http://doi.acm.org/10.1145/1921168.1921203

[18] X. Hou, P. Deshpande, and S. Das, “Moving bits from 3g to metro-scale wifi for vehicular network access: An integrated transport layer solution,” in *2011 19th IEEE Int. Conf. on Network Protocols (ICNP)*, 2011, pp. 353–362.

[19] S. Dimatteo, P. Hui, B. Han, and V. Li, “Cellular traffic offloading through wifi networks,” in *2011 IEEE 8th Int. Conf. on Mobile Adhoc and Sensor Systems (MASS)*, 2011, pp. 192–201.

[20] P. Bahl and V. Padmanabhan, "Radar: an in-building rf-based user location and tracking system," in *INFOCOM 2000. Proc. 19th Annu. Joint Conf. of the IEEE Computer and Communications Societies*, vol. 2, 2000, pp. 775–784.

[21] V. Brik, A. Mishra, and S. Banerjee, "Eliminating handoff latencies in 802.11 wlans using multiple radios: Applications, experience, and evaluation," in *Proc. 5th ACM SIGCOMM Conf. on Internet Measurement*, ser. IMC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 27–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251086.1251113

[22] I. Ramani and S. Savage, "Syncscan: practical fast handoff for 802.11 infrastructure networks," in *INFOCOM 2005. Proc. 24th Annu. Joint Conf. of the IEEE Computer and Communications Societies*, vol. 1, 2005, pp. 675–684 vol. 1.

[23] R. Chakravorty, P. Vidales, K. Subramanian, I. Pratt, and J. Crowcroft, "Performance issues with vertical handovers - experiences from gprs cellular and wlan hot-spots integration," in *Proc. 2nd IEEE Int. Conf. on Pervasive Computing and Communications (PerCom'04)*, ser. PERCOM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 155–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977406.978694

[24] M. Shin, A. Mishra, and W. A. Arbaugh, "Improving the latency of 802.11 hand-offs using neighbor graphs," in *Proc. 2nd Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys '04. New York, NY, USA: ACM, 2004, pp. 70–83. [Online]. Available: http://doi.acm.org/10.1145/990064.990076

[25] P. Deshpande, A. Kashyap, C. Sung, and S. R. Das, "Predictive methods for improved vehicular wifi access," in *Proc. 7th Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys '09. New York, NY, USA: ACM, 2009, pp. 263–276. [Online]. Available: http://doi.acm.org/10.1145/1555816.1555843

[26] A. J. Nicholson and B. D. Noble, "Breadcrumbs: Forecasting mobile connectivity," in *Proc. 14th ACM Int. Conf. on Mobile Computing and Networking*, ser. MobiCom '08. New York, NY, USA: ACM, 2008, pp. 46–57. [Online]. Available: http://doi.acm.org/10.1145/1409944.1409952

[27] A. J. Nicholson, Y. Chawathe, M. Y. Chen, B. D. Noble, and D. Wetherall, "Improved access point selection," in *Proc. 4th Int. Conf. on Mobile Systems, Applications and Services*, ser. MobiSys '06. New York, NY, USA: ACM, 2006, pp. 233–245. [Online]. Available: http://doi.acm.org/10.1145/1134680.1134705

[28] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, K. Papagiannaki, J. Crowcroft, and D. Wetherall, "Staying online while mobile: The hidden costs," in *Proc. 9th ACM Conf. on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13. New York, NY, USA: ACM, 2013, pp. 315–320. [Online]. Available: http://doi.acm.org/10.1145/2535372.2535408

[29] P. Deshpande, X. Hou, and S. R. Das, "Performance comparison of 3g and metro-scale wifi for vehicular network access," in *Proc. 10th ACM SIGCOMM Conf. on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 301–307. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879180

[30] J. Eriksson, H. Balakrishnan, and S. Madden, "Cabernet: Vehicular content delivery using wifi," in *Proc. 14th ACM Int. Conf. on Mobile Computing and Networking*, ser. MobiCom '08. New York, NY, USA: ACM, 2008, pp. 199–210. [Online]. Available: http://doi.acm.org/10.1145/1409944.1409968

[31] J. Paek, J. Kim, and R. Govindan, "Energy-efficient rate-adaptive gps-based positioning for smartphones," in *Proc. 8th Int. Conf. on Mobile Systems, Applications, and Services*, ser.

MobiSys '10. New York, NY, USA: ACM, 2010, pp. 299–314. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814463

[32] M. B. Kjærgaard, S. Bhattacharya, H. Blunck, and P. Nurmi, "Energy-efficient trajectory tracking for mobile devices," in *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 307–320. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000025

[33] U. Shevade, Y.-C. Chen, L. Qiu, Y. Zhang, V. Chandar, M. K. Han, H. H. Song, and Y. Seung, "Enabling high-bandwidth vehicular content distribution," in *Proc. 6th Int. Conf.*, ser. Co-NEXT '10. New York, NY, USA: ACM, 2010, pp. 23:1–23:12. [Online]. Available: http://doi.acm.org/10.1145/1921168.1921199

[34] B. Han, P. Hui, V. S. A. Kumar, M. V. Marathe, J. Shao, and A. Srinivasan, "Mobile data offloading through opportunistic communications and social participation," *IEEE Trans. on Mobile Computing*, vol. 11, no. 5, pp. 821–834, May 2012. [Online]. Available: http://dx.doi.org/10.1109/TMC.2011.101

[35] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed mobile prefetching," in *Proc. 10th Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 155–168. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307651

[36] L. Song, U. Deshpande, U. Kozat, D. Kotz, and R. Jain, "Predictability of wlan mobility and its effects on bandwidth provisioning," in *INFOCOM 2006. Proc. 25th IEEE Int. Conf. on Computer Communications. Proc.*, 2006, pp. 1–13.

[37] C. Rossi, N. Vallina-Rodriguez, V. Erramilli, Y. Grunenberger, L. Gyarmati, N. Laoutaris, R. Stanojevic, K. Papagiannaki, and P. Rodriguez, "3gol: Power-boosting adsl using 3g onloading," in *Proc. 9th ACM Conf. on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13. New York, NY, USA: ACM, 2013, pp. 187–198. [Online]. Available: http://doi.acm.org/10.1145/2535372.2535400

[38] N. Vallina-Rodriguez, V. Erramilli, Y. Grunenberger, L. Gyarmati, N. Laoutaris, R. Stanojevic, and K. Papagiannaki, "When David Helps Goliath: The Case for 3G Onloading," in *Proc. 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI. New York, NY, USA: ACM, 2012, pp. 85–90. [Online]. Available: http://doi.acm.org/10.1145/2390231.2390246

14 Mobile cloud offloading

A seemingly straightforward way to conserve the battery life of a mobile device is to migrate application execution partially to a cloud. This technique is called offloading or sometimes cyber foraging. Offloading computationally-expensive processing from smartphones onto more powerful servers has been widely discussed in the literature. In this section, we introduce four frameworks (MAUI [1], ThinkAir [2], Cuckoo [3], and CloneCloud [4]) that allow mobile applications to dynamically migrate part of their execution from the smartphone to the cloud. These frameworks have been developed to offload CPU-intensive tasks. We call it computation offloading. Note that most popular mobile applications also involve network communications, which in fact consume a significant part of the overall battery life. Hence, it is justified to ask: Can offloading techniques help save energy, if applied to such applications? To answer this question, we also discuss communication offloading, a technique that focuses on reducing communication costs through offloading.

14.1 Computation offloading

Offloading computationally-expensive processing (such as speech recognition, face recognition, language translation, and 3D modeling) from smartphones onto more powerful servers has proved to be useful for improving performance and energy efficiency. Take face detection as an example, Simoens et al. [5] implemented a video-sharing application in which faces detected from the video captured by mobile devices were first blurred before the video was shared. The authors partitioned the program that face detection, a compute-intensive function, was offloaded from mobile devices to a nearby four-core server. According to their power measurement, processing one 1080p video frame using the face detector provided by Android SDK takes on average 3.2 seconds and costs 9.0 joules on a Samsung Galaxy Nexus I9250. This means a fully charged battery will be drained by continuous face detection within 3.5 hours. By offloading face detection to the remote server, the throughput of face detection increases 10 times, with the cost of face detection removed from the phone. Although uploading images from a smartphone to the remote server causes an overhead, the overhead is much smaller compared to the energy savings gained from offloading.

In the above example, the system was implemented following the client-server architecture. Developers manually divide the functionalities into two parts, one for local execution and the other for remote execution of compute-intensive function. The first part is implemented in the application client, while the other part is implemented in the application server. The client and server can be written in different programming languages and are not necessarily run on the same operating system. However, the client's code can only be executed locally and cannot be migrated to the server at runtime. In other words, what to run remotely is predefined and hardcoded.

Because there are already thousands of mobile applications available in application stores like Google Play, we ask the following questions. How to use the computing power in the cloud to improve the performance and energy efficiency of those applications that are originally designed for single-machine execution? Instead of re-implementing these applications as described in the above example, can we find a way to enable computation offloading with minimum work? Due to the potential network disconnection and communication overhead caused by data exchange between client and server, offloading is not always feasible and profitable. What would be needed to implement context-aware dynamic offloading?

In this section, we discuss the energy tradeoff of computation offloading, and introduce four representative offloading frameworks, including MAUI [1], ThinkAir [2], Cuckoo [3], and CloneCloud [4]. These frameworks enable fine-grained offloading from a smartphone to its device clone in the cloud. A device clone can be a virtual machine of a complete smartphone system, such as an Android emulator for an Android phone. So far none of the existing frameworks can work across Windows, Android, and iOS. MAUI supports only applications written for the Microsoft .NET Common Language Runtime (CLR), while ThinkAir, Cuckoo, and CloneCloud are designed for Java Applications written for Android. Although these frameworks apply to different application-level virtual machines, the work flow they follow to implement offloading is still similar. Except CloneCloud, the other three frameworks require modification to the source code of mobile applications. To make it easy to explain, we first describe the workflow using MAUI-like frameworks as examples, and then give a description of CloneCloud in a separate section.

14.1.1 Computation offloading energy considerations

To analyze the energy benefit of offloading, we need to determine the energy components that comprise offloading. The offloading of a mobile computing task is a tradeoff between the energy used for local processing and the energy required for offloading the task, uploading its data, and downloading the result, if necessary. The offloading energy tradeoff can be expressed by the formula

$$E_{trade} = E_{local} - E_{delegate} > 0,$$

where E_{local} is the energy used for complete local execution and $E_{delegate}$ is the energy used from the perspective of the mobile device if the task is offloaded. If E_{trade} is greater than zero, then there is an energy benefit in delegating the task to the cloud. We can break down E_{local} into the power used for local execution P_x, and the time taken, T_x:

$$E_{local} = P_x \times T_x.$$

$E_{delegate}$, the cost of delegating the task to the cloud, combines the energy spent sending the task and data to the cloud, E_s, idly waiting for the cloud to complete the task, E_i, and receiving the result of the task, E_r:

$$E_{delegate} = E_s + E_i + E_r.$$

Since $E = P \times T$, each energy variable can be broken down to the corresponding average power, P, and time duration, T. This is shown below:

$$E_{delegate} = P_s \times T_s + P_i \times T_i + P_r \times T_r.$$

In an optimal case, T_i could be minimized by spending the idle time performing other scheduled tasks, but this may not always be the case. The energy spent for data transfers could also be mitigated by scheduling offloading to coincide with other tasks that require network connectivity, such as VoIP or SIP calls. Here we consider the unmitigated case.

Finally, when we take into account the amount of data to be sent and received, D_s and D_r, and the sending and reception bandwidths, B_s and B_r, the trade-off formula becomes

$$E_{trade} = P_x \times T_x - P_i \times T_i - P_s \frac{D_s}{B_s} - P_r \frac{D_r}{B_r} > 0$$

where $P_x \times T_x$, $P_i \times T_i$, P_s, and P_r depend on the local and cloud platforms. It can now be seen that a higher local execution time and a larger gap between local execution power and idle power increases the benefits of offloading the task, while a smaller bandwidth and higher amount of data to be transferred result in offloading becoming less attractive.

Note that the network and data transfer latency and other delays are included in the bandwidth in this formula. We do not consider such delays separately.

Notice that the tradeoff is positive when

$$P_x \times T_x > P_i \times T_i + P_s \frac{D_s}{B_s} + P_r \frac{D_r}{B_r}$$

or

$$P_x \times T_x > P_i \times T_i + \frac{D_s}{B_s} P_s \tag{14.1}$$

if no data needs to be received by the mobile device, and

$$P_x \times T_x > \frac{D_s}{B_s} P_s \tag{14.2}$$

if we assume the idle time is negligible or that the cloud platform is fast compared to the mobile device.

In summary, when considering offloading a computing task, the main factors are the time spent executing the task locally and the time taken sending the task and its data to the cloud and receiving the result. The cloud platform is often hundreds of times faster than the mobile device, which makes the time waiting for the result negligible. In good network conditions, offloading is often preferable. Tasks with small data requirements can be offloaded in worse network conditions as well.

14.1.2 Constraints for remote execution

The first step of code offloading is to identify the partitions that can be executed. There are two types of constraints that prevent code from being executed remotely. We call them hardware constraints and software constraints. Hardware constraints refer to the code that require access to the hardware of the local device. It can be the code that reads data from on-device sensors such as GPS receivers, accelerometers, and the compass, or the code that implement the user interface, such as getting input from keyboards, showing text messages on the display, and vibrating the phone. Saarinen et al. [6] went through the Android system APIs and identified a set of 20 constrained subsystems. If a method accesses one of the subsystems included in the set, the method is considered to be non-offloadable.

Software constraints refer to the code partitions that cannot be migrated to the cloud at all due to the requirement of the offloading framework in use, or the partitions that cause unexpected behavior when executed remotely due to inconsistent states between local and remote execution environments. The framework-specific requirement is usually related to serialization, callbacks, and stateless issues. Take ThinkAir as example, if the encapsulating class of a method is not serializable using Java's serialization APIs, or if a method accesses a state outside the serialized context, the method in question is considered to be non-offloadable. Similar constraint can be found in Cuckoo. Some methods could be modified to be serializable with minor changes, but in practice not all the code can be easily modified to implement the serializable interface.

Regarding unexpected behavior, Saarinen et al. [6] give an example scenario where ThinkAir does not synchronize the file system between a smartphone and its device clone in the cloud, and when the program tries to access files, unexpected behavior may happen. A potential solution to the synchronization issue is a system which automatically synchronizes all relevant states. This is notably a hard problem on its own. Until efficient automatic solutions are presented, the developer must be assisted in overcoming the problems manually.

At the moment most offloading frameworks, including MAUI, ThinkAir, and Cuckoo, require developers to manually go through the source code and check if the methods or classes are constrained from offloading. This procedure is non-trivial. It may include many cycles of trial-and-error and the only way to know whether all issues have been resolved is to test the application to see whether it works correctly or not. It is essential to have tools that can automate this procedure and guide the programmers into developing more remotable code. An example of such tools is SmartDiet [6], a static source-code

```
//original interface

public interface IEnemy
[remotable]bool SelectEnemy(int x, int y);
[Remotable]void ShowHistory();
void UpdateGUI();
```

```
//remote service interface

public interface IEnemyService
MAUIMessage<Appstate, bool> SelectEnemy (AppState state,
int x, int y);
MAUIMessage<AppState, MauiVoid> ShowHistory(AppState
state);
```

Figure 14.1 Example annotation of remotable methods using MAUI [1]

analyzer which identifies the methods that cannot be offloaded using ThinkAir. Moreover, SmartDiet can suggest minor changes that can be made to the code to allow more methods to become remotable.

14.1.3 Program partitioning and execution runtime

The previous section explains how to work out whether the methods or classes are offloadable or not. Offloading frameworks, including MAUI [1], ThinkAir [2], and Cuckoo [3], require developers to manually annotate these remotable methods or classes in the source code, following the framework-specific rules. The annotations will be automatically identified by the framework during runtime. Take MAUI as an example, it uses the custom attribute feature of the Microsoft .NET CLR for identifying remotable methods. In practice, developers simply add the *[Remoteable]* attribute to each method that can be executed remotely. An example can be found in Figure 14.1.

The MAUI framework automatically generates a wrapper for each method with the remotable attribute. Compared with the original method, the wrapper adds an additional input parameter and an additional return value. These two values are used for transferring the state between the mobile device and the cloud. At compile time, the source code of the Microsoft .NET CLR applications is translated into Common Intermediate Language (CIL) and assembled into an object code that is not platform or processor-specific.[1] The object code is dynamically converted into native code by CLR's just-in-time compiler at runtime. Note that the executables that contains the object code must be available on both the smartphone and the MAUI server before the application starts.

As shown in Figure 14.2, when the remotable methods or classes are triggered to run remotely, the framework can take care of the necessary code, data, and program state migration between mobile devices and the cloud.

In summary, the offloading frameworks provide at least the following features:

- automatically identifying which methods or classes have been marked as remotable,
- supporting the execution of the same program on different CPU architectures, if the smartphone uses a different CPU architecture to the remote surrogate, and

[1] CIL is a CPU-and platform-independent instruction set that can be executed in any environment supporting the Common Language Infrastructure (CLI), such as the .NET runtime on Windows.

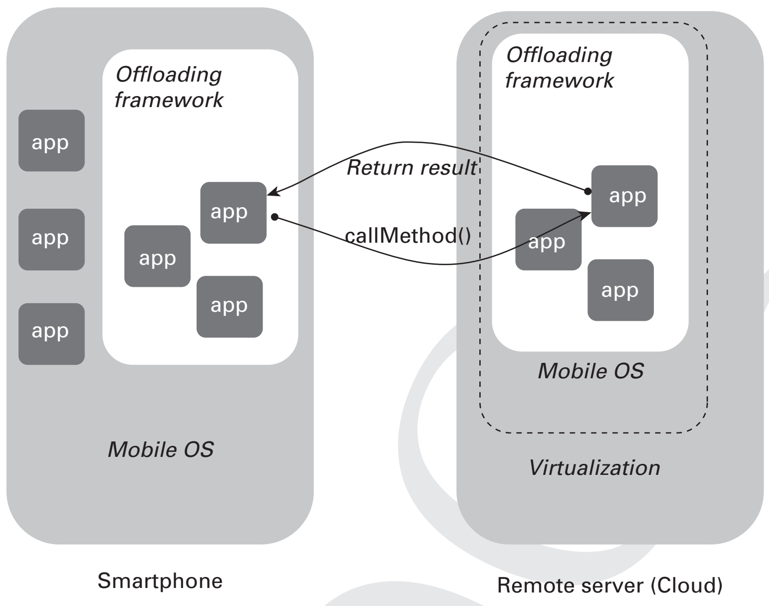


Figure 14.2 System architecture of computation offloading

- automatically transferring code, data, and the state of the running program between smartphone and the cloud.

Similar to MAUI, ThinkAir [2] requires developers to annotate the remotable methods with *@Remote*. At compile time, the ThinkAir Remote Code Generator generates wrappers for the remotable methods. As most mobile devices are ARM-based but not x86-based, ThinkAir also provides a customized NDK to support the execution of native methods on the cloud. At runtime, if the Execution Controller decides to offload the execution of a remotable method, the Java Reflection feature is used and the calling object is sent to the device clone in the cloud. The smartphone waits for the execution results and the modified local state to be returned. If the remote execution fails, the ThinkAir framework falls back to local execution.

Cuckoo [3] uses the activity/service model in Android for identification. Developers define the compute-intensive parts to be offloaded as services, using an interface definition language (AIDL). The interactive parts of the applications are implemented as activities. For each local service implemented by the programmer, Cuckoo generates a dummy method implementation of the same interface for a remote service. After that, developers must replace the dummy method implementation with a real method implementation that will be executed remotely. At runtime, activities can communicate with services through Inter Process Communication, using the predefined interface and a stub/proxy pair generated by the Android Pre Compiler.

CloneCloud [4] takes this a step further by automating the application partitioning without modifying the application. For CloneCloud, developers do not need to write code in a specific way and partitioning can be done directly for the application executable instead of the source code. More details of CloneCloud can be found in Section 14.1.5.

14.1.4 Runtime context profiling and decision-making

After discovering what can be offloaded (see Section 14.1.2) and making the necessary modifications to the source code following the requirement of the offloading framework (see Section 14.1.3), the next step is to determine whether it is profitable in terms of performance and energy efficiency to execute the offloading. Offloading can reduce the computational cost on the mobile device. There is a caveat, though. The operations of offloading cause extra communication between the mobile device and the remote system, which also consumes energy. Only when the reduction in computational cost overtakes the increase in communication cost can the offloading achieve energy savings.

The timing constraints and the complexity of offloading should also be taken into account when deciding whether to offload. Taking timing constraints as an example, the added delay caused by exchanging data should be less than the reduction in computation duration (offloading to a more powerful machine). Otherwise, users might experience the offloading as a slowdown of the application.

Some offloading frameworks, like MAUI, provide profilers for continuously monitoring the runtime system and the environment. Based on the profiling information, the frameworks can estimate the energy costs based on predefined power models. As offloading can be implemented at various granularity levels, we can accordingly decompose the software into functional units at the corresponding granularity level and then apply appropriate models to estimate the energy cost of each functional unit. The computational cost of each software functional unit can be modeled statistically based on power measurement or be derived from usage-based power models. For example, the profiler included in MAUI estimates the CPU cost based on a linear power model using least-squares linear regression. The linear model predicts the power consumption as a function of the number of CPU cycles required to execute the method in question.

The transmission cost can be estimated using the power models based on traffic statistics, such as the number of bytes transferred and the network throughput. It is important to understand that these models are dependent on the network conditions, which can be described using metrics such as round-trip time, SNR, and the available network interfaces (e.g. 3G or Wi-Fi). Such metrics need to be continuously monitored and they may have a heavy impact on whether offloading a given piece of code is profitable or not from the perspective of energy consumption. For MAUI, for each method, the profiler measures the runtime duration, the CPU cycles required, the size of data potentially referenced by the method, and the amount of traffic caused by state transfer. The last two metrics are used for estimating the transmission overhead.

14.1.5 CloneCloud

To lower the burden on programmers, Chun et al. proposed a system called CloneCloud [4] which enables automatic application partitioning without modifications to the source code. Given an application executable, a component of CloneCloud called the static analyzer first identifies possible migrating points in the code according to a set of constraints. For example, migration can only be triggered at the boundaries of application methods. The migrated methods are allowed to invoke native methods, whereas the migration of native methods and state is not allowed. To avoid nested migration, the static analyzer builds a static control-flow graph which describes the caller–callee method relation of the application. If a migrating point is placed at the entry point of a method, no more migrating points can be placed before the re-integration point at the exit of this method.

Which migrating points to choose is pre-computed by a standard integer linear programming solver that tries to minimize the expected cost of the partitioned application. The solver takes the execution conditions as input, and outputs a partition configuration that shows the chosen migration and re-integration points in the code. It estimates the cost of execution time and energy consumption based on predefined cost models. These models are constructed from the data collected by the dynamic profiler while running the application under different execution settings and conditions. For example, an execution is repeated on the smartphone and the device clone. For each run, a profile tree is generated, with one node corresponding to a method and the edge between nodes corresponding to the measured cost.

At runtime, the executing thread is suspended at a chosen migrating point and its state, including the virtual state, program counter, registers, and stack, is packaged and shipped to the device clone in the cloud. A new thread is then initiated with the state in the clone. When the migrated thread is completed, its state is packaged and shipped in the same way back to the original process running on the smartphone. In practice, the operations related to suspending/resuming, packaging and state synchronization are taken care of by a per-process migrator thread, while a per-node node manager is responsible for the communication between the smartphone and the clone.

14.2 Communication offloading

In this section we introduce the concept of communication offloading. Similarly to computation offloading, communication offloading tries to save energy by offloading part of the program execution to the cloud. The frameworks used for computation offloading can also be used for implementing communication offloading. In fact, the examples of communication offloading we give in this section were implemented using ThinkAir [2]. Different to computation offloading, communication offloading focuses on reducing the communication cost but not the computational cost on mobile devices.

14.2.1 Can communication offloading save energy?

There are two ways to save energy when offloading methods responsible for communication-related tasks. The first way is to reduce the network traffic that needs to be handled by the mobile device. This can be achieved, for instance, by offloading

Table 14.1. AndTweet measurements for a single timeline refresh event

Metric	Wi-Fi		3G	
	Original	Offloaded	Original	Offloaded
Energy(J)	2.67±0.59	2.00±0.3	3.92±1.97	4.63±2.1
Execution time(s)	2.69±0.59	2.85±0.5	3.86±2.02	5.13±2.1
Traffic size(KB)	7.7±0.8	6.39±0.5	6.0±2.1	4.14±1.8

methods that handle communication with a server or other peers in a P2P system. Such communication contains signaling traffic [7], part of which can be suppressed. The second way is to optimize traffic patterns and improve overall latency and/or throughput. Packet interval patterns and throughput, for instance, have a significant impact on communication energy cost [8]. As a consequence, grouping packets into bursts is more energy efficient. Bursting benefits can be achieved, for example, when a group of methods, where each method fetches data, is offloaded. In the resulting sequence, only the end result is downloaded to the mobile device in one burst, instead of downloading the data in several small bursts for each method.

We chose AndTweet[2], an open-sourced Android twitter client, for a case study. The aim of our study is to investigate the potential energy savings through communication offloading. We first manually look for all places, which induce network traffic between the mobile device and Twitter servers, and mark all these methods as “Remotable” except the ones restricted by the hardware and software constraints described in Section 14.1.2. Our intention is to reduce the amount of traffic to/from the smartphone as much as possible.

We then compare the energy consumption between the original and the offloadable AndTweet, under two different network conditions. In the first setup the surrogate is in the same Wi-Fi network as the phone. In the second scenario, the phone used 3G as the access network. We use a Monsoon Power Monitor (www.msoon.com) to measure the energy consumption of the phone, and exclude the one-time cost of transferring the application image. The applications could be, for example, pre-installed to the offloading infrastructure.

The results in Table 14.1 show the energy consumption of a single Twitter event which checks and fetches new tweets, and then displays them. According to the results, offloading saves one-fourth of the total energy consumed in the Wi-Fi setup. As expected, the savings clearly come from having less network traffic. However, the execution takes slightly longer when offloaded, which increases the energy consumed by the display. The results are very different when switching to 3G. More energy is consumed with the offloaded version even if less data is transmitted and received. Because the RTT in 3G is an order of magnitude longer than with Wi-Fi, the remote invocation takes more time. Combined with long 3G inactivity timer values, this causes the

[2] `https://code.google.com/p/andtweet/` accessed January 6, 2014.

network interface to be in the active high-power state during the whole invocation. Execution time also has a big variance, because 3G latencies vary depending on network conditions and the radio connection state when starting the transmission.

From the results we can see that communication offloading can yield notable benefits under certain network conditions. Both network conditions and traffic patterns play a major role in the profitability of offloading. We must also point out that identifying remotable methods that could bring notable benefits is non-trivial. The estimation of potential benefits is difficult using mere intuition.

14.2.2 SmartDiet: Toolkit for constraint analysis and energy profiling

As mentioned in Section 14.1.2, SmartDiet provides constraint analysis based on static source-code analysis. In addition to that, SmartDiet also implements a measuring and modeling setup for profiling and visualizing the energy consumption of a given application. It can show how much communication cost is consumed by each part of the application. We first explain how this tool works and then show examples of the analysis results provided by the tool.

SmartDiet collects data during the application execution and analyzes it afterwards. It collects two kinds of information: the traffic trace and a trace of the program execution flow to later produce class- and method-level statistics. A packet trace is collected by a kernel module that captures and timestamps all traffic in the device using netfilter hooks (www.netfilter.org). To track the program execution, the tool uses execution tracking features in the Android Debug Monitor Server (DDMS), which produces a trace of all classes and methods executed during the run. In addition, the tool can annotate the traces with system-wide timestamps. These timestamps can be used for matching the program execution traces with packet traces and power consumption readings.

SmartDiet provides method-level communication energy accounting based on predefined power models. It first associates each collected packet with an individual method in the program execution trace. In detail, the tool divides the execution trace into threads and the packet trace into separate flows (TCP connection or UDP flow). Note that only network-related method calls are filtered for each thread. After this step, two separate time series are generated: the network-related method calls of each thread and the packet arrival events of each flow. These series are compared by computing cross-correlations to associate each flow with a particular thread which is generating that traffic. The idea is that each network-related method is associated with the corresponding packets. In practice, each packet of a flow is associated with the closest (in time) method call of the corresponding thread. This way, the tool generates a method trace of the program execution annotated with information about the methods that caused network traffic.

Figure 14.3 shows part of the traces of a simple Android test application that performs an HTTP GET request when a button is clicked. The thread executing the HTTP request correlates strongly with the packet trace. Another thread in the figure has a single network-related call. This is the garbage collector thread running finalization for a network-related object that is no longer used. Since it correlates weakly with the packet trace, no packets are associated with it. Program execution in each thread can be viewed

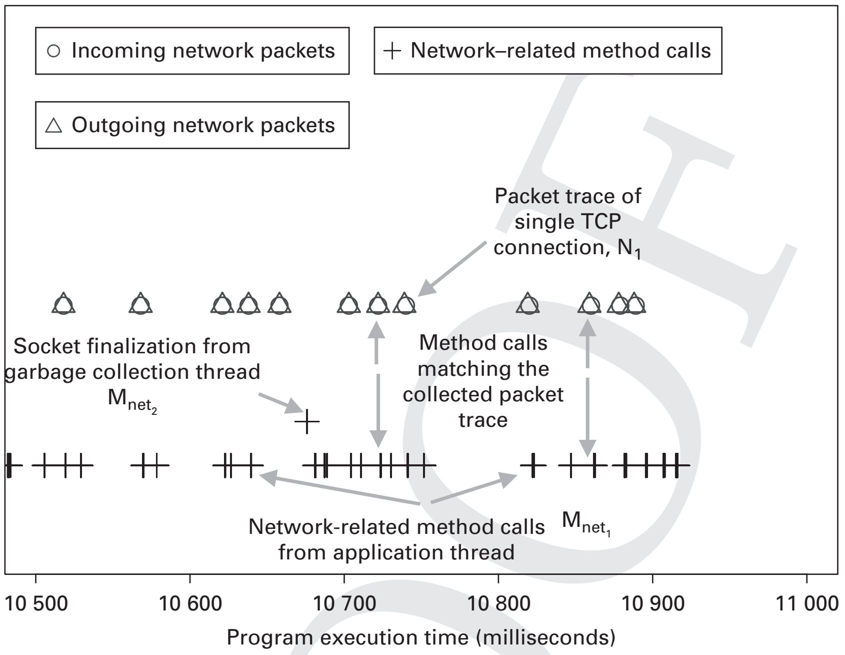


Figure 14.3 TCP packets and network-related method calls for a test application fetching an HTML page

as a hierarchical call tree, where a method calls another method which calls another and so on. SmartDiet reconstructs this tree, carrying along the information of the detected network usage. It then aggregates the traffic of the nodes up in the tree, so that the root method, where the execution starts, is associated with all packets that have been sent or received within each thread.

Given a method and its associated packets, SmartDiet uses the deterministic power models described in [8] to estimate the communication cost. After that, the tool recursively sums up the energy consumed by these methods as the energy consumed by their parent methods. In this way, we end up with a complete execution trace with the communication energy consumed by each of the methods. Figure 14.4 is an example graph, automatically produced by SmartDiet, for a simple test case requesting an HTML document over HTTP. Traffic is cumulatively assigned to the MainActivity.onClick method and from there on, divided between various library functions that open the connection, send an HTTP request, receive the response, and finally close the connection. At each step, the following metrics are updated: the total number of calls made, the number of packets and the amount of data generated by each call, and the model-based energy consumption estimates. Numbers show how many times methods have been invoked during the whole procedure and how much energy they consumed.

The energy usage estimate for each method is shown as a range between two values. In estimation, assumptions need to be made about the interdependencies of methods within the program, which is why there are two numbers: a lower bound and an upper bound. This is because communication energy consumption is heavily dependent on

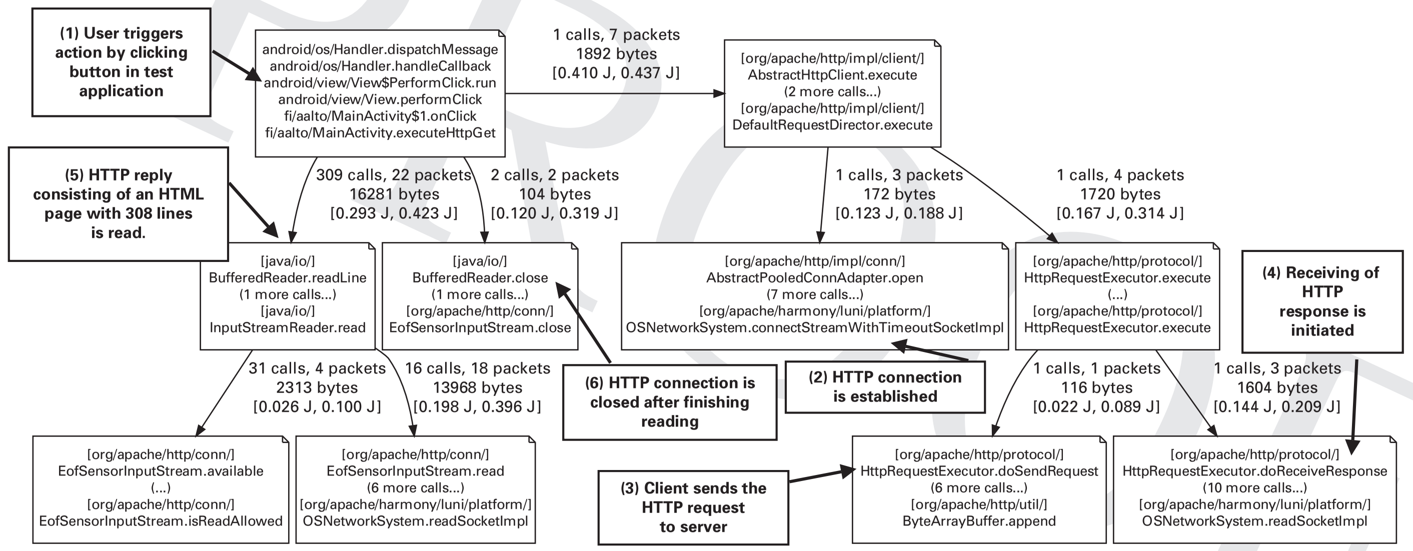

(1) User triggers action by clicking button in test application
android/os/Handler.dispatchMessage
android/os/Handler.handleCallback
android/view/View$PerformClick.run
android/view/View.performClick
fi/aalto/MainActivity$1.onClick
fi/aalto/MainActivity.executeHttpGet
1 calls, 7 packets
1892 bytes
[0.410 J, 0.437 J]
[org/apache/http/impl/client/]
AbstractHttpClient.execute
(2 more calls...)
[org/apache/http/impl/client/]
DefaultRequestDirector.execute
(5) HTTP reply consisting of an HTML page with 308 lines is read.
309 calls, 22 packets
16281 bytes
[0.293 J, 0.423 J]
2 calls, 2 packets
104 bytes
[0.120 J, 0.319 J]
1 calls, 3 packets
172 bytes
[0.123 J, 0.188 J]
1 calls, 4 packets
1720 bytes
[0.167 J, 0.314 J]
[java/io/]
BufferedReader.readLine
(1 more calls...)
[java/io/]
InputStreamReader.read
[java/io/]
BufferedReader.close
(1 more calls...)
[org/apache/http/conn/]
EofSensorInputStream.close
[org/apache/http/impl/conn/]
AbstractPooledConnAdapter.open
(7 more calls...)
[org/apache/harmony/luni/platform/]
OSNetworkSystem.connectStreamWithTimeoutSocketImpl
[org/apache/http/protocol/]
HttpRequestExecutor.execute
(...)
[org/apache/http/protocol/]
HttpRequestExecutor.execute
(4) Receiving of HTTP response is initiated
31 calls, 4 packets
2313 bytes
[0.026 J, 0.100 J]
16 calls, 18 packets
13968 bytes
[0.198 J, 0.396 J]
(6) HTTP connection is closed after finishing reading
(2) HTTP connection is established
1 calls, 1 packets
116 bytes
[0.022 J, 0.089 J]
1 calls, 3 packets
1604 bytes
[0.144 J, 0.209 J]
[org/apache/http/conn/]
EofSensorInputStream.available
(...)
[org/apache/http/conn/]
EofSensorInputStream.isReadAllowed
[org/apache/http/conn/]
EofSensorInputStream.read
(6 more calls...)
[org/apache/harmony/luni/platform/]
OSNetworkSystem.readSocketImpl
(3) Client sends the HTTP request to server
[org/apache/http/protocol/]
HttpRequestExecutor.doSendRequest
(6 more calls...)
[org/apache/http/util/]
ByteArrayBuffer.append
[org/apache/http/protocol/]
HttpRequestExecutor.doReceiveResponse
(10 more calls...)
[org/apache/harmony/luni/platform/]
OSNetworkSystem.readSocketImpl

Figure 14.4 Network usage graph for the test application, which fetches an HTML page

traffic patterns, meaning that the energy consumed is determined by the exact number of bits transmitted and their timing (refer to [8], for instance). The lower bound corresponds to the energy consumed by the packets associated with the method in question, while the upper bound is computed so that it also includes the packets belonging to other methods that arrive between the first and the last packet of this method.

14.3 Where to offload: Centralized cloud vs. distributed CLOUD

With smartphones evolving into cognitive phones, we can envision the emergence of cognition-based applications that would entirely transform the ecosystem of computing, communications, and human interaction. Compared with popular applications like Facebook and Twitter, the emerging applications would demand more computing and networking resources. Moreover, most of these applications include human interaction and therefore are latency-sensitive.

The public clouds of today have been mainly designed for enterprise applications, without considering the needs of mobile applications. The cloud infrastructure has been designed in such a way that the dispersed computing capacity is consolidated into a few large data centers. The centralization exploits the economics of scale to lower the marginal cost of system administration and operations. However, the centralization also often implies a large distance between the mobile devices and the cloud, which poses challenges to the end-to-end networking performance in mobile cloud computing. One example of these challenges is the potentially high latencies caused by the multiple networking hops between the mobile device and the cloud. Therefore, the current cloud infrastructure may not satisfy the networking performance requirements of the emerging mobile applications.

Much effort has been invested in optimizing the current cloud architecture for mobile cloud applications. A change in the design of cloud infrastructure we can envision is moving from centralization to ubiquitous so that much of the processing will take place as close as possible to where the data is captured. In practice, the existing centralized large data centers can be enhanced with distributed small data centers that are much closer to mobile devices. A small data center can be a private cloud owned by a business or community, or a small data center such as Myoonet's Micro data center that is deployed by a cloud operator. We refer to such small data centers as Cloudlets [9].

Mobile devices may access cloudlets via Wi-Fi, or any of the current or emerging cellular technologies. In practice, it is suggested that cloudlets are deployed in wireless access networks, with only one hop away from mobile devices. The computing model in ubiquitous cloud computing is derived from the computation offloading model that was originally proposed for saving energy. The most demanding computational tasks will be the first to be offloaded from mobile devices to cloudlets. In case the computation is too heavy to be carried out on a single cloudlet, the computing tasks may be further partitioned between cloudlets and the public clouds.

Here we give an example scenario of cloudlet-based personalized cloud computing. As shown in Figure 14.5, the first layer is composed of the mobile devices, whose roles

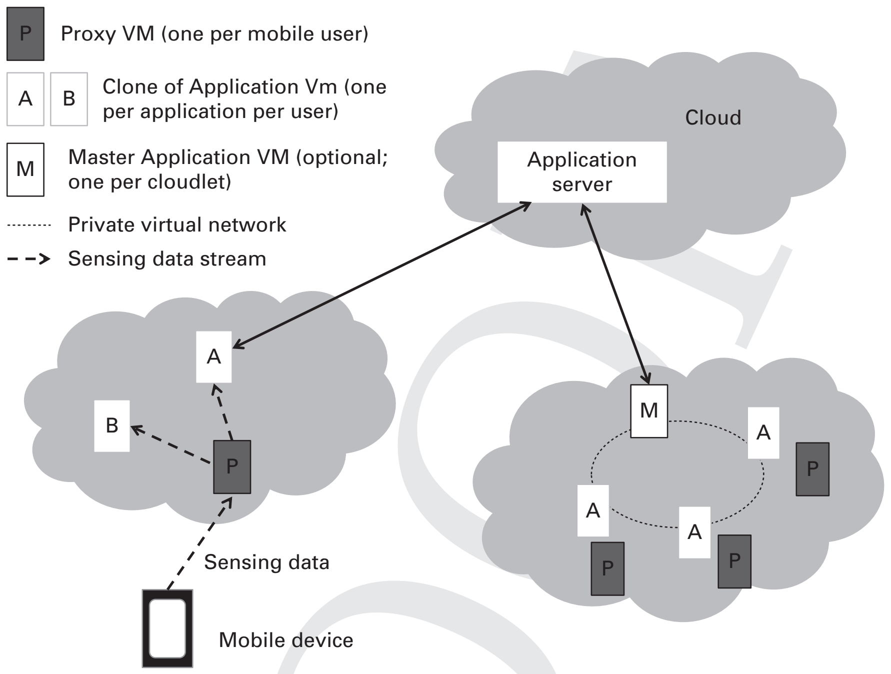

P
Proxy VM (one per mobile user)
A B
Clone of Application Vm (one per application per user)
M
Master Application VM (optional; one per cloudlet)
Private virtual network
Sensing data stream
Cloud
Application server
A
B
P
M
A
P
A
A
P
P
Sensing data
Mobile device

Figure 14.5 System architecture of cloudlet-based personalized cloud computing [10]

are essentially reduced to that of (multi-input) sensors forwarding captured data to *proxy VMs* in the second layer. The second layer comprises distributed cloudlets.

Each proxy VM is associated with a single smartphone, and is kept physically close to the user through VM migration to other cloudlets or public clouds. This ensures that the network resources to transfer data from the mobile device is minimized. It is essentially an extension of the mobile device into the cloud. It can be a clone of the smartphone OS, or just a virtual machine that handles all the requests for sensor data on behalf of the mobile device. Computation-intensive data preprocessing can be offloaded from the smartphone to the proxy VM. From here, data is forwarded to one or more *application VMs* also running on the cloudlet infrastructure.

Each application VM hosts a single application, which is not customized to any particular mobile platform. Generally, one application VM is assigned to each participant, making it easy to migrate a user's proxy VM together with the associated application VMs, preserving any hard state they may contain. If an application does not need to maintain a hard state for each user, then a single application VM can be shared by all the users on a particular cloudlet.

14.4 Summary

In this chapter we explained the concepts of computation offloading and communication offloading, and introduced the workflow of offloading frameworks including

MAUI, ThinkAir, Cuckoo, and CloneCloud. Given an application, part of it cannot be offloaded due to constraints such as access to local hardware devices and Java serialization issues. We introduced a toolkit called SmartDiet that can help developers identify the constraints automatically.

For the part of application that can be executed remotely, whether to offload it at runtime depends on the tradeoff between the potential energy savings and the offloading-induced overhead (e.g. cost of transmitting code and data). Performance, like latency, is also an important factor that should be taken into account while deciding whether to offload and where to offload. In the last section, we showcased a scenario where the clones of the smartphone are deployed on distributed cloud infrastructure instead of a centralized one. Compared with a centralized cloud infrastructure, the distributed cloud infrastructure that provides computing and storage at the edge can provide lower latency and can save bandwidth between the access network and the core network. However, the operational cost would increase and maintaining the QoS in mobility scenarios would be more challenging.

References

[1] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” in *Proc. 8th Int. Conf. on Mobile Systems, Applications, and Services (MobiSys ’10)*. New York, NY, USA: ACM, 2010, pp. 49–62.

[2] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *Proc. IEEE INFOCOM, 2012*, 2012, pp. 945–953.

[3] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, “Cuckoo: A computation offloading framework for smartphones,” in *2nd Int. ICST Conf. Mobile Computing, Applications, and Services, MobiCASE 2010*, 10 2012.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: elastic execution between mobile device and cloud,” in *Proc. 6th Conf. on Computer Systems*. New York, NY, USA: ACM, 2011, pp. 301–314.

[5] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan, “Scalable crowdsourcing of video from mobile devices,” in *Proc. 11th Annu. Int. Conf. on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2013, pp. 139–152. [Online]. Available: http://doi.acm.org/10.1145/2462456.2464440

[6] A. Saarinen, M. Siekkinen, Y. Xiao, J. K. Nurminen, M. Kemppainen, and P. Hui, “Can offloading save energy for popular apps?” in *Proc. 7th ACM Int. Workshop on Mobility in the Evolving Internet Architecture*. New York, NY, USA: ACM, 2012, pp. 3–10. [Online]. Available: http://doi.acm.org/10.1145/2348676.2348680

[7] ——, “Offloadable apps using smartdiet: Towards an analysis toolkit for mobile application developers,” *CoRR*, vol. abs/1111.3806, 2011.

[8] Y. Xiao, Y. Cui, P. Savolainen, M. Siekkinen, A. Wang, L. Yang, A. Yla-Jaaski, and S. Tarkoma, “Modeling energy consumption of data transmission over Wi-Fi,” *IEEE Trans. on Mobile Computing*, vol. 99, no. PrePrints, 2013.

[9] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for vm-based cloudlets in mobile computing,” *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1109/MPRV.2009.82

[10] Y. Xiao, P. Simoens, P. Pillai, K. Ha, and M. Satyanarayanan, “Lowering the barriers to large-scale mobile crowdsensing,” in *Proc. 14th Workshop on Mobile Computing Systems and Applications*. New York, NY, USA: ACM, 2013, pp. 9:1–9:6. [Online]. Available: http://doi.acm.org/10.1145/2444776.2444789

15 Example scenarios for energy optimization

We now turn our attention from individual optimization techniques to applications. We investigate a few different cases where the principles and techniques we have learned earlier can be applied in mobile applications.

We first look at a specific application, namely video streaming which is one of the most important Internet applications today, from the mobile Internet's perspective. We first study the way that video streaming consumes energy and illustrate that through measurement results from real systems. We then cover different strategies that can be used to save energy in video streaming. It turns out that there are a few things that need to be taken into account when applying generic energy-saving techniques to mobile video streaming.

The next two examples are not really specific applications but rather integral parts of many applications and, therefore, they represent extremely important cross-application scenarios. The first of those is sensing. Sensing is a hot research topic at the moment and it is expected to become a very important part of smartphone applications. We study ways to reduce energy consumption with applications that require different kinds of sensing by exploring separately each category of sensors included in modern smartphones. We focus on two well-established techniques: sensor selection and duty cycling.

The second cross-application energy optimization scenario is security. We look at the energy overhead caused by security protocols and algorithms, based on measurement studies. Then, we discuss whether and when it is possible to find a tradeoff between the level of security and energy consumption.

15.1 Video streaming

Multimedia streaming provides an interesting example from several perspectives. First of all, these services, including YouTube, NetFlix, Spotify, and Pandora, have become extremely popular among smartphone users. In particular, video streaming is currently among the largest, if not the largest, mobile application in terms of traffic volumes generated, and further growth is predicted [1]. Therefore, any major improvement in the energy efficiency of streaming applications can have a very large impact.

Video-streaming applications also happen to be very power hungry, stressing the most energy-consuming components of the smartphone. They consume wireless communication power through continuous downloading of the video content, CPU power through decoding of video frames, and display power through video presentation. Luckily, there are many possibilities for optimizing the energy efficiency of these applications. Specifically, the energy spent in communication and computation can be reduced through clever optimization techniques. We first look at how using popular mobile video-streaming services consumes energy. Then, we discuss different opportunities to optimize the energy consumption of the wireless communication, and finally discuss what can be done about the computational energy efficiency of streaming applications.

15.1.1 How mobile video streaming consumes energy

Streaming applications differ from traditional file transfers in an important way. To provide a smooth user experience, the content needs only to be downloaded fast enough so that it arrives just in time before it is scheduled to be viewed, that is, at least at the encoding rate. Any faster downloading does not necessarily lead to any better quality of experience. This observation can also be exploited to save energy.

Wireless communication is one of the main sources of energy consumption for streaming applications. This energy consumption is influenced by the way that the video service provider delivers the content. A number of different strategies are currently in use by popular service providers. A detailed study is available in [2]. We illustrate their differences in Figure 15.1, where the height of the bar corresponds to the rate at which the client receives the content and the width to the time it takes to deliver it.

The first thing to notice is that typically all video-streaming services deliver a sizable chunk of content as quickly as possible to the client at the beginning of the streaming session. The goal is to rapidly fill up the client's playback buffer so that playback can start as soon as possible. Buffering is essential to cope with the jitter and fluctuations in available bandwidth. We refer to this phase as fast start. After the start, the different strategies continue differently. Encoding-rate streaming delivers, as the name suggests, the content at the rate of consumption at the client. Alternatively, the server can deliver content faster than that but still at a throttled rate, typically 1.5 to 2 times the encoding rate. The client player can generate an on-off delivery pattern by reading from the TCP socket only periodically. There are two variations: either the client player uses a persistent TCP connection or it uses multiple consecutive non-persistent connections. In the latter version, the player establishes a new TCP connection each time a new chunk of content is to be downloaded and closes the connection right after the download is complete. Dynamic adaptive streaming over HTTP (DASH) is a kind of special case of on-off where the client explicitly requests each chunk. In fact, the main purpose of DASH is to enable dynamic switching of video quality on the fly while streaming to cope with bandwidth fluctuations [3]. Finally, the client may in some cases download the entire video clip in one go using all the available bandwidth.

Each of these strategies is in use by one or more current video-streaming services, but the selection of the strategy depends on the particular player being used, the selected video quality, and the service provider. The interested reader is advised to look at [2]

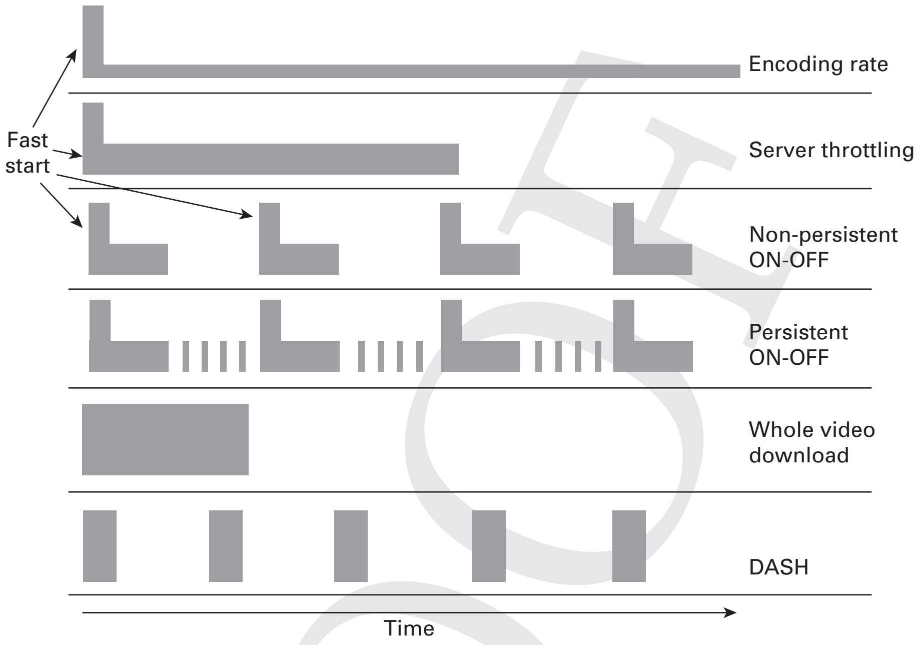


Figure 15.1 The rate of delivery and amount of content served at a time shape the streaming strategy

for further details on how this selection occur with the services and players used at the time of writing. We now turn our attention to the energy consumption of these different strategies.

To give concrete numbers, we show some results from measuring the current draw of smartphones using the video-streaming applications in Figure 15.2. The total current draw was measured using an external power monitor connected to the smartphones. The aggregate current was then divided into current drawn by playback only and current drawn by wireless communication only by looking manually at the current draw in different phases of the streaming session. The figure shows only the wireless communication part.

Downloading content just in time at the encoding rate is the worst possible choice from the energy-consumption perspective. To understand why, we need to again revisit the tail energy phenomenon (Section 7.3.4). When packets carrying the stream content are received at regular intervals but at a rate which is far below the wireless link capacity, tail energy is wasted at the reception of each packet. In the worst case, the packet interval is just a bit shorter than the inactivity timer which is the source of the tail energy. In that case the radio is continuously fully powered on. In other words, the diagrams in Figure 15.1 can also be interpreted so that whenever the bar exists, energy is drained, in the worst case, at a high rate, no matter how tall the bar is. In this way, we can immediately notice that server throttling is also suboptimal from the energy-consumption perspective. Server throttling may also occur in conjunction with the on-off behavior caused by the client player, which is also illustrated in the figure.

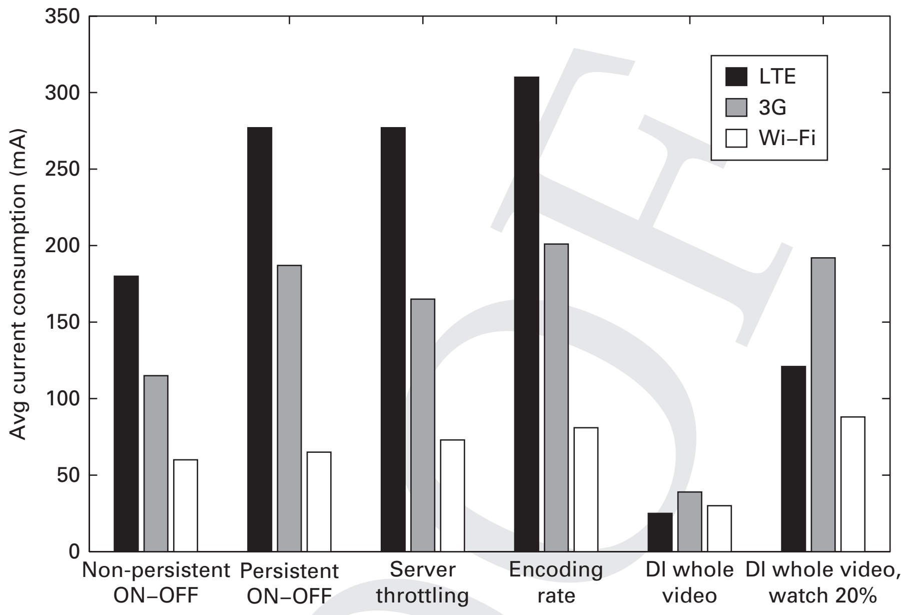


Figure 15.2 Average current draw varies widely between different strategies to deliver video content

The two different cases of on-off deserve a closer look. Surprisingly, the persistent on-off shows poor energy efficiency compared to the non-persistent one when using 3G or LTE access even though their behavior is almost identical. The diagram in Figure 15.1 contains a hint for the underlying reason. When using a persistent TCP connection, in between receiving bursts of traffic, the client stops reading from the TCP socket which then becomes full. Consequently, TCP flow control activates and the client-side TCP forbids the server-side TCP to send more packets by advertising a receiver window of zero bytes. The server, having still more data to send, continues to poll the TCP receiver by sending zero window probes to which the receiver answers with additional zero window acknowledgments. The resulting periodic exchange of small control packets keeps the WNI almost continuously powered on 3G and LTE because of the long inactive timers. Hence, the non-persistent on-off is clearly a more energy-efficient strategy. Note that DRX was not enabled in these LTE tests. Enabling DRX would considerably improve the energy efficiency in this particular case. Finally, downloading the video all at once leads overall to the smallest average current because the WNI is efficiently used and tail energy is only suffered once.

15.1.2 Optimizing the energy consumed by wireless communication

To prefetch or not?

Let us now look at the possibilities for optimizing the energy consumed by the wireless communication. We have already learned that downloading content at the encoding rate is the worst choice from the energy-consumption perspective. So, what can be done

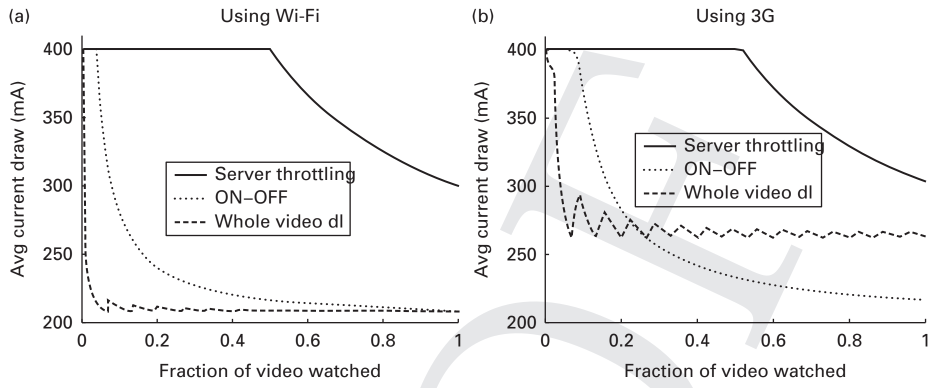


Figure 15.3 Tradeoff between tail energy and downloading unviewed content depends on the streaming strategy and WNI used

to improve the situation? A straightforward answer is that the streaming client should download content using all the available bandwidth to minimize the penalty caused by the tail energy.

There is a caveat, though. Research has shown that for video streaming, users rarely view the entire content. For example, the measurement study presented in [4] reports that more than half of the videos are watched for less than 20% of their duration. The quantitative part of the result may not be generalizable, but there are several equivalent studies that corroborate the qualitative part, that is the fact that it is rare for users to view entire video clips, especially for user-generated content. For this reason, downloading the whole video clip in one shot is in many cases not an energy optimal strategy because some energy will be spent downloading content that will never be consumed. The rightmost set of bars in Figure 15.2 showing the average current drawn when only 20% of the entire content is viewed demonstrate this fact as well.

We further illustrate with Figure 15.3 how the choice of streaming strategy influences the energy consumption when the user is likely to abandon the video-streaming session before the end. We used simplified power models, similar to the ones used in Section 12.2, to compute the results shown by the figure. The plots should be interpreted as follows: Pick an abandoning time on the x-axis (relative time) and the corresponding value on the y-axis tells the resulting average current consumption. We observe that a simple strategy that strikes a middle ground is to deliver the content to the client in larger fixed-size chunks, that is using the non-persistent on-off. This strategy delivers a good tradeoff between tail energy and the amount of unnecessarily downloaded content. Tail energy is spent only once per chunk of content, which, while not optimal, is far better than per packet. If the streaming server delivers a constant bit rate stream, the resulting traffic can also be reshaped at a proxy to generate such a burst pattern (recall our discussion of traffic shaping in Section 12.2). Interestingly, downloading the whole video is almost never a good idea with Wi-Fi connectivity because of the small amount of tail energy that allows prefetching the video energy efficiently in chunks. On the contrary, when using a 3G access, there is a cutoff point after which downloading the whole video is more energy efficient and before which on-off is a better choice.

The optimal strategy would be to download only exactly the content that the user will consume. However, the player or server obviously cannot know beforehand when exactly the user will abandon the session. One way to deal with this uncertainty is to make an educated guess. For example, if viewing statistics for particular video clips are available, it is possible to predict when the user is more or less likely to abandon the session. YouTube, for instance, collects such statistics and presents them in the form of audience retention that the owner of a particular video clip can access. We suggest that the reader who is interested to learn how such statistics could be used to compute an energy optimal download schedule, takes a look at the work presented in [5].

Considering finite buffer space at the client

Another important practical issue to consider when the server or a proxy shapes the traffic into bursts of content is the amount of buffer allocated by the smartphone application. If the buffer size is smaller than the chunk of content being sent by the streaming server or proxy, TCP flow control will again intervene once the buffer is full. The rest of the content is then transmitted at the rate that the buffer is emptied, in other words, at the encoding rate, which we know to be very energy inefficient. Fortunately, the zero window acknowledgments generated by the TCP flow control can be identified by the proxy/server and used as indicators of a too large chunk size. By probing with different chunk sizes and tracking the acknowledgment stream, the proxy/server can determine the optimal size for a particular client, which is precisely what EStreamer does [6].

Finally, as we cannot cover all the details in this book, we refer readers to a fairly thorough survey of the energy efficiency of streaming applications with a focus on the wireless communication which is available at [7].

15.1.3 Computational energy

Energy spent in computing while using streaming applications consists mainly of decoding work and running the player application in general. Measurement results from video playback using a specific smartphone are shown in Figure 15.4. The video quality refers to the resolution so that 240 means 360×240 resolution. Figure 15.4(a) shows that there is a correlation between video quality and playback energy, as expected, although the results also indicate that high-resolution video can be played with quite a small amount of extra energy compared to a low-resolution video. Somewhat surprisingly, the overall playback energy consumption differs significantly between different kinds of player when viewing the same video of the same quality. The video container makes a difference, but also browser-based players (Flash and HTML5) seem to be currently less optimized than players implemented as native applications. Figure 15.4(b) confirms that the CPU usage of these players, especially the HTML5-based one, is at a higher level than the CPU usage of the native application. HTML5 is at the moment rapidly gaining in popularity so we are likely to see more optimized players in the future.

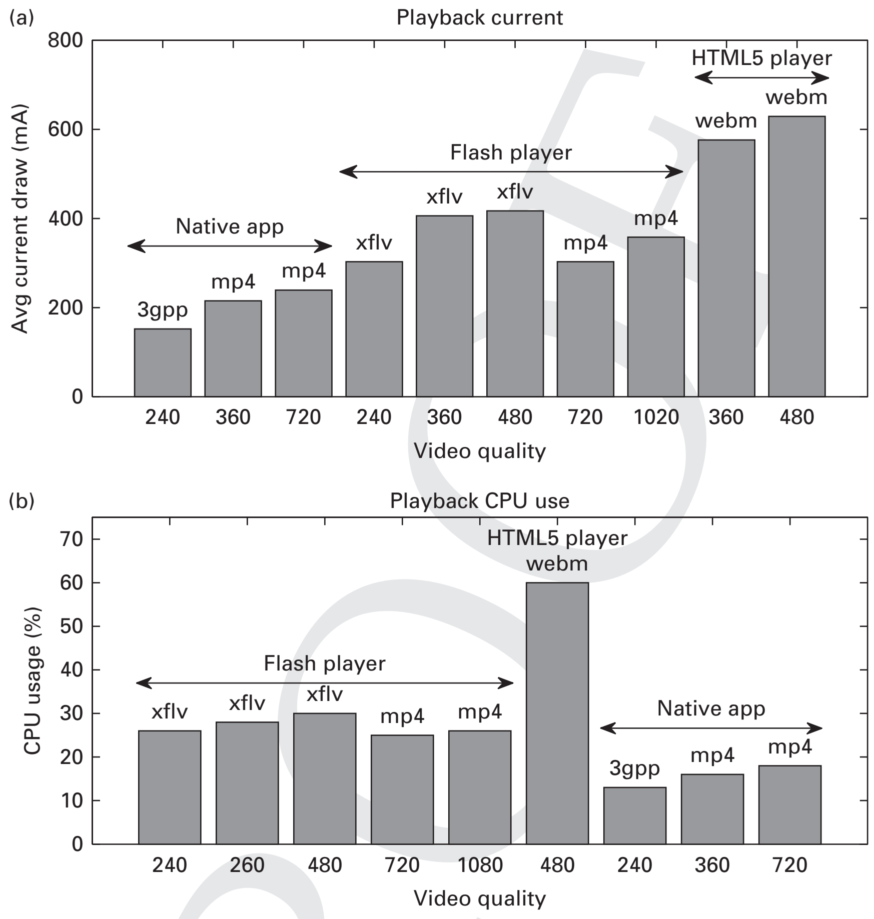


Figure 15.4 Performance measurements of video playback on a smartphone

Video quality intuitively has an impact on the amount of work that the smartphone needs to do to decode and present it. In addition, streaming higher-quality content requires downloading more data. Although in light of the measurements we just looked at, it seems that a significant increase in resolution does not necessarily lead to proportionally as significant an increase in energy consumption. Nevertheless, given the limitations of the current display technologies used in smartphones and the sizes of the displays, it makes sense to adapt the video quality because beyond a certain point it becomes difficult for the human eye to notice the quality improvement given by increased resolution. An example system striving for such adaptation is presented in [8].

15.1.4 Race to sleep vs. dynamic scaling

If we think about the nature of the workload that video streaming generates on the underlying hardware, we notice that it is typically a fairly stable one over a relatively long time. Furthermore, the load is such that it, at least in an ideal case, does not use all the available resources for computation and wireless communications. This type of workload naturally gives rise to two opposing strategies for achieving energy efficiency: race to sleep and dynamic resource scaling.

We have discussed the race to sleep strategy already on several occasions. For wireless communication, shaping the constant bit rate video traffic into high-rate bursts is a direct application of it. It allows the radio to be at least partially powered off in between the bursts, while consuming full power during the reception of them. DPS, which can virtually shut down the whole CPU, represents the equivalent mechanism in CPU power management, as we discussed early on in Section 7.1. However, that is not really applicable for CPU power management during video streaming because the video must play continuously.

However, DVFS, which represents the dynamic resource scaling approach, is an ideal approach for computational power management in video streaming. DVFS, as we have learned, scales the processor frequency and voltage down to such a level that the computational demands of the continuous decoding of video can still just be met. As for wireless communication, equivalent mechanisms do exist. In Section 7.3, we explained the basics of DMS in which the modulation scheme is varied according to the required rate: the lower the symbol rate of the modulation scheme, the lower the transmission rate and the power drawn by the RF processing. Therefore, this approach can be applied in constant bit rate video streaming as well, to match the modulation rate according to the video encoding rate after the initial buffering phase in which the data is received at a faster rate. An example scheme where this principle is applied for video transmission is presented in [9].

15.2 Energy-efficient sensing through sampling

The next application scenario that we study is sensing. We covered the basic principles of the energy consumption of sensing with a smartphone in Section 7.4. In this section, we focus on optimizing the energy consumption through sampling.

The two most important activities in energy-efficient sensor management are (i) sensor selection and (ii) sensor duty cycling. Identifying the minimum set of necessary sensors for a given task is an important step which can easily contribute half of the overall conserved energy. The other half of energy efficiency comprises designing an efficient sensor duty cycling strategy. We next outline some usable techniques for both sensor selection and sensor duty cycling, going through the four sensor categories presented in Section 7.4.1.

15.2.1 Motion sensors

For motion sensing, the accelerometer is typically the primary choice of sensors, augmented with a magnetometer and/or gyroscope if the horizontal phone orientation is required. The horizontal orientation provided by the magnetometer can give a course estimate, but is susceptible to magnetic inferences and errors due to tilt-compensation from the accelerometer during periods of substantial physical movement. More accurate phone orientation is obtainable from the gyroscope, which can be used in combination with the accelerometer and magnetometer to create a fairly accurate, sensor fusion based phone orientation tracking using quaternions [10].

Considering the energy consumption of motion sensing, the accelerometer alone consumes only around 20 mW at the fastest sampling rate. Processing the accelerometer measurements is also computationally light, as the data points typically number only in a few hundred. For a thorough study on accelerometer preprocessing and the complexity of the commonly used features, we refer to work by Figo et al. [11]. A magnetometer consumes roughly twice the energy of an accelerometer, up to around 50 mW. Gyroscope consumption can be up to 140 mW, which already has a significant impact on an application's energy consumption, and should be used carefully in applications that run for extended periods of time. The computation required to transfer the magnetometer and/or gyroscope measurements into Euler-angles or quaternions is relatively light, since the number of data-points is relatively few and the existing algorithms for the task are well optimized.

In addition to the energy consumed by the motion sensors, background applications have to consider the energy overhead from holding the device active for sensor sampling. This energy overhead can be highly significant (e.g., 140 mW for Samsung Nexus S phones), often surpassing the energy consumed by the sensor alone. Furthermore, continuous sampling of the inertial sensors prohibits efficient CPU duty cycling, as there are only short idle periods between receiving the samples from the sensors. Consequently, the most efficient strategy for energy-efficient motion sensing in the background is to interleave bursts of sensor data with idle periods.

As an example of a sampling strategy, see Figure 15.5, which illustrates the energy consumed by an application using an accelerometer, magnetometer, and gyroscope. In the figure, the application periodically samples the accelerometer to decide whether to put the device to sleep or to sample other sensors. Before sampling the more-consuming sensors, the application performs a CPU-intensive double check to verify the requisite of other sensors.

The application's energy efficiency can be further increased by setting a balance between the sensor-sampling frequency and the required accuracy. Modern phones can sample sensors at frequencies well over 100 Hz. However, for typical activity-recognition tasks (e.g., step counting, movement detection, or transportation-mode detection) a lower accelerometer sampling rate, in the range of 32–64 Hz, is sufficient and can conserve energy in both sensor sampling and data processing. To further optimize the accelerometer sampling rate, the rate can be adapted in real-time in accordance with the prevalent activity using existing methods [12]. For a magnetometer, a low (<5 Hz) sampling rate is theoretically sufficient to obtain a crude horizontal orientation. In practice, however, a very low sampling rate makes the sensor more vulnerable to errors due to magnetic inference, and a sampling rate close to 20 Hz should be preferred for better robustness in the presence of magnetic inference. A gyroscope requires a higher frequency, as the measurements are used for tracking a relative orientation rather than

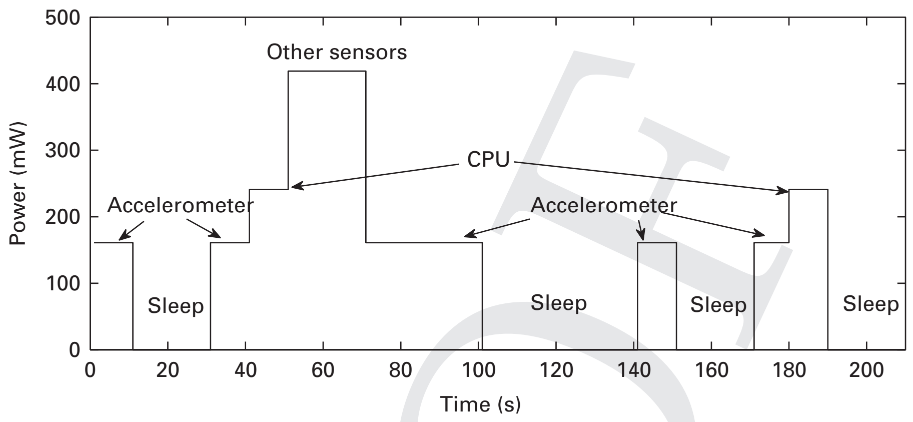


Figure 15.5 Power draw of a sample application

an absolute one. The sensor is typically required for applications that require accurate and real-time phone orientation, for example, for games or gesture tracking, or for situations where high magnetic inference renders the magnetometer unusable, for example, inside a car, due to disturbances from its metallic frame, radio, and other electric components. An example of the latter is detecting driving maneuvers [13], where accurate horizontal-orientation tracking is required to capture characteristics from turning maneuvers.

15.2.2 Wireless sensors

Energy-efficient sensor management for wireless sensors can be divided into sampling strategies for positioning and for data communication. Here we focus on techniques related to positioning.

GPS is an especially interesting case, as it is widely used in sensing applications and capable of draining the phone's battery rapidly. Moreover, planning an efficient duty-cycling strategy for GPS is made challenging due to its inherent properties, such as the inability to tell before-hand whether it can obtain a fix on the location at all, dependence on an unobstructed view to the satellites and low accuracy when only one or two satellites can be tracked, and a relatively long and variable time to achieve a fix on the location.

As these issues with GPS have been well known for over a decade, several established techniques and algorithms for positioning and trajectory tracking have been developed [14], which are relatively easy to implement on mobile devices. The core idea in the majority of the methods is to reduce the need for GPS sampling by using information from other less energy-consuming sensors. For example, the user's heading and speed, and changes in them can be obtained from the motion sensors, which can be used to estimate the current location by extrapolating from the previous GPS location. Significant changes in heading or speed can then be used as hints to get a new position from the GPS sensor [15]. Requesting other GPS-related information, such as indoor–outdoor detection or tracking the user's traveling speed, require slightly modified strategies. Nevertheless, the core idea is typically the same.

Another approach to reduce the energy consumption of positioning is to use one of the other sensors, such as GSM or Wi-Fi, to obtain a coarse location of the user. GSM can provide an almost energy-free solution for coarse-location estimation by using the location of the phone's primary cell tower and its RSSI value. Scanning the Wi-Fi signal environment can be used to detect a familiar configuration of access points and RSSI values, which can be used to pin the user to a location by using previously collected Wi-Fi fingerprints. Besides being more energy efficient, another benefit of these techniques is that they can provide the location estimation instantly, and can also function indoors. However, these techniques require careful calibration involving significant human effort and are susceptible to varying Wi-Fi access point density and GSM cell sizes between different locations, thus, making them rather unreliable outside urban areas and hard to generalize to new environments. Usually data communication is also required to map the collected information to physical coordinates, which consumes some energy.

Considering the power consumption of scanning the wireless environment, the sensors typically consume some energy when being switched on and off, which needs to be accounted for when planning sensor-sampling strategies. Keeping the sensors always active is a viable solution when the sampling rate is fairly high and the application is actively used (i.e., not a background application). When the idle periods become longer, the cost of keeping the sensors always active exceeds the energy cost from switching the sensor on and off. To decide whether to keep the sensor active and when to trigger the sensor off can be formulated as a function of the switching energy and power drawn during the sensing and being idle.

15.2.3 Environmental sensors

When planning energy-efficient sampling for the audio and visual sensors, that is the microphone and camera, a central concern is efficient handling of the large amount of data the sensors output, as the CPU and hardware cost for processing large data can quickly exceed the cost of sampling the sensor. Downsampling and dimensionality reduction are suitable approaches when the full quality is not required. For audio data, a popular method is to use a root mean square (RMS) to extract the energy of audio frames, which can be used to detect the level and variation in volume of audio data. For further methods for audio/visual data processing we refer to [16, 17]. The rest of the environmental sensors consume relatively little energy and, as with motion sensors, the main concern is the background energy of keeping the device active for sensor sampling.

The environmental sensors are in fact cleverly used for energy conservation by the current mobile operating systems. For example, when a proximity sensor reports high proximity combined with an ongoing call, the screen is switched off. The real-world analogy is to turn off the screen when the phone is held near the ear while speaking on the phone. During these periods, the phone's touch screen is usually also turned off to avoid accidentally touching the screen while holding the phone near the head.

The light sensor is used by the phone to adjust the screen brightness dynamically to balance energy efficiency and screen visibility. In the more recent phones, the phone's front-facing camera is used for eye-tracking to deduce when the user is looking at the screen.

In addition to the in-built functions, the various contexts provided by the environmental sensors can be used to adapt the phone behavior, providing further opportunities for energy efficiency. For example, the light and proximity sensors, along with roll and tilt angles provided by the accelerometer can be used to detect smartphone placement. For instance, low-levels of light combined with close proximity indicates that the phone is in a pocket or a bag; while far proximity, normal to high levels of light, and near zero roll and tilt indicate placement on a table or other surface. The phone's barometer can be used for weather forecasting or as an altimeter, but can detect various situations which momentarily effect air pressure, for example, indoor–outdoor switches, and local high-pressure areas such as tunnels, subway entrances, or even urban canyons between skyscrapers. Humidity and temperature sensors can be used in conjunction with the barometer to obtain more accurate weather sensing and to detect situations that display significant changes in these sensors, for example indoor–outdoor or outdoor–underground switches.

15.2.4 Internal sensors

The phone's internal sensors work to adjust and guide the sampling of the other sensors. From the perspective of energy efficiency, the most important central sensors are those measuring the state of the phone's battery, that is, the battery charge, temperature, and voltage. Rapid battery charge depletion or critically low energy can be used to prevent the sampling of high energy-consuming sensors or the performing of non-critical tasks. Another common practice is to schedule energy-heavy tasks, such as data transmission or large I/O operations, to periods when the phone is connected to a power source.

Other internal sensors can be used for monitoring user interaction with the phone. Screen and call states can help to recognize user interaction and phone calls, reducing the need for other sensors during these periods. The accuracy can be enhanced by combining the in tilt and roll angles from the accelerometer to detect typical orientation for these activities, or proximity and light sensors to filter out cases where the screen is accidentally turned on inside a pocket or bag. In addition to providing information about user interaction or calling activities, many activity-recognition applications that rely on motion sensors can significantly benefit from detecting and suppressing periods of extraneous user activities [18].

15.3 Security

Cryptographic algorithms and security protocols are essential for certain kinds of internet usage. The algorithms are based on performing some kind of computational task which means that they necessarily cause some extra energy to be consumed. In addition, the protocols used to secure transactions and data communications therein are in addition to the cryptographic operations based on some kind of exchange of messages between the communicating parties. Therefore, some energy is also consumed on additional communication. In this section, we study the energy consumption of these algorithms and protocols in detail.

15.3.1 Secured communications using cryptographic algorithms

Security protocols are a must for many mobile services. There are different kinds of protocol that can be used to guarantee confidentiality, authentication, message integrity, and nonrepudiation of message origin. Confidentiality ensures that no third party can eavesdrop on communication between two parties. User authentication is required by many services, but those protocols also allow users to make sure that they are using an authentic banking service, for instance, and not a fake one set up for phishing. Message integrity ensures that message tampering does not go unnoticed and nonrepudiation is sometimes necessary to verify that a message has been sent by a specific party.

There are three basic families of cryptographic algorithm: asymmetric, symmetric, and hash algorithms. When using symmetric algorithms, the same algorithm operations are used for the encryption of plain text and the decryption of cipher text. Consequently, both parties use the same shared secret key for encryption and decryption. Examples of such algorithms are AES (Rijndael) and 3DES. In contrast, asymmetric cryptography is based on a setup where each party possesses a pair of public and private keys, and the public key can be used to decrypt messages encrypted with the private key and vice versa. As the name suggests, the public key is shared to everyone and the private key is a secret known only by the party associated with the keypair. Asymmetric algorithms are based on intractable mathematical problems, such as integer factorization and discrete logarithms. The difficulty of the problems ensures that given a public key and valid plain and cipher texts, it is computationally infeasible to find the matching private key. Examples of asymmetric algorithms are RSA, DSA, and those based on elliptic curves, such as ECDSA. Hash algorithms take a variable size message as input and compute a fixed, typically shorter length output that is also sometimes called a message digest.

Symmetric algorithms are typically used to provide confidentiality, that is to encrypt data transfers between mobile devices and servers, while asymmetric algorithms are very useful for authentication and nonrepudiation purposes through digital signatures. Signing a message essentially means encrypting it, or some digest of it, using the private key. The receiver can then verify that it was indeed produced using the private key of the authentic user by decrypting the signature using the corresponding public key. Combined with a cryptographic secret, for example a symmetric key or digital signature, hash algorithms are often used to protect the integrity of messages.

These algorithms are usually combined together in a security protocol to provide a completely secure transaction with the desired level of protection. Transport Layer Security (TLS), previously known as the Secure Sockets Layer (SSL), is a widely used example protocol. With TLS, the client and server negotiate a so-called cipher suite, which is simply the set of cryptographic algorithms to be used at the beginning of a transaction. For a comprehensive coverage of the cryptographic algorithms and security protocols, we suggest the reader looks at, for instance, the book by Stallings [19].

15.3.2 Energy overhead of security

It is obvious that the use of security protocols causes an energy consumption overhead compared to unsecured communication and computation. The overhead comprises both cryptographic and non-cryptographic components. We next discuss both in turn.

Energy spent on cryptographic operations

All cryptographic algorithms necessitate a certain amount of computation which consumes energy, but the amount of computation depends on the particular algorithm being used. Overall, asymmetric cryptography is the most computationally intensive followed by symmetric cryptography. Computing message digests with hash functions usually causes only negligible amount of computational work.

A measurement study of the energy consumption of cryptographic algorithms is presented in [20]. Although it is already a bit dated and embedded hardware has since evolved quite a bit, it remains a good demonstration of the core differences between the algorithms. We summarize some of the results from that study on the cryptographic energy consumption overhead in Table 15.1. The key point is that the algorithms often have different tradeoffs. For example, RSA verification is extremely efficient but generating a signature consumes a large amount of energy, while DSA energy consumption is more balanced between the different operations. Therefore, depending on the particular usage, one algorithm may be better than another one. Elliptic curve cryptography is in general computationally efficient which also shows in the energy consumption. One of the reasons is that a shorter key can be used with ECDSA to achieve a level of security comparable to DSA. There are also clear differences between the different symmetric algorithms in terms of the energy consumed per byte of data encrypted or decrypted. The fact that DES consumes less energy than 3DES should be complemented by the fact that it is clearly less secure than its triple version (3DES). On the other hand, AES combines a high level of security with energy efficiency. The hash algorithms used in the study are already becoming somewhat obsolete as newer versions of the protocols have been developed (e.g., SHA-3 and MD6).

Non-cryptographic energy overhead

The non-cryptographic part refers to additional communication and computation due to the actual security protocol. Running any security protocol involves a certain number of messages being exchanged between the two communicating parties prior to the actual data communication during which information required by the cryptographic algorithms, such as signatures and certificates for authentication, is exchanged and parameters are negotiated. Some extra data is also piggybacked with data packets during the data communication, such as message digests for integrity protection. The non-cryptographic overhead caused by the protocol message exchange prior to the data

Table 15.1. Energy consumption of example cryptographic algorithms

Operation		Energy consumption	
Asymmetric	*RSA*	*DSA*	*ECDSA*
Key generation	medium	medium	medium
Sign	high	medium	low
Verify	ultra low	medium	low
Symmetric	*DES*	*3DES*	*AES*
Encrypt/decrypt unit of data	medium	high	low
Hash	*MD5*	*SHA1*	
Compute digest	medium	medium	

communication can make up a major part of the total energy consumption for small transactions but this overhead decreases rapidly when the transaction size increases.

The measurement results presented in [21] suggest that the extra energy spent in symmetric cryptography, that is encryption and integrity protection, when using TLS is mostly insignificant compared to the energy spent in the actual data transfer. However, the message exchange and authentication part of the protocol, namely the handshake, that is done before the data transfer can cause a significant overhead. It can even contribute up to 90% of the total energy consumption of very small 1 KB transactions, but this overhead reduces to a negligible amount with transactions of several megabytes. An interesting observation is that this overhead differs significantly between different popular websites, such as Google and Facebook. The reason turned out to be the server's certificate length and the round-trip time of the network path from the client to the server, both of which directly influence the duration that the WNI needs to be powered on during the handshake.

15.3.3 Can we trade energy for security?

Many different kinds of algorithm exist for symmetric and asymmetric cryptographic operations and hashing. Most can be used for the same purposes but the different algorithms typically strike different tradeoffs between specific properties, such as the amount of computation required and the level of protection offered against brute force attacks, for example. Furthermore, a given algorithm can be used with different parameters. One of the most important parameters is the key length, which directly influences the robustness of the scheme against brute force attacks but also the computational complexity and, hence, the energy consumption. Table 15.2 gives some figures on the key length vs. energy consumption tradeoff with different kinds of algorithm collected from two studies [20, 22].

These results demonstrate that the energy consumption overhead can be to a certain extent controlled by trading off some security. The impact of key length on energy consumption in algorithms for asymmetric key cryptography (RSA in the table) is notably large, whereas the effect is much smaller with symmetric key cryptographic algorithms (AES in the table). However, as we noted earlier, the computational energy overhead is

Table 15.2. Energy consumption of different cryptographic protocols and key lengths

Algorithm	Low security keysize (bit)	High security keysize (bit)	Energy cons, difference
AES	128	256	+ 20–40%
RSA	1024	2048	+ 450–750%
EC	163	283	+ 220%

often overshadowed by the energy spent in the actual data transmission during a transaction, in which case trading off some security by reducing the cryptographic algorithm's key length, for example, may provide only small relative energy savings in the end.

15.4 Summary

In this chapter, we looked at a few cases of mobile application usage where the energy efficiency plays an important role and optimization is important. We first looked at the energy efficiency in mobile video streaming which is one of the most popular but also most energy-consuming applications used today with smartphones. Fortunately, there is much room to optimize the energy consumption in mobile video streaming. For example, traffic shaping is highly effective with audio and video streaming. It is also possible to optimize the pre-fetching behavior of the video player so that downloading of unnecessary content, that is video that the user will never watch, is minimized.

The second case we studied was the use of the built-in sensors by smartphone applications. In such applications, it is useful to consider two kinds of optimization: sensor selection and duty cycling. Selection means that a lower power sensor could be used most of the time to check whether a higher power sensor should be activated. So, the idea is similar to the concept of wake-up radios (Chapter 13). Duty cycling means that the rate at which sensors output values is adjusted so that the accuracy and timeliness required by the application is suitably balanced with the energy consumed by the sensor sampling.

The third case focused on security. Security protocols and the cryptographic algorithms used therein consume energy in the form of computation and wireless communication. Hence, secure mobile communications comes at a price of a certain energy overhead. The different kinds of algorithm consume very different amounts of energy and sometimes even the same level of security can be achieved with less energy consumed through careful selection of the cipher suite being used.

References

[1] Cisco, "Cisco visual networking index: Forecast and methodology, 2012-2017," Cisco, White Paper, May 2013.

[2] M. A. Hoque, M. Siekkinen, J. K. Nurminen, and M. Aalto, "Dissecting mobile video services: An energy consumption perspective," in *Proc. 14th IEEE Int. Sym. on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, Jun. 2013.

[3] T. Stockhammer, "Dynamic adaptive streaming over http –: standards and design principles," in *Proc. 2nd Annu. ACM Conf. on Multimedia Systems*. New York, NY, USA: ACM, 2011, pp. 133–144.

[4] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao, "Youtube everywhere: impact of device and infrastructure synergies on user experience," in *Proc. 2011 ACM SIGCOMM Conf. on Internet Measurement*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 345–360.

[5] M. A. Hoque, M. Siekkinen, and J. K. Nurminen, "Using crowd-sourced viewing statistics to save energy in wireless video streaming," in *Proc. 19th Annu. Int. Conf. on Mobile Computing and Networking*. New York, NY, USA: ACM, 2013, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/2500423.2500427

[6] M. Hoque, M. Siekkinen, and J. K. Nurminen, "TCP receive buffer aware wireless multimedia streaming – an energy efficient approach," in *Proc. 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, 2013, pp. 13–18.

[7] M. Hoque, M. Siekkinen, and J. Nurminen, "Energy efficient multimedia streaming to mobile devices: A survey," *IEEE Communications Surveys Tutorials,* vol. PP, no. 99, pp. 1–19, 2012.

[8] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian, "Integrated power management for video streaming to mobile handheld devices," in *Proc. 11th ACM Int. Conf. on Multimedia*. New York, NY, USA: ACM, 2003, pp. 582–591. [Online]. Available: http://doi.acm.org/10.1145/957013.957134

[9] Y. Li, M. Reisslein, and C. Chakrabarti, "Energy-efficient video transmission over a wireless link," *IEEE Transactions Veh. Technol.*, vol. 58, no. 3, pp. 1229–1244, 2009.

[10] S. Madgwick, A. J. L. Harrison, and R. Vaidyanathan, "Estimation of imu and marg orientation using a gradient descent algorithm," in *2011 IEEE Int. Conf. on Rehabilitation Robotics (ICORR),* 2011, pp. 1–7.

[11] D. Figo, P. C. Diniz, D. R. Ferreira, and J. a. M. Cardoso, "Preprocessing techniques for context recognition from accelerometer data," *Personal Ubiquitous Comput.*, vol. 14, no. 7, pp. 645–662, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1007/s00779-010-0293-9

[12] Z. Yan, V. Subbaraju, D. Chakraborty, A. Misra, and K. Aberer, "Energy-efficient continuous activity recognition on mobile phones: An activity-adaptive approach," in *Proc. 2012 16th Annu. Int. Symp. on Wearable Computers (ISWC)*, ser. ISWC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 17–24. [Online]. Available: http://dx.doi.org/10.1109/ISWC.2012.23

[13] H. Eren, S. Makinist, E. Akin, and A. Yilmaz, "Estimating driving behavior by a smartphone," in *2012 IEEE Intelligent Vehicles Symp. (IV),* 2012, pp. 234–239.

[14] J. Paek, J. Kim, and R. Govindan, "Energy-efficient rate-adaptive gps-based positioning for smartphones," in *Proc. 8th Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 299–314. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814463

[15] M. B. Kjærgaard, S. Bhattacharya, H. Blunck, and P. Nurmi, "Energy-efficient trajectory tracking for mobile devices," in *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 307–320. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000025

[16] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell, “Soundsense: Scalable sound sensing for people-centric applications on mobile phones,” in *Proc. 7th Int. Conf. on Mobile Systems, Applications, and Services*, ser. MobiSys ’09. New York, NY, USA: ACM, 2009, pp. 165–178. [Online]. Available: http://doi.acm.org/10.1145/1555816.1555834

[17] G. Raffa, J. Lee, L. Nachman, and J. Song, “Don’t slow me down: Bringing energy efficiency to continuous gesture recognition,” in *2010 International Symp. on Wearable Computers (ISWC),* 2010, pp. 1–8.

[18] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell, “The jigsaw continuous sensing engine for mobile phone applications,” in *Proc. 8th ACM Conf. on Embedded Networked Sensor Systems*, ser. SenSys ’10. New York, NY, USA: ACM, 2010, pp. 71–84. [Online]. Available: http://doi.acm.org/10.1145/1869983.1869992

[19] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2010.

[20] N. Potlapally, S. Ravi, A. Raghunathan, and N. Jha, “A study of the energy consumption characteristics of cryptographic algorithms and security protocols,” *IEEE Trans. on Mobile Computing,* vol. 5, no. 2, pp. 128–143, 2006.

[21] P. Miranda, M. Siekkinen, and H. Waris, “TlS and energy consumption on a mobile device: A measurement study,” in *2011 IEEE Symp. on Computers and Communications (ISCC)*, 2011, pp. 983–989.

[22] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz, “Energy analysis of public-key cryptography for wireless sensor networks,” in *Proc. 3rd IEEE Int. Conf. on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 324–328.

16 Future trends

Smartphones have evolved in leaps and bounds during the last decade and this evolution is continuing with new software and hardware capabilities as well as new application domains, such as augmented reality and the Internet of Things (IoT). In this chapter we briefly examine future smartphone trends from the perspective of energy consumption. We conclude that radical disruptions are not likely in the near future, keeping energy a first-class resource for mobile devices and warranting new solutions to maintain and improve device battery life

16.1 Future smartphone

Smartphones have become the essential instrument for our daily lives, supporting a plethora of usage scenarios from the workplace to the home and leisure. Today's smartphone is a small-factor, high-performance computer that can offer graphics resolution and power on a par with the previous generation of game consoles. The communications capabilities of smartphones have evolved tremendously over the past decade and now high-speed, low-latency 4G and beyond networks, Wi-Fi, and local area networks are ubiquitously supported. The sensing capabilities have improved dramatically with GPS, acceleration, light, and temperature to give some examples of on-board sensors and the sensing capabilities include complex gestures and voice.

In addition to the hardware capabilities, the mobile-application ecosystem has had a stellar success in recent years since iOS, Android, and Windows Phone were launched. Application developers have used the mobile platform APIs and the hardware capabilities of smartphones to create new innovative applications that in many cases combine sensors and communications capabilities in unexpected ways.

However, despite the improvements in hardware and the success of the mobile-application sector, smartphone battery lives are seen to be limited, typically less than one day of operating time with moderate to heavy use, and mobile software is still ridden with energy bugs and inefficiencies. These challenges are not likely to go away in the near future, because radical improvements are not expected in the battery technology, and the applications and the way they use the hardware are becoming more complex and more demanding, for example always-on sensing and high-quality audio and video playback and recording. Smartphones will need new solutions to maintain the three-watt limit with all these activities happening simultaneously.

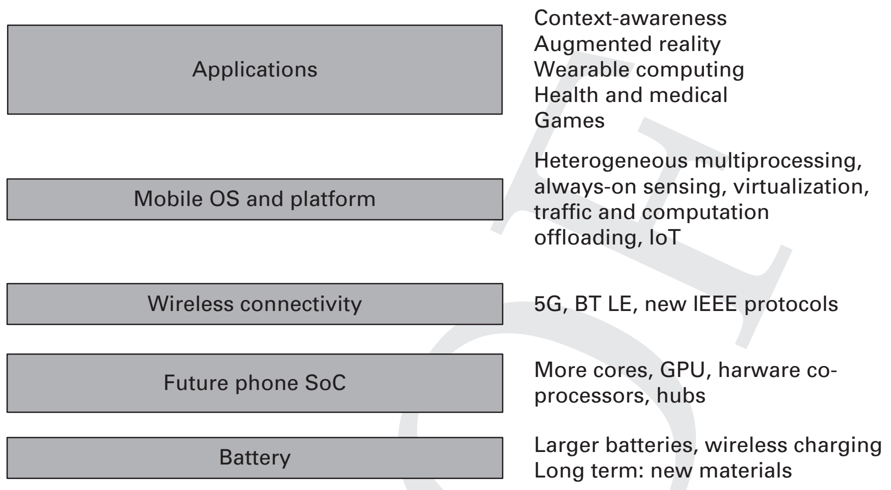

Applications
Context-awareness
Augmented reality
Wearable computing
Health and medical
Games
Mobile OS and platform
Heterogeneous multiprocessing, always-on sensing, virtualization, traffic and computation offloading, IoT
Wireless connectivity
5G, BT LE, new IEEE protocols
Future phone SoC
More cores, GPU, harware co-processors, hubs
Battery
Larger batteries, wireless charging
Long term: new materials

Figure 16.1 Near-future smartphone trends

To illustrate the battery challenge in the face of evolving hardware, we can examine the specifications of iPad 2 and iPad 4. The latter has significantly more advanced components, for example 4G, retina display, and improved processor; however, the expected talk time for the two devices is the same, 9 hours. The latter is heavier being 662 grams compared to 607 grams and has a significantly larger battery of 11 560 mAh compared to the 6930 mAh of iPad 2. Even though the newer device has a significantly larger battery the talk time is the same.

In this chapter, we briefly examine anticipated future smartphone features from the energy consumption perspective. Figure 16.1 gives an overview of the key topics:

- Battery technology for which there are potential materials science breakthroughs looming on the horizon; however, it will take many years for the new technologies to mature.
- Future smartphone SoC for which the number of cores and co-processors is rapidly increasing.
- Wireless connectivity that is becoming more heterogeneous with 5G, Bluetooth LE, and the 802.11 family of protocols.
- Mobile OS and platform that is being extended to support heterogeneous multiprocessing and advanced offloading techniques.
- Emerging applications domains, such as context-aware applications, augmented reality, and wearable computing.

16.2 Battery

Modern battery technology is fairly mature and there does not appear to be a viable competitor to Li-Ion and Li-Po batteries in the next five years. Battery technology improvement is limited by the amount of electrical energy that can be obtained from the materials [1]. Carbon nanotubes and other nano materials are currently being researched as anode materials, but these are still in the conceptual and experimental stage [2]. A nanowire battery replaces the traditional graphite anode of Li-Ion batteries with a stainless steel anode that is covered with silicon nanowires [3]. This new form of Li-Ion technology can potentially hold ten times more charge than the current technology allows. This technology is currently being commercialized.

Alternative anode materials, for example clusters of silicon between the graphene sheets, are showing promise in extending charge life up to ten times with ten times faster charging [4]. A nanotube electrode as the positive electrode and lithium titanium oxide as a negative electrode have been reported to achieve approximately five times the amount of charge and ten times the power of conventional Li-Ion batteries [5]. High-power Li-Ion microbatteries could potentially enable even smaller IoT devices with a higher wireless transmission range [6]. As an alternative to Li-Ion, aluminum-ion batteries have been proposed that would have significantly higher energy densities. A wooden battery has been proposed for cheap and similar energy densities to current Li-Ion technologies [7].

Metal-air batteries have been researched since the 1970s, but there are still challenges relating to recharging and recycling. IBM started the Battery 500[1] initiative in 2009 to produce a battery that can power a car for 500 miles. The Li-Air battery uses the oxidation of lithium at the anode and the reduction of oxygen at the cathode for moving current [8]. Discharging IBM's air breathing Li-Air battery involves oxygen reacting with lithium that results in lithium peroxide and electrical energy. When charging, the reverse process releases oxygen. The main benefit of Li-Air is the high energy density of the battery that is comparable to gasoline. A Li-Air battery can have as much as 15 times greater energy density than a Li-Ion battery. A critical review of Li-Air batteries predicts that the technology will remain a research topic for the next several years due to remaining fundamental challenges [9].

Energy harvesting [10] is an interesting direction for very small devices, such as wearables and sensors; however, it is not feasible for smartphones due to their power requirements. A technique based on the reverse electrowetting phenomenon has been proposed that can harness energy from walking. The technique could result in a small generator that powers mobile devices, such as smartphones [11].

Heat output is a major challenge for the design of smartphones that limits the performance and capabilities of the device. Today's smartphones have been designed so that they do not require active cooling such as fans; however, in certain heavy load circumstances they can become noticeably warm. To maintain the operating temperature range, the SoC will throttle the performance of the device to reduce the heat output. This throttling can have negative effects on the user experience. The basic thermal design techniques relate to the placement of the components and the materials used. In addition, the SoC can tune the multicore system by choosing the cores, frequencies, and

[1] `http://www.ibm.com/smarterplanet/` accessed January 6, 2014.

voltages to minimize power draw and heat output. SoC components, such as the GPU, can be switched to low-power modes.

Passive liquid cooling has recently been introduced for smartphones by the NEC Medias X 06E phone that has a liquid-filled heatpipe that transfers heat from the processor to a graphite radiator across the device. Liquid cooling is one potential solution for alleviating smartphone heat output.

16.3 Future smartphone SoC

The near-future evolution of smartphones is likely to see a continued increase in the number of cores and better tuning of the multicore system, high-performance GPUs, more co-processors and cores for dedicated application activities such as sensing, and more coordination of activities in hardware. The evolution appears to be following Moore's Law that states that the performance of electronics doubles every 18 months.

As discussed in Chapter 2, the static power leakage affect the feasibility of voltage scaling. To address leakage, SoCs use power gating to allow portions of the system to be switched off when necessary. Low power requirements are also driving the trend for multicore systems with many voltage and frequency controlled cores.

Alternative CPU technologies are being investigated; however, these will still take many years to reach the market. For example, graphene CPUs are expected to support much higher frequencies than today's technology. A prototype graphene CPU has been developed, but it is too large for practical deployments [12].

The smartphone DSPs and GPUs are becoming powerful general purpose computation engines. OpenCL, CUDA, and advanced shader languages allow general algorithms to be developed and offloaded to mobile GPUs [13].

The SoC designs will have to address always-on sensors and communications, while at the same time minimizing the energy consumption of the system. This requires that unnecessary cores and components can be shutdown in a coordinated manner. The recent trend of coordinating activities, such as threads and tasks, data transfers, and sensing operations, is expected to continue with more intelligence in the hardware, drivers, and OS.

16.4 5th generation mobile networks (5G)

5G (5th generation mobile networks)[2] is the term generally used to describe the mobile world next to come into existence [14]. It is used for all characteristics which are beyond the current official standards 4G/IMT-Advanced. The use of the term is therefore not formal and 5G does not cover any particular set of specifications and the standardization authorities for telecommunication have not published or endorsed anything beyond 4G.

[2] `https://www.metis2020.com` accessed January 6, 2014.

Thus, the updates of 4G are not grouped under the 5G. Still, it is quite natural to use the name 5G for the still open future.

What kind of network will 5G be? There is no clear consensus beyond the basic fact that a new standard will be needed in the near future, possibly in the early 2020s. One difficulty is the nature of frequency bands: they are a scarce resource and ensuring that the new generation of telecommunication will have a markedly greater peak bit rate is not easy or straightforward. So the throughput will probably not be the most critical technological development/marketing point of the emerging 5G. Other measurable quantities and characteristics could be more important.

There are many limitations relating to the wireless communication of current mobile devices. The new standards should address problems such as the increase in battery usage due to the ever-growing set of services offered by mobile devices. Energy efficiency is expected to be a key requirement for 5G network equipment and mobile terminals. In addition, the latency should be lower than in current devices and the cellular network should support scalability, better coverage, and higher versatility. The costs of network deployment should go down and requirements for data volume increase per area unit are growing steadily. All these combined with higher bit rates present formidable problems for the designers of the future cellular network.

Europe has traditionally been the leader in mobile communications technology and the European Union has recognized the need for new standards, that is 5G. The discussions of development beyond 4G have introduced many new concepts deemed important and desirable. Notably among these are those traits which will support emerging IT trends, such as the IoT. IoT means a new product/consumer world where most or all products are automatically given IP addresses and connected with the Internet. This is a profound revolution which sets new and demanding tasks and requirements to all parts of digital technology, not least to the telecommunication sector. The number of connected devices will grow astronomically, requiring better overall control of the system, greater speeds, reduced latency, and, foremost, a new level of reliability. Also an important factor in this would be the pervasiveness of the network, providing ubiquitous computing. The need to be connected simultaneously and smoothly to several access points with differing technologies is very important and in fact necessary for cloud technologies. All this naturally requires the existence of IPv6 for assigning unimpeded mobile IPs.

The network in its entirety must be changed and the starting points and targeting of its design should reflect new aims and requirements. 5G should be user-centric instead of operator-centric or bound to developers' interests like the previous 1G–4G were. 5G must be a comprehensive W4 (World Wide Wireless Web) tool supporting new and much-needed characteristics of the mobile web, instead of catering for the W4 needs only in passing, like now. Because the users are often moving, the system should support the changing of location in and between the cells; higher bit rates are needed throughout the cell boundaries. This can be achieved by using cellular repeaters and also grouping cooperative relays (macro-diversity), where users themselves are an active part of the network through direct D2D (device-to-device) telecommunication capacity. These would be major steps forwards in getting true multi-hop networks. The flexibility of resources will be greatly enhanced when each cell can act in both uplink and downlink communication. Interference problems are bound to increase and they must be taken care of.

Currently it is trendy to speak of massive data analysis, storage, and handling. These also set new challenges which should be managed. Techniques like Massive Distributed MIMO are helpful because they facilitate sending intensive message streams from transmission points with large numbers of antennas. This maximizes the gain from a single resource and minimizes interference. The emerging network greatly exceeds the density of the current one. In general the management of interference and mobility needs advanced solutions where the transmission points at least cooperate with each other and accept overlapping of the areas. Multiple concurrent paths for data are also a prerequisite for massive telecommunication and computation.

The scarcity of available frequency spectrum requires new dynamic radio technologies. Smart radio is the name under which we collect technologies allowing efficient sharing of the same spectral area by different wireless technologies. The sharing is dynamic because unused spectrum is actively searched for and adaptive because the transmission devices automatically adapt to the changing technological environment sharing a spectral area. Management of this cognitive radio is necessarily distributed and wireless parameters must be defined by software.

Requirements for massive data transfer necessitate, for instance, multiplicity in access architecture, modified physical layers, AND new filter bank handling (non-orthogonal). Furthermore, the solutions should be flexible and allow for reducing the end-to-end latency. Ad hoc networking and smart antenna systems with flexible modulation could critically enhance the efficiency of 5G. Nevertheless, the main problem with 5G is probably not its technological complexity but the difficulty in achieving a truly global and universally accepted standard. The targets of various interest groups differ, often considerably, and thus the task of standardizing 5G will not be easy.

16.5 Mobile OS and platform

Smartphones are already in many cases always-on and connected to the Internet. This trend has resulted in the rise of mobile cloud computing, in which applications and services are distributed between mobile devices and the fixed-network cloud computing infrastructure that is provided by data centers. Mobile cloud allows the development of computationally heavy applications that are distributed: the sensing and initial computation happens on the mobile devices after which the data is sent to the cloud for the heavy processing. The cloud then provides results back to the mobile devices.

For example, voice and face recognition are classical examples of computationally heavy processes that can be offloaded to the cloud. We investigated well-known techniques in Chapter 14. In addition, mobile 3D graphics processing can be divided between the device and its GPU and a cloud-based, high PERFORMANCE GPU cluster [13].

Virtualization is an emerging technique that is supported by modern mobile processors, such as the Cortex-A15. Virtualization allows the execution of multiple OS instances simultaneously over the same hardware. The isolation provided by virtualization can keep the multiple work environments with their data separate. The hypervisor can support multiple CPU clusters and migrate guest OSs across different clusters.

Nowadays, it is very common for mobile devices to handle multiple tasks concurrently. This requires the power management at the system level to be able to handle more complex situations. For example, it is possible for multiple application-level solutions to be applied to the same system, with one solution for one running application. In that situation, power-management software must extend its functionality from hardware-resource management to the management of these solutions, for example scheduling the context sharing among solutions and avoiding the conflicts in resource usage between them. Despite the increased complexity in power management, we are also seeing opportunities for coordinating these solutions to further improve the energy savings. This coordination can happen not only in the smartphone but also between smartphones and the surrounding devices and the cloud.

Energy savings that can be achieved from these solutions depend on the trade-offs between power consumption and performance, and between computational cost and transmission cost. Predicting the near-future values of context variables can help in deciding whether executing the adaptations defined in the solutions would save energy or not. Accurate prediction requires knowledge of the system and its power consumption. A challenge comes from the complexity of wireless data transmission. Compared to computation, the execution of wireless data transmission includes many more uncertainties because it depends on the network protocols used for implementing the transmission, the network devices carrying the data through the network, and the network environment where the transmission happens. If these influential factors can be described using context, it follows that the power-management software must be aware of the context and be able to adapt to their changes.

16.6 Application domains

With the advent of wearable computing and augmented reality applications, always-on sensing will become commonplace as well as local communication not only with other smartphones but also with all kinds of active and passive auxiliary devices around the smartphone. We examined energy-efficient sensing in Section 15.2. Indeed, the smartphone is rapidly becoming an important interface for the IoT. Wearable computing and the IoT present unique challenges for energy-efficient operation, because they involve wireless communications and typically the sensors have very small batteries. The smartphone as a hub, on the other hand, may need to interact with a large number of sensors and connect them to the Internet thus posing challenges in terms of the scalability of communications and networking and battery life. The Bluetooth Low Energy examined in Section 7.3.2 is now becoming a key protocol for wearables and the IoT.

As the demand for hardware resources comes from mobile applications, many energy-efficient mobile applications have been proposed in accordance with the criteria of trying to reduce as much processing and data transmission workload as possible. For example, the power consumption of video playback depends on the quality of the videos. Transcoding proxies were introduced into video-streaming systems for compressing the videos into ones with lower quality before forwarding them to the mobile devices [15]. Another recent example of ways to reduce the workload of the mobile devices is offloading computation from mobile devices to the cloud [16, 17].

Mobile cloud computing promises to integrate mobile devices and scalable cloud platforms into a unified distributed computing environment that can address the challenges of wearables, the IoT, and ubiquitous computing. We examined mobile cloud related solutions, namely offloading, in Chapter 14.

16.7 Summary

Heterogeneity of the operating environment is an overall trend overarching the different layers and systems. We are seeing heterogeneity in the SoC, platform, applications, and the distributed environment. This heterogeneity creates many possibilities for innovation; however, it also increases engineering complexity and makes it more difficult to model, optimize, and debug these systems. Energy efficiency will be a key challenge for smartphones and many of the emerging application domains, such as wearable computing, augmented reality, and the IoT. This book has presented an overview of the challenges of energy modeling and optimization for smartphones, but many of the techniques are generic and applicable also to wearable devices and sensors. The smartphone is expected to become an important interface for interacting with the emerging heterogeneous computing environment. Thus the energy optimization of smartphones and their subsystems is a crucial challenge today and in the near future.

References

[1] D. Linden and T. B. Reddy, *Handbook of Batteries*, 3rd ed. McGraw-Hill Professional, 2001.

[2] P. Bruce, B. Scrosati, and J.-M. Tarascon, “Nanomaterials for rechargeable lithium batteries,” *Angewandte Chemie International Edition*, vol. 47, no. 16, pp. 2930–2946, 2008. [Online]. Available: http://dx.doi.org/10.1002/anie.200702505

[3] C. K. Chan, H. Peng, G. Liu, K. McIlwrath, X. F. Zhang, R. A. Huggins, and Y. Cui, “High-performance lithium battery anodes using silicon nanowires,” *Nature Nanotechnology*, vol. 3, no. 1, pp. 31–35, 2007. [Online]. Available: http://www.nature.com/nnano/journal/v3/n1/full/nnano.2007.411.html

[4] X. Zhao, C. M. Hayner, M. C. Kung, and H. H. Kung, “In-plane vacancy-enabled high-power sigraphene composite electrode for lithium-ion batteries,” *Advanced Energy Materials*, vol. 1, no. 6, pp. 1079–1084, 2011. [Online]. Available: http://dx.doi.org/10.1002/aenm.201100426

[5] S. W. Lee, N. Yabuuchi, B. M. Gallant, S. Chen, B. S. Kim, P. T. Hammond, and Y. Shao-Horn, "High-power lithium batteries from functionalized carbon-nanotube electrodes," *Nature Nanotechnology*, vol. 5, no. 7, pp. 531–537, Jul.

[6] J. H. Pikul, H. Gang Zhang, J. Cho, P. V. Braun, and W. P. King, "High-power lithium ion microbatteries from interdigitated three-dimensional bicontinuous nanoporous electrodes," *Nat. Commun.*, vol. 4, p. 1732, Apr. 2013, supplementary information available for this article at http://www.nature.com/ncomms/journal/v4/n4/suppinfo/ncomms2747_S1.html.

[7] H. Zhu, Z. Jia, Y. Chen, N. Weadock, J. Wan, O. Vaaland, X. Han, T. Li, and L. Hu, "Tin anode for sodium-ion batteries using natural wood fiber as a mechanical buffer and electrolyte reservoir," *Nano Letters*, vol. 13, no. 7, pp. 3093–3100, 2013. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/nl400998t

[8] G. Girishkumar, B. McCloskey, A. C. Luntz, S. Swanson, and W. Wilcke, "Lithium-air battery: Promise and challenges," *The Journal of Physical Chemistry Letters*, vol. 1, no. 14, pp. 2193–2203, 2010. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/jz1005384

[9] J. Christensen, P. Albertus, R. S. Sanchez-Carrera, T. Lohmann, B. Kozinsky, R. Liedtke, J. Ahmed, and A. Kojic, "A Critical Review of Li/Air Batteries," *Journal of the Electrochemical Society*, vol. 159, no. 2, pp. R1–R30, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1149/2.086202jes

[10] S. Chalasani and J. Conrad, "A survey of energy harvesting sources for embedded systems," in *2008. IEEE Southeastcon*, 2008, pp. 442–447.

[11] T. Krupenkin and J. A. Taylor, "Reverse electrowetting as a new approach to high-power energy harvesting," *Nature Communications*, vol. 2, no. 448, Aug. 2011. [Online]. Available: http://www.nature.com/ncomms/journal/v2/n8/full/ncomms1454.html

[12] Y.-M. Lin, C. Dimitrakopoulos, K. A. Jenkins, D. B. Farmer, H.-Y. Chiu, A. Grill, and P. Avouris, "100-ghz transistors from wafer-scale epitaxial graphene," *Science*, vol. 327, no. 5966, p. 662, 2010. [Online]. Available: http://www.sciencemag.org/content/327/5966/662.abstract

[13] A. Edelsten, "TEGRA: Attacking Mobile Entertainment with Sword and SHIELD," July 2013, *SIGGRAPH 2013 Tech Talk*. [Online]. Available: http://www.nvidia.com/object/siggraph2013-tech-talks.html

[14] C.-I. Badoi, N. Prasad, V. Croitoru, and R. Prasad, "5g based on cognitive radio," *Wirel. Pers. Commun.*, vol. 57, no. 3, pp. 441–464, Apr. 2011. [Online]. Available: http://dx.doi.org/10.1007/s11277-010-0082-9

[15] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian, "Integrated power management for video streaming to mobile handheld devices," in *Proc. 11th ACM Int. Conf. on Multimedia*. New York, NY, USA: ACM, 2003, pp. 582–591.

[16] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proc. 6th Conf. on Computer Systems*. New York, NY, USA: ACM, 2011, pp. 301–314.

[17] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proc. 8th Int. Conf. on Mobile Systems, Applications, and Services (MobiSys '10)*. New York, NY, USA: ACM, 2010, pp. 49–62.

Appendix A An energy profile application

The following is the source code for a simple Android application that estimates energy use based on the screen brightness value and an energy profile, as discussed in Section 4.4. The source code consists of several files:

- BrightnessEnergyProfile.java The main source code file
- res/values/integers.xml Definitions of numbers
- res/values/styles.xml Definitions of UI styles for all SDK versions
- res/values/strings.xml Definitions of text constants
- res/menu/activity_brightness_energy_profile.xml Definition of menu layout
- res/values-v11/styles.xml Visual styles for Android SDK version 11 and newer
- res/layout/activity_brightness_energy_profile.xml Definition of UI layout
- res/values-v14/styles.xml Visual styles for Android SDK version 14 and newer
- AndroidManifest.xml Application description file for Android packaging

A.1 Included files

A.1.1 BrightnessEnergyProfile.java

```java
package edu.helsinki.cs.nodes.energy;

import android.os.Bundle;
import android.provider.Settings.SettingNotFoundException;
import android.app.Activity;
import android.view.Menu;
import android.view.WindowManager;
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;
import android.widget.TextView;

public class BrightnessEnergyProfile extends Activity {

    /**
     * Default
     */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_brightness_energy_profile);
        SeekBar bright = (SeekBar) findViewById(R.id.brightnessSeek);
        bright.setOnSeekBarChangeListener(new OnSeekBarChangeListener() {
            @Override
            public void onProgressChanged(SeekBar seekBar, int progress,
                    boolean fromUser) {
                // Get text field reference
                SeekBar bright = (SeekBar) findViewById(R.id.brightnessSeek);

                // Brightness, 0 to 255
                int brightnessLevel = bright.getProgress();

                // Save brightness in phone settings
                android.provider.Settings.System.putInt(getContentResolver(),
                        android.provider.Settings.System.SCREEN_BRIGHTNESS,
                        brightnessLevel);
                // Apply brightness to our app
                WindowManager.LayoutParams lp = getWindow().getAttributes();
                float brightness = 1.0f / 255 * brightnessLevel;
                if (brightness == 0)
                    brightness = 0.01f;
                lp.screenBrightness = brightness;
                getWindow().setAttributes(lp);
                updateEstimate(brightnessLevel);
                TextView brn = (TextView) findViewById(R.id.brightnessValue);
                brn.setText(brightnessLevel+"");
            }

            @Override
            public void onStartTrackingTouch(SeekBar seekBar) {
                // Unused
            }

            @Override
            public void onStopTrackingTouch(SeekBar seekBar) {
                // Unused
            }
        });
    }

    /**
     * Default
     */
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.activity_brightness_energy_profile,
                menu);
        return true;
    }

    /**
     * Show the current brightness in the text field
     */
    public void showBrightness() {
        SeekBar bright = (SeekBar) findViewById(R.id.brightnessSeek);
        int b = getScreenBrightness();
        bright.setProgress(b);
        TextView brn = (TextView) findViewById(R.id.brightnessValue);
        brn.setText(b+"");
    }

    /**
     * Get current screen brightness.
     */
    public int getScreenBrightness() {
        int screenBrightnessValue = 0;
        try {
            screenBrightnessValue = android.provider.Settings.System.getInt(
                    getContentResolver(),
                    android.provider.Settings.System.SCREEN_BRIGHTNESS);
        } catch (SettingNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return screenBrightnessValue;
    }

    /**
     * Update estimated energy use based on brightness.
     *
     * @param brightness
     *            a number from -1 to 255.
     */
    public void updateEstimate(int brightness) {
        // TODO: Use actual model here
        double s4_max = 933.48;
        double s4_150 = 766.77;
        double s4_min = 632.21;

        double estimate = 0.0;
        // Take exact values
        switch (brightness) {
        case 255:
            estimate = s4_max;
            break;
        case 150:
            estimate = s4_150;
            break;
        case 0:
            estimate = s4_min;
            break;
        default:
            break;
        }
        // Rough estimate for non-exact values
        if (estimate == 0) {
            if (brightness < 255 && brightness > 150) {
                double multiplier = (brightness - 150.0) / (255 - 150.0);
                estimate = multiplier * (s4_max - s4_150) + s4_150;
            } else if (brightness < 150 && brightness > 0) {
                double multiplier = brightness / 150.0;
                estimate = multiplier * (s4_150 - s4_min) + s4_min;
            }
        }
        // Update field on screen
        if (estimate != 0) {
            TextView est = (TextView) findViewById(R.id.estimate);
            est.setText(estimate + " mW");
        }
    }

    /*
     * Checks for brightness change on resume and updates the text field.
     *
     * @see android.app.Activity#onResume()
     */
    @Override
    protected void onResume() {
        super.onResume();
        showBrightness();
    }
}
```

A.1.2 res/values/integers.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer name="maxBrightness">255</integer>
</resources>
```

A.1.3 res/values/styles.xml

```
<resources>

    <!--
        Base application theme, dependent on API level. This theme is replaced
        by AppBaseTheme from res/values-vXX/styles.xml on newer devices.
    -->
    <style name="AppBaseTheme" parent="android:Theme.Light">
        <!--
            Theme customizations available in newer API levels can go in
            res/values-vXX/styles.xml, while customizations related to
            backward-compatibility can go here.
        -->
    </style>

    <!-- Application theme. -->
    <style name="AppTheme" parent="AppBaseTheme">
        <!-- All customizations that are NOT specific to a particular API-level can go here. -->
    </style>

</resources>
```

A.1.4 res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">BrightnessEnergyProfile</string>
    <string name="menu_settings">Settings</string>
    <string name="brn">Set Brightness Level:</string>
    <string name="mw">632.21 mW</string>
    <string name="max">255</string>
    <string name="est">Estimated power used:</string>

</resources>
```

A.1.5 res/menu/activity_brightness_energy_profile.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menu_settings"
        android:orderInCategory="100"
        android:showAsAction="never"
        android:title="@string/menu_settings"/>

</menu>
```

A.1.6 res/values-v11/styles.xml

```
<resources>

    <!--
        Base application theme for API 11+. This theme completely replaces
        AppBaseTheme from res/values/styles.xml on API 11+ devices.
    -->
    <style name="AppBaseTheme" parent="android:Theme.Holo.Light">
        <!-- API 11 theme customizations can go here. -->
    </style>

</resources>
```

A.1.7 res/layout/activity_brightness_energy_profile.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".BrightnessEnergyProfile" >

    <TextView
        android:id="@+id/brightnessLevel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_margin="10dp"
        android:text="@string/brn"
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <TextView
        android:id="@+id/estimateLegend"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:layout_margin="10dp"
        android:text="@string/est"
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <TextView
        android:id="@+id/estimate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/estimateLegend"
        android:layout_centerHorizontal="true"
        android:text="@string/mw"
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <LinearLayout
        android:id="@+id/progressLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/brightnessLevel"
        android:layout_margin="10dp" >

        <SeekBar
            android:id="@+id/brightnessSeek"
            android:layout_width="0dip"
            android:layout_height="wrap_content"
            android:layout_marginRight="10dp"
            android:layout_weight="1.0"
            android:indeterminate="false"
            android:max="@integer/maxBrightness"
            android:progress="@integer/maxBrightness" />

        <TextView
            android:id="@+id/brightnessValue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/max"
            android:textAppearance="?android:attr/textAppearanceLarge" />
    </LinearLayout>

</RelativeLayout>
```

A.1.8 res/values-v14/styles.xml

```
<resources>

    <!--
        Base application theme for API 14+. This theme completely replaces
        AppBaseTheme from BOTH res/values/styles.xml and
        res/values-v11/styles.xml on API 14+ devices.
    -->
    <style name="AppBaseTheme" parent="android:Theme.Holo.Light.DarkActionBar">
        <!-- API 14 theme customizations can go here. -->
    </style>

</resources>
```

A.1.9 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.helsinki.cs.nodes.energy"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>

    <application
        android:allowBackup="true"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".BrightnessEnergyProfile"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Index